TLP **19** (3): 412–448, 2019.
doi:10.1017/S1471068418000571 First published online 18 January 2019

Description, Implementation, and Evaluation of a Generic Design for Tabled CLP

JOAQUÍN ARIAS and MANUEL CARRO
IMDEA Software Institute, Madrid, Spain
Universidad Politécnica de Madrid, Madrid, Spain
(*e-mails:* joaquin.arias@imdea.org, manuel.carro@{imdea.org,upm.es})

submitted 10 November 2017; revised 5 November 2018; accepted 26 November 2018

Abstract

Logic programming with tabling and constraints (TCLP, *tabled constraint logic programming*) has been shown to be more expressive and in some cases more efficient than LP, CLP, or LP + tabling. Previous designs of TCLP systems did not fully use entailment to determine call/answer subsumption and did not provide a simple and well-documented interface to facilitate the integration of constraint solvers in existing tabling systems. We study the role of projection and entailment in the termination, soundness, and completeness of TCLP systems and present the design and an experimental evaluation of Mod TCLP, a framework that eases the integration of additional constraint solvers. Mod TCLP views constraint solvers as clients of the tabling system, which is generic w.r.t. the solver and only requires a clear interface from the latter. We validate our design by integrating four constraint solvers: a previously existing constraint solver for difference constraints, written in C; the standard versions of Holzbaur's CLP($\mathbb{Q}$) and CLP($\mathbb{R}$), 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 solvers. Mod TCLP is developed in Ciao Prolog, a robust, mature, next-generation Prolog system.

KEYWORDS: constraints, tabling, prolog, interface, implementation

1 Introduction

Constraint logic programming (CLP) (Jaffar and Maher 1994) extends logic programming (LP) with variables that can belong to arbitrary constraint domains and with constraint solvers that can incrementally simplify equations setup during program execution. CLP brings additional expressive power to LP, since constraints can very concisely capture complex relationships. Also, the shift from "generate-and-test" to "constrain-and-generate" code patterns reduces the search tree and, therefore, brings additional performance, even if constraint solving is in general more expensive than unification.

Tabling (Tamaki and Sato 1986; Warren 1992) is an execution strategy for logic programs which suspends repeated calls that could cause infinite loops. Answers from non-looping branches are used to resume suspended calls that 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[1] and can improve efficiency for terminating 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 (Warren *et al.* 1988; Dawson *et al.* 1996; Zou *et al.* 2005; Ramakrishna *et al.* 1997; Charatonik *et al.* 2002).

The combination of CLP and tabling (Toman 1997b; Schrijvers *et al.* 2008; Cui and Warren 2000; Chico de Guzmán *et al.* 2012) brings several advantages: it enhances termination properties, increases speed in a range of programs, and provides additional expressiveness. It has been applied in several areas, including constraint databases (Kanellakis *et al.* 1995; Toman 1997b, verification of timed automata and infinite systems (Charatonik *et al.* 2002), and abstract interpretation (Toman 1997a).

The theoretical basis of TCLP (Toman 1997b) was 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 (Schrijvers *et al.* 2008; Cui and Warren 2000) did not fully use these two operations, likely due to performance issues and also due to implementation difficulty.

On the other hand, previous TCLP frameworks featuring a more complete treatment of constraint projection and entailment (Chico de Guzmán *et al.* 2012) focused on adapting the implementation of a tabling algorithm to be used with constraints. As a result, and although the ideas therein were generic, they 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 not useful for another constraint solver; in turn, constraint solvers had to be modified in order to make them aware of internal characteristics and capabilities of the tabling algorithm. These adaptations generate a *technical debt* that made using the full potential of TCLP very difficult.

In this work, we complete previous work on conditions for termination of TCLP, we provide a richer, more flexible answer management mechanism, we generalize the design of a tabling implementation so that it can use the projection and entailment operations provided by a constraint solver presented to the tabling engine as a *server*, and 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 by extending the solver.

We have validated our design (termed Mod TCLP) with an implementation in Ciao Prolog (Hermenegildo *et al.* 2012) where we interfaced four non-trivial constraint solvers to provide four different TCLP systems. We have experimentally evaluated these implementations with several benchmarks using TCLP. Graphical step-by-step executions of TCLP programs and additional performance comparisons are provided in the Supplementary Material accompanying the paper.

[1] That is, programs which can only generate terms with a finite bound on their depth.

https://doi.org/10.1017/S1471068418000571 Published online by Cambridge University Press

```
1 dist(X, Y, D) :-
2     dist(X, Z, D1),
3     edge(Z, Y, D2),
4     D is D1 + D2.
5 dist(X, Y, D) :-
6     edge(X, Y, D).
7
8 ?- dist(a,Y,D), D < K.
```

```
1  :- use_package(clpq).
2
3  dist(X, Y, D) :-
4      D1 #> 0, D2 #> 0,
5      D  #= D1 + D2,
6      dist(X, Z, D1),
7      edge(Z, Y, D2).
8  dist(X, Y, D) :-
9      edge(X, Y, D).
10
11 ?- D #< K, dist(a,Y,D).
```

Fig. 1: Left-recursive distance traversal in a graph: Prolog (left)/CLP (right). Note: The symbols `#>` and `#=` are (in)equalities in CLP.

2 Motivation

In order to highlight some of the advantages of TCLP vs. LP, tabling, and CLP with respect to declarativeness and logical reading, we will compare how different versions of a program to compute distances between nodes in a graph behave under these three approaches. 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 rather than to differences in the code itself.

2.1 LP vs. CLP

The code in Figure 1, left, is the Prolog version of a program used to find nodes in a graph within a distance `K` from each other.[2] Figure 1, right, is the CLP version of the same code. The queries used to find the nodes `Y` from the node `a` within a maximum distance `K` appear in the figures themselves.

In the Prolog version, the distance between two nodes is calculated by adding variables `D1` and `D2`, corresponding to distances to and from an intermediate node, once they are instantiated. In the CLP version, addition is modeled as a constraint and placed at the beginning of the clause. Since the total distance is bound, this constraint is expected to prune the search in case it tries to go beyond the maximum distance `K`. These checks are not added to the Prolog version, since they would not be useful for termination: they would have to be placed after the calls to `edge/3` and `dist/3`, when it is too late to avoid infinite loops. In fact, none of the queries shown before terminates as left recursion makes the recursive clause enter an infinite loop even for acyclic graphs.

If we convert the program to a right-recursive version by swapping the calls to `edge/3` and `dist/3` (Figure 2), the LP execution will still not terminate in a cyclic graph. The right-recursive version of the CLP program will, however, finish because the initial bound to the distance eventually causes the constraint store to become inconsistent, which

[2] This is a typical query for the analysis of social networks (Swift and Warren 2010).

https://doi.org/10.1017/S1471068418000571 Published online by Cambridge University Press

Table 1: *Comparison of termination properties*

	LP	CLP	TAB	TCLP	Graph
Left recursion	×	×	✓	✓	Without cycles
Right recursion	✓	✓	✓	✓	
Left recursion	×	×	×	✓	With cycles
Right recursion	×	✓	×	✓	

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

Fig. 2: Right-recursive distance traversal in a graph: Prolog (left)/CLP (right). Note: The symbols `#>` and `#=` are (in)equalities in CLP.

provokes a failure in the search. This behavior is summarized in columns "LP" and "CLP" of Table 1.

Note that this transformation is easy in this case, but in other cases, such as language interpreters or tree/graph traversal algorithms, left (or double) recursion is much more natural. While there are techniques to remove left/double recursion, most Prolog compilers do not feature them. Therefore, we assume that the original source code is straightforwardly mapped to the low-level runtime system, and, if necessary, left/double recursion has to be manually removed by adding extra arguments implementing, for example, explicit stacks – precisely the kind of manual program transformation that we would like to avoid due to the difficulties that it brings with respect to maintenance and clarity.

2.2 LP vs. tabling

Tabling records the first occurrence of each call to a tabled predicate (the *generator*) and its answers. In variant tabling, the most usual form of tabling, when a call equal up to variable renaming to a previous generator is found (a variant), its execution is suspended, and it is marked as a *consumer* of the generator. For example, `dist(a,Y,D)` is a variant of `dist(a,Z,D)` if `Y` and `Z` are free variables. When a generator finitely finishes exploring all of its clauses and its answers are collected, its consumers are resumed and are fed the answers of the generator. This may make consumers produce new answers that will in turn cause more resumptions.

Tabling is a complete strategy for all programs with the bounded term-depth property, which in turn implies that the Herbrand model is finite. Therefore, left- or right-recursive

https://doi.org/10.1017/S1471068418000571 Published online by Cambridge University Press

reachability terminates in finite graphs with or without cycles. However, the program in Figure 1, left, has an infinite minimum Herbrand model for cyclic graphs: every cycle can be traversed an unbounded number of times, giving rise to an unlimited number of answers with a different distance each. The query `?- dist(a,Y,D), D < K.` will, therefore, not terminate under variant tabling.

2.3 TCLP vs. tabling and CLP

The program in Figure 1, right, can be executed with tabling and using constraint entailment to suspend calls which are more particular than previous calls and, symmetrically, to keep only the most general answers returned.

Entailment can be seen as a generalization of subsumption for the case of general constraints; in turn, subsumption was shown to enhance termination and performance in tabling (Swift and Warren 2010). For example, the goal $G_0 \equiv$ `dist(a,Y,D)` is subsumed by $G_1 \equiv$ `dist(X,Y,D)` because the former is an instance of the latter ($G_0 \sqsubseteq G_1$). All the answers for G_1 where $\texttt{X} = \texttt{a}$ are valid answers for G_0; on the other hand, all the answers for G_0 are also answers for G_1.

The main idea behind the use of entailment in TCLP is that more particular calls (consumers) can suspend and later reuse the answers collected by more general calls (generators). In order to make this entailment relationship explicit, we define a TCLP goal as $\langle g, c_g \rangle$ where g is the call (a literal) and c_g is the projection of the current constraint store onto the variables of the call. Then, $\langle \texttt{dist(a,Y,D)}, \texttt{D} < 150 \rangle$ is entailed by the goal $\langle \texttt{dist(a,Y,D)}, \texttt{D} > 0 \wedge \texttt{D} < 75 \rangle$ because $\texttt{D} > 0 \wedge \texttt{D} < 75 \sqsubseteq \texttt{D} < 150$. We also say that the former (the generator) is more general than the latter (the consumer). All the solutions of the consumer are solutions of the generator or, in other words, the space of solutions of the consumer is a subset of that of the generator. However, not all the answers from a generator are valid for its consumers. For example, $\texttt{Y} = \texttt{b} \wedge \texttt{D} > 125 \wedge \texttt{D} < 135$ is a solution for our generator, but not for our consumer, since the consumer call was made under a constraint store more restrictive than the generator. Therefore, the tabling engine should check and filter, via the constraint solver, that the answer from the generator is consistent w.r.t. the constraint store of the consumer.

The use of entailment in calls and answers enhances termination properties and can also increase speed (Section 6.1). The column "TCLP" in 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. Our TCLP framework not only facilitates the integration of constraint solvers with the tabling engine thanks to its simple interface (Section 4.1) but also minimizes the effort required to execute existing CLP programs under tabling (Figure 5), since the changes required to the source code are minimal.

3 Background

In this section we present the syntax and semantics of constraint logic programs (Section 3.1) and extend the semantics and termination, soundness, and completeness proofs of Toman (1997b) for a TCLP top-down execution (Sections 3.2 and 3.3, resp.).

https://doi.org/10.1017/S1471068418000571 Published online by Cambridge University Press

3.1 Constraint logic programs

Constraint logic programming (Jaffar and Maher 1994) introduces constraint solving methods in logic-based systems. A constraint logic program consists of clauses of the form:

$$h \text{ :- } c_h, l_1, \ \ldots, \ l_k.$$

where h is an atom, c_h is a constraint, and l_i are literals. The head of the clause is h and the rest is called the body. The clauses where the body is always true, h :- *true*, are called facts and usually written omitting the body (h.). We will use L to denote the set of l_i in a clause. We will assume throughout this paper that the program has been rewritten so that clause heads are linearized (all the variables are different) and all head unifications take place in c_h. We will assume that we are dealing with *definite programs*, that is, programs where the literals in the body are always positive (non-negated) atoms. *Normal programs* require a different treatment.

A query to a CLP program is a clause without head ?- $c_q, \ q_1, \ \ldots, \ q_k$, where c_q is a constraint and q_i are the literals in the query. We denote the set of q_i as Q.

During the evaluation of a query to a CLP program, the *inference engine* generates constraints whose consistency with respect to the current constraint store is checked by the *constraint solver*. If the check fails, the engine backtracks to a previous state and takes a pending branch of the search tree.

A constraint solver, denoted by CLP($\mathcal{D}$), is a (partial) executable implementation of a constraint domain $\mathcal{D}$. A valuation v over a set of variables $S = \{X_1, \ldots, X_n\}$ maps each variable X_i to a value d_i in $\mathcal{D}$, denoted by $v(X_i) = d_i$. v is extended to expressions by substituting the variables in the expressions by the value they are mapped onto. A constraint can be a singleton constraint or a conjunction of simpler constraints. We denote constraints with lowercase letters and sets of constraints with uppercase letters. A solution of a constraint c is a valuation v over the variables occurring in c if $v(c)$ holds in the constraint domain.

The minimal set of operations that we expect a constraint solver to support in order to interface it successfully with our tabling system is as follows:

- Test for consistence or satisfiability: A constraint c is consistent in the domain $\mathcal{D}$, denoted $\mathcal{D} \vDash c$, if c has a solution in $\mathcal{D}$.
- Test for entailment ($\sqsubseteq_{\mathcal{D}}$):[3] We say that a constraint c_0 entails another constraint c_1 ($c_0 \sqsubseteq_{\mathcal{D}} c_1$) if any solution of c_0 is also a solution of c_1. We extend the notion of constraint entailment to a set of constraints: a set of constraints C_0 entails another set of constraints C_1 (and we write it as $C_0 \sqsubseteq_{\mathcal{D}} C_1$) if $\forall c_i \in C_0 \, \exists c_j \in C_1 . c_i \sqsubseteq_{\mathcal{D}} c_j$.
- An operation to compute the projection of a constraint c onto a set of variables S to obtain a constraint c_S involving only variables in S such that any solution of c is also a solution of c_S, and a valuation v over S that is a solution of c_S is a partial solution of c (i.e., there exists an extension of v that is a solution of c). We denote the projection as $Proj(S, c)$.

[3] We may omit the subscript $\mathcal{D}$ if there is no ambiguity.

https://doi.org/10.1017/S1471068418000571 Published online by Cambridge University Press

3.2 Semantics of CLP and TCLP

The CLP fixpoint S-semantics (Falaschi *et al.* 1989; Toman 1997b) is defined as usual as the least fixpoint of the immediate consequence operators $S_P^{\mathcal{D}}$ where all the operations behave as defined in the constraint domain $\mathcal{D}$:

Definition 1 (*Operator* $S_P^{\mathcal{D}}$ *(Falaschi et al. 1989; Toman 1997b)*)
Let P be a CLP program and I an interpretation. The immediate consequence operator $S_P^{\mathcal{D}}$ is defined as:

$$S_P^{\mathcal{D}}(I) = I \cup \{ \langle h, c\rangle \mid \begin{array}{l} h \text{ :- } c_h, l_1, \ldots, l_k \text{ is a clause of } P, \\ \langle a_i, c_i\rangle \in I,\ 0 < i \leq k, \\ c' = Proj(vars(h),\ c_h \wedge \bigwedge_{i=1}^{k}(a_i = l_i \wedge c_i)), \\ \mathcal{D} \vDash c', \\ \text{if } c' \sqsubseteq c'' \text{ for some } \langle h, c''\rangle \in I \text{ then } c = c'' \text{ else } c = c' \ \} \end{array}$$

Note that $S_P^{\mathcal{D}}$ may not add a pair $\langle literal, constraint\rangle$ when a more general constraint is already present in the interpretation being enlarged. However, to guarantee monotonicity, it does not remove existing, more particular constraints. The operational semantics of TCLP will however be able to do that.

The operational semantics of TCLP extends that of CLP programs under a top-down execution scheme (Jaffar and Maher 1994) that is defined in terms of a transition system between states:

Definition 2
A *state* is a tuple $\langle R, c\rangle$ where:

- R, the resolvent, is a multiset of literals and constraints that contains the collection of as-yet-unseen literals and constraints of the program. For brevity, when the set is a singleton, we will write its only element using a lowercase letter instead of an uppercase letter, for example, t instead of $\{t\}$.
- c, the constraint store, is a (possibly empty) conjunction of constraints. It is acted upon by the constraint solver.

In Jaffar and Maher (1994), the constraint store c is divided in a collection of *awake* constraints and a collection of *asleep* constraints. This separation is ultimately motivated by implementation issues. We do not need to make that distinction here.

Given a query $\langle Q, c_q\rangle$, the initial state of the evaluation is $\langle Q, c_q\rangle$. Every transition step between states resolves literals of the resolvent against the clauses of the program and adds constraints to the constraint store. A derivation is *successful* if it is finite, and the final state has the form $\langle \emptyset, c\rangle$ (i.e., the resolvent becomes empty). The answer to the query corresponding to this derivation is $c' = Proj(vars(Q), c)$.

The transitions due to constraint handling are deterministic (there is only one possible descendant in every node), while the transitions due to literal matching are nondeterministic (there are as many descendants as clauses match with the node literal). We denote the set of tabled predicates in a TCLP program P by Tab_P. The set of generators (calls to tabled predicates that do not entail previous calls) is denoted by Gen_P. The evaluation of a query to a TCLP program is usually represented as a forest of search trees, where each search tree corresponds to the evaluation of a generator and where its nodes

https://doi.org/10.1017/S1471068418000571 Published online by Cambridge University Press

are the states generated during the evaluation (see Appendix A of the Supplementary Material for examples).

The order in which the literals/constraints are selected is decided by the computation rule. During the computation of a TCLP program, two main phases are interleaved for the evaluation of every tabled goal: the call entailment phase (Definition 3) and the answer entailment phase (Definition 4).

Definition 3 (*Call entailment phase*)
The call entailment phase checks if a new goal $\langle t, c\rangle$, where t is a tabled literal (i.e., $t \in Tab_P$), entails a previous goal (called its generator) or it is a new generator itself.[4] In both the cases, $\langle t, c\rangle$ is resolved by answer resolution consuming the answers c_i such that

- Either $c_i \in Ans(g, c_g)$, where $Ans(g, c_g)$ is the set of answers of the oldest generator $\langle g, t\rangle \in Gen_P$, such that g and t are equal upon variable renaming, and $c \wedge (t = g) \sqsubseteq_{\mathcal{D}} c_g$, where $t = g$ is an abbreviation for the conjunction of equations between the corresponding arguments of t and g, that is, $\langle g, c_g\rangle$ is more general than $\langle t, c\rangle$. In this case, the goal $\langle t, c\rangle$ is marked as a consumer of $\langle g, c_g\rangle$.
- Or $c_i \in Ans(t, c')$, where $c' = Proj(vars(t), c)$, and $Ans(t, c')$ is the set of answers of a new generator $\langle t, c'\rangle$ that is added to Gen_P, the set of generators.

In TCLP, goals that match heads of tabled predicates are not resolved against program clauses. Instead, they are resolved consuming the answer constraints from a generator; this is termed *answer resolution*.

Definition 4 (*Answer entailment phase*)
The answer constraints of a generator $\langle g, c_g\rangle$ are collected in the answer entailment phase in such a way that an answer which entails another more general answer is discarded. Let $\langle \emptyset, c_i\rangle$ and $\langle \emptyset, c_j\rangle$ be (different) final states of successful derivations of $\langle g, c_g\rangle$. Then, the set of answers of $\langle g, c_g\rangle$, denoted by $Ans(g, c_g)$, is the set of more general (w.r.t. $\sqsubseteq_{\mathcal{D}}$) answer constraints c'_i obtained as the projection of c_i onto $vars(g)$:

$$Ans(g, c_g) = \{c'_i \mid \begin{array}{l} c'_i = Proj(vars(g), c_i), \\ \forall j \neq i,\ \nexists\ c'_j = Proj(vars(g), c_j).\ \ c'_i \sqsubseteq_{\mathcal{D}} c'_j\ \} \end{array}$$

We will assume, without loss of generality, that the subscripts i, j correspond to the order in which answers are found. The answer management strategy used in the answer entailment phase aims at keeping only the more general answers by discarding/removing more particular answers. This is specified by the quantification $\forall j \neq i$, where i and j are the indexes of the final constraint store. Simpler answer management strategies are possible: the implementations in Cui and Warren (2000) and Chico de Guzmán *et al.* (2012), following Toman (1997b), only discard answers that are more particular than a previous one, that is, they implement $\forall j < i$, and keep previously saved answers. A third possibility is to remove previous answers that are more particular than new ones, implementing $\forall j > i$. The choice among them does not impact soundness or completeness properties. However, discarding and removing redundant answers can greatly increase the efficiency of the implementation, as we experimentally show in Section 6.3.

[4] Note that this entailment check includes subsumption in the Herbrand domain.

https://doi.org/10.1017/S1471068418000571 Published online by Cambridge University Press

The order in which we search in the TCLP forest for a previous generator during the call entailment phase does not impact the completeness, soundness, or termination properties of the execution, but it can change its efficiency. Let us discuss two particular orders: from older to younger generators or the other way around. In both the cases, in order to determine whether a call is a consumer, we need to find a previous generator call. Younger generators can in general be expected to be more general than older generators, since new (tabled) calls are flagged as generators only when they are more general than all previous calls. Starting at older, less general generators may need to examine several generators and perform potentially expensive entailment checks before finding one that suits the needs of the call. On the other hand, starting at younger, more general generators should allow us to locate a suitable generator faster. However, these more general generators will likely have more answers than what less general generators have, and therefore, more filtering/selection would be necessary. Thus, there does not seem to be a clear best strategy: either more generators have to be traversed or more answers have to be filtered.

3.3 Soundness, completeness and termination properties of TCLP

Toman (1997b) proves soundness and completeness of SLG^C (SLG (Chen and Warren 1996) with constraints) for CLP Datalog programs by reduction to soundness and completeness of bottom-up evaluation. It is possible to extend these results to prove soundness and completeness of our proposal: they only differ in the answer management strategy and the construction of the TCLP forest. The strategy used in SLG^C only discards answers that are more particular than a previous answer, while in our proposal, we in addition remove previously existing more particular answers (Definition 4). The result of this is that only the most general answers are kept. In SLG^C, the generation of the forest is modeled as the application of rewriting rules, while in TCLP, it is defined in terms of a transition system.

Theorem 1 (*Soundness w.r.t. the fixpoint semantics*)
Let P be a TCLP definite program and $\langle q, c_q\rangle$ a query. Then for any answer c'

$$c' \in Ans(q, c_q) \;\Rightarrow \exists \langle q, c\rangle \in lfp(S_P^{\mathcal{D}}(\emptyset)).\; c' = c_q \wedge c$$

That is, all the answers derived from the forest construction are also derived from the bottom-up computation (and are therefore correct).

Answer resolution recovers constraints from the $Ans(g, c_g)$ sets corresponding to a goal g and a constraint c_g, instead of repeating Selective Linear Definite (SLD) resolution on $\langle g, c_g\rangle$. These sets were ultimately generated by saving SLD resolution results, possibly using previously generated sets of answers for intermediate goals.[5] Therefore, we can substitute any point where answers are recovered from $Ans(g, c_g)$ for the corresponding SLD resolution: if $c' \in Ans(q, c_q)$, then there is an SLD derivation $\langle q, c_q\rangle \rightsquigarrow \langle \emptyset, c'\rangle$. Moreover, if $\langle q, \emptyset\rangle \rightsquigarrow \langle \emptyset, c\rangle$, then $c' = c \wedge c_q$, and we can construct the answer to $\langle q, c_q\rangle$ from that of $\langle q, \emptyset\rangle$. So, if $c' \in Ans(q, c_q)$, then there is $\langle q, \emptyset\rangle \rightsquigarrow \langle \emptyset, c\rangle$ and $c' = c \wedge c_q$. From

[5] Answers more particular than other answers may be removed, but those which remain are still *correct* answers.

https://doi.org/10.1017/S1471068418000571 Published online by Cambridge University Press

the correctness of SLD resolution, we have that if there is a derivation $\langle q, \emptyset\rangle \rightsquigarrow \langle \emptyset, c\rangle$, then $\langle q, c\rangle \in lfp(S_P^{\mathcal{D}}(\emptyset))$ and, as we said before, $c' = c \wedge c_q$.

Theorem 2 (*Completeness w.r.t. the fixpoint semantics*)
Let P be a TCLP definite program and $\langle h, true\rangle$ a query. Then, for every $\langle h, c\rangle$ in $lfp(S_P^{\mathcal{D}})$

$$\langle h, c\rangle \in lfp(S_P^{\mathcal{D}}(\emptyset)) \;\Rightarrow\; \exists c' \in Ans(h, true).\ c \sqsubseteq c'$$

That is, all the answers derived from the bottom-up computation entail answers generated by the TCLP execution.

For any answer derived from the bottom-up computation $\langle h, c\rangle \in lfp(S_P^{\mathcal{D}}(\emptyset))$, there exists a successful SLD derivation $\langle h, true\rangle \rightsquigarrow \langle \emptyset, c\rangle$. Since answer resolution may keep the most general answer when generating comparable answers (Definition 4), it is also complete if entailment with this most general answer is used instead of equality with the more particular answers (which were removed). Therefore, it will always be the case that $\exists c' \in Ans(h, true).\ c \sqsubseteq c'$.

Termination of TCLP Datalog programs under a top-down strategy when the domain is constraint-compact (Definition 5) is proven in Toman (1997b).

Definition 5 (*Constraint-compact*)
Let $\mathcal{D}$ be a constraint domain, and D the set of all constraints expressible in $\mathcal{D}$. Then $\mathcal{D}$ is constraint-compact iff
- for every finite set of variables S and
- for every subset $C \subseteq D$ such that $\forall c \in C. vars(c) \subseteq S$,
  there is a finite subset $C_{fin} \subseteq C$ such that $\forall c \in C. \exists c' \in C_{fin}. c \sqsubseteq_{\mathcal{D}} c'$

In that case, the evaluation will suspend the exploration of a call whose constraint store is less general than or comparable to a previous call. Eventually, the program will generate a set of call constraint stores which can cover any (potentially infinite) set of constraints in the domain that use a finite set of variables, therefore finishing evaluation. That is because, intuitively speaking, a domain $\mathcal{D}$ is constraint-compact if for any (potentially infinite) set of constraints C expressible in $\mathcal{D}$, there is a *finite* set of constraints $C_{fin} \subseteq C$ that covers (in the sense of $\sqsubseteq_{\mathcal{D}}$) C. In other words, C_{fin} is as general as C.

Many TCLP applications use domains that are not constraint-compact because constraint-compact domains are not very expressive. Therefore, we refined the termination theorem (Theorem 23 in Toman (1997b)) for Datalog programs with constraint-compact domains to cover cases where a program uses a non-constraint compact domain but, during its evaluation, it generates only a constraint-compact subset of all the constraints that are expressible in the domain.

Theorem 3 (*Termination*)
Let P be a TCLP($\mathcal{D}$) definite program and $\langle Q, c_q\rangle$ a query. Then, the TCLP execution terminates iff
- For every literal g, the set C_g is constraint-compact, where C_g is the set of all the constraint stores c_i, projected and renamed w.r.t. the arguments of g, s.t. $\langle g, c_i\rangle$ is in the forest $\mathcal{F}(Q, c_q)$.
- For every goal $\langle g, c_g\rangle$, the set $A_{\langle g, c_g\rangle}$ is constraint-compact, where $A_{\langle g, c_g\rangle}$ is the set of all the answer constraints c', projected and renamed w.r.t. the arguments of g, s.t. c' is a successful derivation in the forest $\mathcal{F}(Q, c_q)$.

https://doi.org/10.1017/S1471068418000571 Published online by Cambridge University Press

```
1 nat(X) :-
2     X #= Y+1,
3     nat(Y).
4 nat(0).
```

```
1 nat(X) :-
2     X #= Y+1,
3     nat(Y).
4 nat(0).
5 nat(X) :- X #> 1000.
```

Fig. 3: Left: definition of natural numbers in TCLP(ℚ). Right: natural numbers with a clause describing infinitely many numbers.

The intuition is that for every subset C of the set of all possible constraint stores C_g that can be generated when evaluating a call to P, if there is a finite subset $C_{fin} \subseteq C$ that covers (i.e., is as general as) C, then, at some point, any call will entail all previous calls, thereby allowing its suspension to avoid loops. Similarly, for every subset A from the set of all possible answer constraints $A_{\langle g,c_g\rangle}$ that can be generated by a call, if there is a finite subset $A_{fin} \subseteq A$ that covers A, then, at some point, any answer will entail a previous one, ensuring that the class of answers $Ans(g, c_g)$ which is entailed by any other possible answer returned by the program is finite.[6]

Example 1
Figure 3, left, shows a program which generates all the natural numbers using TCLP(ℚ). Although CLP(ℚ) is not constraint-compact, the constraint stores generated by that program for the query `?- X #< 10, nat(X)` are constraint-compact and the program finitely finishes. Let us look at its behavior from two points of view:

Compactness of call/answer constraint sets The set of all active constraint stores generated for the predicate `nat/1` under the query $\langle \mathtt{nat(X)}, \mathtt{X} < 10\rangle$ is $C_{\mathtt{nat(V)}} = \{\mathtt{V} < 10, \mathtt{V} < 9, \ldots, \mathtt{V} < -1, \mathtt{V} < -2, \ldots\}$. It is constraint-compact because every subset $C \in C_{\mathtt{nat(V)}}$ is covered by $C_{fin} = \{\mathtt{V} < 10\}$. The set of all possible answer constraints for the query, $A_{\langle \mathtt{nat(V)}, \mathtt{V<10}\rangle} = \{\mathtt{V} = 0, \ldots, \mathtt{V} = 9\}$, is also constraint-compact because it is finite. Therefore, the program terminates.
Suspension due to entailment The first recursive call is $\langle \mathtt{nat(Y_1)}, \mathtt{X} < 10 \wedge \mathtt{X} = \mathtt{Y_1} + 1\rangle$, and the projection of its constraint store after renaming entails the initial one since $\mathtt{V} < 9 \sqsubseteq \mathtt{V} < 10$. Therefore, TCLP evaluation suspends in the recursive call, shifts execution to the second clause, and generates the answer $\mathtt{X} = 0$. This answer is given to the recursive call, which was suspended, produces the constraint store $\mathtt{X} < 10 \wedge \mathtt{X} = \mathtt{Y_1} + 1 \wedge \mathtt{Y_1} = 0$, and generates the answer $\mathtt{X} = 1$. Each new answer $\mathtt{X_n} = \mathtt{n}$ is used to feed the recursive call. When the answer $\mathtt{X} = 9$ is given, it results in the (inconsistent) constraint store $\mathtt{X} < 10 \wedge \mathtt{X} = \mathtt{Y_1} + 1 \wedge \mathtt{Y_1} = 9$ and the execution terminates.

Example 2
The program in Figure 3, left, does not terminate for the query `?- X #> 0, X #< 10, nat(X)`. Let us examine its behavior:

The constraint store sets are not compact The set of all constraint stores generated by the query $\langle \mathtt{nat(X)}, \mathtt{X} > 0 \wedge \mathtt{X} < 10\rangle$ is $C_{\mathtt{nat(V)}} = \{\mathtt{V} > 0 \wedge \mathtt{V} < 10,$

[6] Note that a finite answer set does not imply a finite domain for the answers: the set of answers $Ans(Q, c_q) = \{V > 5\}$ if finite, but the domain of V is infinite.

https://doi.org/10.1017/S1471068418000571 Published online by Cambridge University Press

$\mathtt{V} > -1 \wedge \mathtt{V} < 9, \ldots, \mathtt{V} > -\mathtt{n} \wedge \mathtt{V} < (10 - \mathtt{n}), \ldots\}$, which it is not constraint-compact. Note that V is, in successive calls, restricted to a sliding interval $[\mathtt{k}, \mathtt{k} + 10]$ which starts at $\mathtt{k} = 0$ and decreases k in each recursive call. No finite set of intervals can cover any subset of the possible intervals.

The evaluation loops The first recursive call is $\langle \mathtt{nat}(\mathtt{Y}_1), \mathtt{X} > 0 \wedge \mathtt{X} < 10 \wedge \mathtt{X} = \mathtt{Y}_1 + 1 \rangle$ and the projection of its constraint store does not entail the initial one after renaming since $(\mathtt{V} > -1 \wedge \mathtt{V} < 9) \not\sqsubseteq (\mathtt{X} > 0 \wedge \mathtt{X} < 10)$. Then, this call is evaluated and produces the second recursive call, $\langle \mathtt{nat}(\mathtt{Y}_2), \mathtt{X} > 0 \wedge \mathtt{X} < 10 \wedge \mathtt{X} = \mathtt{Y}_1 + 1 \wedge \mathtt{Y}_1 = \mathtt{Y}_2 + 1 \rangle$. Again, the projection of its constraint store, $\mathtt{Y}_2 > -2 \wedge \mathtt{Y}_2 < 8$, does not entail any of the previous constraint stores, and so on. The evaluation therefore loops.

Example 3

The program in Figure 3, left, also does not terminate for the query `?- nat(X)`. Let us examine its behavior:

The answer constraint set is not compact The equation in the body of the clause $\mathtt{X} = \mathtt{Y}_1 + 1$ defines a relation between the variables, but, since the domain of X is unrestricted, its projection onto $\mathtt{Y}_1$ will return no constraints (i.e., $Proj(\mathtt{Y}_1, \mathtt{X} = \mathtt{Y}_1 + 1) = \mathtt{true}$). Therefore, the set of all call constraint stores generated by the query $\langle \mathtt{nat}(\mathtt{X}), \mathtt{true} \rangle$ is $C_{\mathtt{nat(V)}} = \{\mathtt{true}\}$ which is finite and constraint-compact. However, the answer constraint set $A_{\langle \mathtt{nat(V)}, \mathtt{true} \rangle} = \{\mathtt{V} = 0, \mathtt{V} = 1, \ldots, \mathtt{V} = \mathtt{n}, \ldots\}$ is not constraint-compact.

Call suspension with an infinite answer constraint set The first recursive call is $\langle \mathtt{nat}(\mathtt{Y}_1), \mathtt{X} = \mathtt{Y}_1 + 1 \rangle$, and the projection of its constraint store entails the initial store. Therefore, the TCLP evaluation suspends the recursive call, shifts execution to the second clause, and generates the answer $\mathtt{X} = 0$. This answer is used to feed the suspended recursive call, resulting in the constraint store $\mathtt{X} = \mathtt{Y}_1 + 1 \wedge \mathtt{Y}_1 = 0$ which generates the answer $\mathtt{X} = 1$. Each new answer $\mathtt{X} = \mathtt{n}$ is used to feed the suspended recursive call. Since the projection of the constraint stores on the call variables is `true`, the execution tries to generate infinitely many natural numbers. Therefore, the program does not terminate.

Example 4

Unlike the situation that happens in pure Prolog/variant tabling, adding new clauses to a program under TCLP can make it terminate.[7] As an example, Figure 3, right, is the same as Figure 3, left, with the addition of the clause `nat(X):- X #> 1000`. Let us examine its behavior under the query `?- nat(X)`:

Compactness of call/answer constraint sets The set of all constraint stores generated remains $C_{\mathtt{nat(V)}} = \{\mathtt{true}\}$. But, the new clause makes the answer constraint set become $A_{\langle \mathtt{nat(V)}, \mathtt{true} \rangle} = \{\mathtt{V} = 0, \mathtt{V} = 1, \ldots, \mathtt{V} = \mathtt{n}, \ldots, \mathtt{V} > 1000, \mathtt{V} > 1001, \ldots, \mathtt{V} > \mathtt{n}, \ldots\}$, for unbound values of n, which is constraint-compact

[7] This depends on the strategy used by the TCLP engine to resume suspended goals. Our implementation gathers all the answers for goals that can produce results first, and then, these answers are used to feed suspended goals. This makes the exploration of the forests proceed in a breadth-first fashion.

https://doi.org/10.1017/S1471068418000571 Published online by Cambridge University Press

because a constraint of the form $\texttt{V} > \texttt{n}$ is entailed by infinitely many constraints, that is, it covers the infinite set $\{\texttt{V} = \texttt{n}+1, \ldots, \texttt{V} > \texttt{n}+1, \ldots\}$. Therefore, since both sets are constraint-compact, the program terminates.

First search, then consume The first recursive call $\langle \texttt{nat}(\texttt{Y}_1), \texttt{X} = \texttt{Y}_1 + 1\rangle$ is suspended and the TCLP evaluation shifts to the second clause, which generates the answer $\texttt{X} = 0$. Then, instead of feeding the suspended call, the evaluation continues the search and shifts to the third clause, `nat(X):- X #> 1000`, and generates the answer $\texttt{X} > 1000$. Since no more clauses remain to be explored, the answer $\texttt{X} = 0$ is used, generating $\texttt{X} = 1$. Then $\texttt{X} > 1000$ is used, resulting in the constraint store $\texttt{X} = \texttt{Y}_1 + 1 \wedge \texttt{Y}_1 > 1000$, which generates the answer $\texttt{X} > 1001$. However, during the answer entailment phase, $\texttt{X} > 1001$ is discarded because $\texttt{X} > 1001 \sqsubseteq \texttt{X} > 1000$. Then, one by one, each answer $\texttt{X} = \texttt{n}$ is used, generating $\texttt{X} = \texttt{n} + 1$. But, when the answer $\texttt{X} = 1000$ is used, the resulting answer $\texttt{X} = 1001$ is discarded, during the answer entailment phase because $\texttt{X} = 1001 \sqsubseteq \texttt{X} > 1000$. At this point, the evaluation terminates because there are no more answers to be consumed. The resulting set of answers is $Ans(\texttt{nat(X)}, \texttt{true}) = \{\texttt{X} = 0, \texttt{X} > 1000, \texttt{X} = 1, \ldots, \texttt{X} = 1000\}$.

4 The *Mod TCLP* framework

In this section, we describe the Mod TCLP framework, the operations required by the interface, and the program transformation that we use to compile programs with tabled constraints (Section 4.1). We also provide a sketch of its implementation, and we describe step-by-step some executions at the level of the TCLP libraries (Section 4.2 and Appendix B of the supplementary material). In Section 4.3, we present the implementation of the TCLP interface for Holzbaur's CLP($\mathbb{Q}$) solver, and in Section 4.4, we present an optimization, the *Two-Step* projection.

4.1 Design of the generic interface

Mod TCLP provides a generic interface (Figure 4) designed to facilitate the integration of different constraint solvers. The predicates of the interface use extensively two objects: `Vars`, the list of constrained variables, provided by the tabling engine to the constraint solver, and `ProjStore`, a representation of the projected constraint store, opaque to the tabling engine, and which should be self-contained and independent (e.g., with fresh variables) from the *main* constraint store. For example, the constraint solver CLP($\mathbb{D}_{\leq}$) (Section 5.1) is written in C, and the projection of a constraint store is a C structure whose representation is its memory address and length.

To implement these predicates, the constraint solver has to support the (minimal) set of operations defined in Section 3.1: projection, test for entailment, and test for consistency. The predicates that the constraint solver must provide in order to make its interaction with the tabling engine possible are

- `store_projection(+Vars, -ProjStore)`, which is invoked before the call and the answer entailment phases:
  - It is used before the call entailment phase to generate the representation of the goal as a tuple ⟨`G`, `ProjStore`⟩, where `ProjStore` represents the projection of

https://doi.org/10.1017/S1471068418000571 Published online by Cambridge University Press

`store_projection(+Vars, -ProjStore)` Returns in `ProjStore` a representation of the projection of the current constraint store onto the list of variables `Vars`.

`call_entail(+ProjStore, +ProjStore`$_{\texttt{gen}}$`)` Succeeds if the projection of the current constraint store, `ProjStore`, entails the projected store, `ProjStore`$_{\texttt{gen}}$, of a previous generator. It fails otherwise.

`answer_compare(+ProjStore, the projected store of the current answer)` , `ProjStore`, entails the projected store of a previous answer, `ProjStore`$_{\texttt{ans}}$, or `Res='>'` if `ProjStore` is entailed by `ProjStore`$_{\texttt{ans}}$ and they are not equal. It fails otherwise.

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

Fig. 4: Generic interface specification.

the constraint store at the moment of the call onto `Vars`, the variables in `G`. Although a generic implementation should include the Herbrand constraints of the call in the constraint store, our implementation does not consider Herbrand constraints to be part of the constraint store by default. Instead, calls are syntactically compared using variant checking, but the programmer can also choose to use subsumption, if required, by using a package described below. There are some reasons for that decision: on the one hand, programmers (even using tabling) are used to this behavior; on the other hand, there are data structures highly optimized (Ramakrishnan *et al.* 1995) to save and retrieve calls together with their input/output substitutions which perform variant checking on the fly while taking advantage of the Warren Abstract Machine (WAM)-level representation of substitutions.

– Similarly, before the answer entailment phase, the projection of the Herbrand constraints onto the variables of the goal is directly taken care of by their WAM-level representation. We use variant checking to detect when the Herbrand constraints associated to two calls are equal. Therefore, an answer constraint is internally represented by a tuple $\langle$`S`, `ProjStore`$\rangle$ where `S` captures the Herbrand constraints of the variables of the goal and `ProjStore` represents the projection of the rest of the answer constraint onto `Vars`, the variables of the answer.

- `call_entail(+ProjStore, +ProjStore`$_{\texttt{gen}}$`)` is invoked during the call entailment phase to check if a new call, represented by $\langle$`G`, `ProjStore`$\rangle$, entails a previous generator, represented by $\langle$`G`$_{\texttt{gen}}$, `ProjStore`$_{\texttt{gen}}\rangle$, where `G` is a variant of `G`$_{\texttt{gen}}$. The predicate succeeds if `ProjStore` $\sqsubseteq$ `ProjStore`$_{\texttt{gen}}$ and fails otherwise. If Herbrand subsumption checking is needed, our implementation provides a package which transforms calls to tabled predicates so that suspension is based on entailment in $\mathbb{H}$. This transformation moves Herbrand constraint handling away from the level of the WAM by creating attributed variables (Cui and Warren 2000) that carry

https://doi.org/10.1017/S1471068418000571 Published online by Cambridge University Press

the constraints – that is, the unifications. Later on, a Herbrand constraint solver is used to check subsumption.[8]

- `answer_compare(+ProjStore, +ProjStore`$_{\texttt{ans}}$`, -Res)` is invoked during the answer entailment phase to check a new answer, represented by $\langle$`S, ProjStore`$\rangle$, against a previous one, represented by $\langle$`S`$_{\texttt{ans}}$`, ProjStore`$_{\texttt{ans}}\rangle$, when the Herbrand constraints `S` and `S`$_{\texttt{ans}}$ are equal. The predicate compares `ProjStore` and `ProjStore`$_{\texttt{ans}}$ and returns `âĂŸ=<'` in its last argument when `ProjStore` $\sqsubseteq$ `ProjStore`$_{\texttt{ans}}$, '`>`' when `ProjStore` $\sqsupset$ `ProjStore`$_{\texttt{ans}}$, and fails otherwise. This bidirectional entailment check, which is used to discard/remove more particular answers, is a potentially costly process, but it brings considerable advantage from saved resumptions (Section 6.3): when an answer is added to a generator, consumers are resumed by that answer. These consumers in turn generate more answers and cause further resumptions in cascade. Reducing the number of redundant answers reduces the number of redundant resumptions, and we have experimentally observed that it results in important savings in execution time.
- `apply_answer(+Vars, +ProjStore)` is invoked to consume an answer from a generator. In variant tabling, since consumers are variants of generators, answer substitutions from generators can always be applied to consumers. That is not the case when using entailment in TCLP: consumers may be called in the realm of a constraint store more restrictive than that of their generators, and answers from the generator have to be filtered to discard those which are inconsistent with the constraint store at the time of the call to the consumer. In our implementation, an answer is represented by $\langle$`S, ProjStore`$\rangle$, where `S`, the set of Herbrand constraints, is applied by the tabling engine, and `ProjStore`, the projection of the constraint answer, is added to the constraint store of the consumer by `apply_answer/2`, which succeeds iff the resulting constraint store is consistent.

The design of the interface assumes that external constraint solvers are compatible with Prolog operational semantics so that when Prolog backtracks to a previous state, the corresponding constraint store is transparently restored. That can be done by adding a Prolog layer which uses the trail to store `undo` information that is used to reconstruct the previous constraint store when Prolog backtracks (this is a reasonable, minimal assumption for any integration of constraint solving and logic programming). The TCLP interface can use any constraint solver which follows this design because the suspension and resumption mechanisms of the tabling are based on the trailing mechanism of Prolog. When a consumer suspends, backtracking takes place, the memory stacks are frozen, and the variable bindings are saved on untrailing. They are reinstalled upon consumer resumption.

However, the entailment operations `call_entail/2` and `answer_compare/3` need to know the correspondence among variables in `ProjStore` and in `ProjStore`$_{\texttt{gen}}$ (resp., `ProjStore`$_{\texttt{ans}}$). To this end, projections are (conceptually) a pair `(VarList, Store)`, where `VarList` is a list of fresh variables in `Store` that correspond to `Vars`, the variables

[8] If there are several constraint domains involved, such as, for example, CLP($\mathbb{H}$) and CLP($\mathbb{Q}$), we assume that we can distinguish them appropriately at run-time, and the entailment is determined as `variant(G, G`$_{\texttt{gen}}$`)` $\wedge$ `ProjStore` $\sqsubseteq_{\mathbb{H}}$ `ProjStore`$_{\texttt{gen}}$ $\wedge$ `ProjStore` $\sqsubseteq_{\mathbb{Q}}$ `ProjStore`$_{\texttt{gen}}$

https://doi.org/10.1017/S1471068418000571 Published online by Cambridge University Press

on which the projection was originally made. Different, independent constraint stores can then be compared by means of these lists. This list is also necessary to apply the `ProjStore` of an answer to the global store: it is used to determine the correspondence of variables between the global and the projected store.

The actual implementation may differ among constraint solvers. For example, the TCLP($\mathbb{Q}$) interface (Figure 9) uses a list of fresh variables following the same order as those in `Vars`, while the TCLP($\mathbb{D}_{\leq}$) interface (Section 5.1) uses a vector containing the index in the matrix corresponding to every variable in `Vars`, again following the same order.

4.2 Implementation sketch

We summarily describe now the implementation of Mod TCLP, including the global table where generators, consumers, and answers are saved. We also present the transformation performed to execute tabled predicates and a (simplified) flowchart showing the interactions between the tabling engine and the constraint solver through the generic interface.

4.2.1 Global table

Tries are the data structure of choice for the call/answer global table (Ramakrishnan *et al.* 1995). In variant tabling, every generator $\mathtt{G}_{\mathtt{gen}}$ is uniquely associated (modulo variable renaming) to a leaf from where the Herbrand constraints for every answer hang. Generators are identified in Mod TCLP by the projection of the constraint store on the variables of the generator, that is, with a tuple $\langle \mathtt{G}_{\mathtt{gen}}, \mathtt{ProjStore}_{\mathtt{gen}} \rangle$. We store generators in a trie where each leaf is associated to a call pattern $\mathtt{G}_{\mathtt{gen}}$ and a list with a frame for each projected constraint store $\mathtt{ProjStore}_{\mathtt{gen}_i}$. Each frame identifies: (i) the projected constraint store $\mathtt{ProjStore}_{\mathtt{gen}_i}$, (ii) the answer table where the generator's answers $Ans(\mathtt{G}_{\mathtt{gen}}, \mathtt{ProjStore}_{\mathtt{gen}_i})$ are stored, and (iii) the list of its consumers.

Answers are represented by a tuple $\langle \mathtt{S}_{\mathtt{ans}}, \mathtt{ProjStore}_{\mathtt{ans}} \rangle$ and are stored in a trie where each leaf points to the Herbrand constraints $\mathtt{S}_{\mathtt{ans}}$ and to a list with the projected constraint stores $\mathtt{ProjStore}_{\mathtt{ans}_i}$ corresponding to answers whose Herbrand constraints are a variant of $\mathtt{S}_{\mathtt{ans}}$. The answers are stored in order of generation, since (as we mentioned before) it is not clear that other orders eventually payoff in terms of speeding up the entailment check of future answers.

4.2.2 TCLP directives and program transformation

Executing a CLP program under the TCLP framework only needs to enable tabling and import a package that implements the bridge CLP/tabling instead of the regular constraint solver. Figure 5, left, shows the TCLP version of the left recursive distance traversal program in Figure 1, right. The constraint interface remains unchanged, and the program code does not need to be modified to be executed under TCLP. The directive `:- use_package(tabling)` initializes the tabling engine and the directive `:- use_package(t_clpq)` imports TCLP($\mathbb{Q}$), the TCLP interface for the CLP($\mathbb{Q}$) solver (Section 4.3). To select another TCLP interface (more examples in Section 5), we just have to import the corresponding package. Finally, the directive `:- table dist/3.` specifies that the predicate `dist/3` should be tabled.

https://doi.org/10.1017/S1471068418000571 Published online by Cambridge University Press

```
:- use_package(tabling).
:- use_package(t_clpq).

:- table dist/3.
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).
```

```
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.
```

Fig. 5: `dist/3` with the directives to enable TCLP (left) and its transformation (right).

```
tabled_call(Call) :-
    call_lookup_table(Call, Vars, Gen),
    store_projection(Vars, ProjStore),
    (
       projstore_Gs(Gen, List_GenProjStore),
       member(ProjStore_G, List_GenProjStore),
       call_entail(ProjStore, ProjStore_G) ->
       suspend_consumer(Call)
    ;
       save_generator(Gen, ProjStore_G, ProjStore),
       execute_generator(Gen, ProjStore_G),
    ),
    answers(Gen, ProjStore_G, List_Ans),
    member(Ans, List_Ans),
    projstore_As(Ans, List_AnsProjStore),
    member(ProjStore_A, List_AnsProjStore),
    apply_answer(Vars, ProjStore_A).
```

Fig. 6: Implementation of `tabled_call/1`.

Figure 5, right, shows the transformation applied to the predicate `dist/3`. The original entry point to the predicate is rewritten to call an auxiliary predicate through the meta-predicate `tabled_call/1` (Figure 6). The auxiliary predicate corresponds to the original one with a renamed head and with an additional `new_answer/0` (Figure 7) at the end of the body to collect the answers. An internal global stack, called Parent Tabled Choice-Point (PTCP), is used to identify the generator under execution when `new_answer/0` is invoked.

4.2.3 Execution flow

Figure 8 shows a (simplified) flowchart to illustrate how the execution of a tabled call proceeds. The calls to predicates in the interface with the constraint solver have a grey background. We explain next the steps of an execution, using the labels in the nodes.

https://doi.org/10.1017/S1471068418000571 Published online by Cambridge University Press

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

new_answer :-
    complete.
```

Fig. 7: Implementation of `new_answer/0`.

0. A call to a tabled predicate `Call` starts the tabled execution invoking `tabled_call/1`, which takes the control of the execution.
1. `call_lookup_table/3` returns in `Gen` a reference to the trie leaf corresponding to the current call pattern `Call` and in `Vars` a list with the constrained variables of `Call`.
2. The tabling engine calls `store_projection/2`, which returns in `ProjStore` the projection onto `Vars` of the current constraint store.
3. The tabling engine uses `member/2` to retrieve in `ProjStore_G` the projected constraint stores from the list of frames associated to `Gen`. If it succeeds, the execution continues in step 5. If it fails, it may be because `Gen` is the first occurrence of this call pattern or because it does not entail any of the previous generators (and it is therefore a new generator).
4. The tabling engine calls `save_generator/3` to add a new frame to `Gen`, identifying the new call as a generator. The projected store `ProjStore` is saved in this new frame, and the answer table and the consumer list are initialized. From this point on, the generator is identified by $\langle$`Gen`, $\texttt{ProjStore}_\texttt{G}\rangle$ and the execution continues in step 7.
5. The constraint solver checks if the current store `ProjStore` entails the retrieved projected constraint store `ProjStore_G` using `call_entail/2`. In that case, `Call` is suspended in step 6. Otherwise, the tabling engine tries to retrieve another projected constraint store in step 3.
6. 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. Execution then continues by backtracking over the youngest generator. Otherwise,

https://doi.org/10.1017/S1471068418000571 Published online by Cambridge University Press

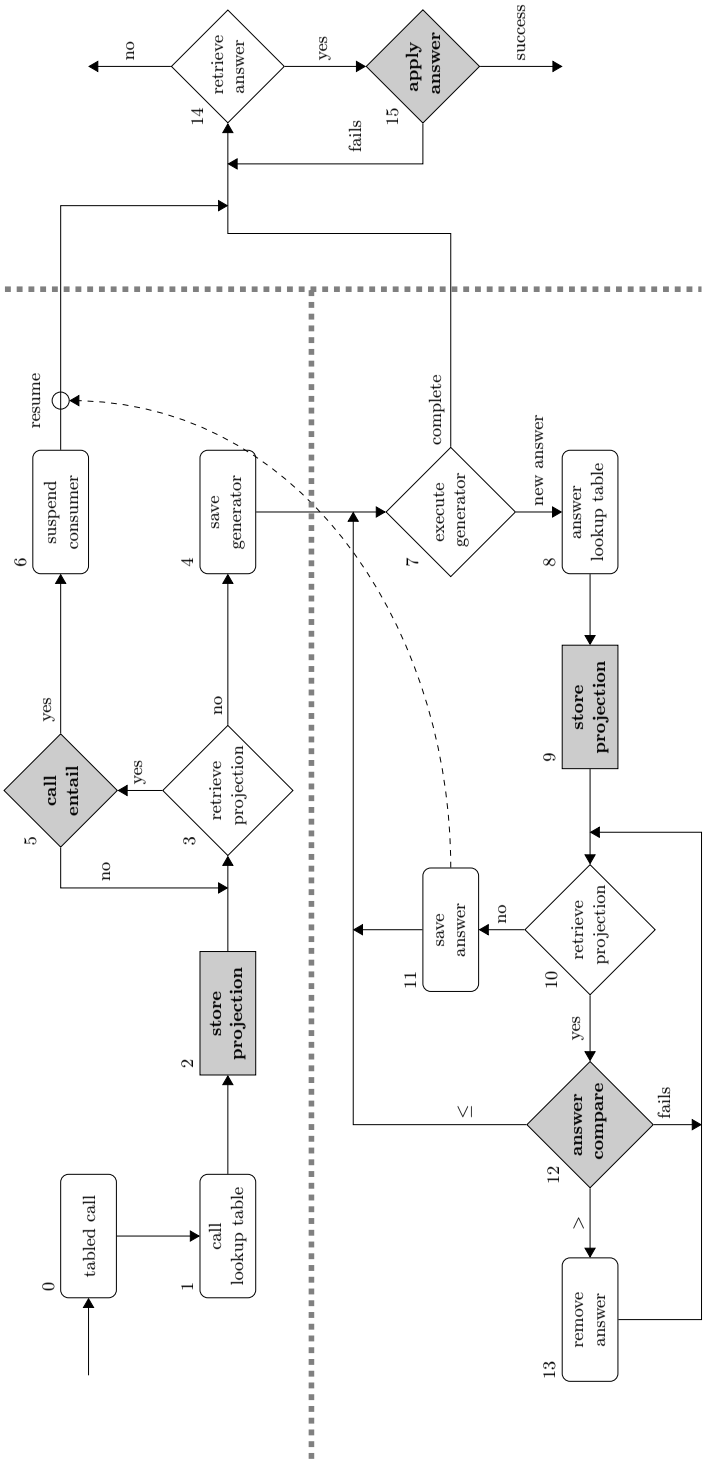

Fig. 8: Flowchart of the execution algorithm of Mod TCLP.

https://doi.org/10.1017/S1471068418000571 Published online by Cambridge University Press

`Call` continues the execution in step 14. A suspended consumer is resumed when its generator produces new answers and also continues in step 14.

7. The generator ⟨`Gen`, `ProjStore_G`⟩ is executed with `execute_generator/2`, which calls the renamed tabled predicate, and its reference is pushed onto the PTCP stack. If the execution reaches the end of a clause, a new answer has been found and `new_answer/0` continues the execution in step 8.
8. This is the entry point for `new_answer/0`. The tabling engine calls `answer_lookup_table/2`, which retrieves a reference to the generator in execution from the PTCP stack. A reference to the Herbrand constraints of the current answer in the generator's answer table is returned in `Ans`, and the list of variables from the call that are now/still constrained is returned in `Vars`.
9. The tabling engine invokes `store_projection/2`. This returns in `ProjStore` the projection of the current constraint store onto the constrained variables of the answer, `Vars`.
10. The tabling engine retrieves from `Ans` the list of projected constraint stores in `List_AnsProjStore` and calls `member/2` to return the stores one at a time in `ProjStore_A`. If it succeeds, the execution continues in step 12; otherwise, it continues in step 11. Failure can happen because all projected constraint stores were already retrieved from `List_AnsProjStore` or because `Ans` is the first answer with these Herbrand constraints.
11. The tabling engine adds `ProjStore` to the list of projected constraint stores (`List_AnsProjStore`) of the corresponding `Ans` with `save_answer/2` and resumes one by one the consumers of the current generator which were suspended in step 6. Since `new_answer/0` always fails, the execution backtracks to complete the execution of the generator (step 7).
12. The constraint solver checks if the current store `ProjStore` entails the retrieved projected constraint store `ProjStore_A` using `answer_compare/3`. 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 7. If `ProjStore` is entailed by `ProjStore_A` and they are not equal, it returns `Res = '>'`, and `ProjStore_A` is removed in step 13. Otherwise, it fails, and the execution continues in step 10, where the tabling engine tries to retrieve another projected constraint store.
13. The tabling engine marks the more particular answer as removed using `remove_answer/1`. Then, the execution continues in step 10.
14. Once the generator has exhausted all the answers and does not have more dependencies, it is marked as complete using `complete/0`, and the generator's reference is popped from the PTCP stack. 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 marked as removed, the answer will be applied in step 15. Otherwise, the execution backtracks to retrieve another answer.
15. Applying the Herbrand constraints `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 goes back to step 14.

https://doi.org/10.1017/S1471068418000571 Published online by Cambridge University Press

```
:- active_tclp.

store_projection(Vars, st(F,Proj) ) :-
        clpqr_dump_constraints(Vars, F, Proj).

call_entail(st(F,Proj), st(FGen,ProjGen) ) :-
        check_entailment(F, FGen, Proj, ProjGen).

answer_compare(st(F,Proj), st(FAns,ProjAns), =<) :-
        check_entailment(F, FAns, Proj, ProjAns), !.
answer_compare(st(F,Proj), st(FAns,ProjAns),  >) :-
        check_entailment(FAns, F, ProjAns, Proj).

apply_answer(Vars, st(FAns,ProjAns) ) :-
        Vars = FAns, clpq_meta(ProjAns).

check_entailment(Vars1, Vars2, Proj1, Proj2) :-
        Vars1 = Vars2, clpq_meta(Proj1), clpq_entailed(Proj2).
```

Fig. 9: The Mod TCLP interface for CLP(ℚ) is a bridge to existing predicates.

In the Appendix B of the Supplementary Material accompanying the paper at the TPLP archive, we examine step by step the execution of a program using the TCLP(ℚ) interface described in Section 4.3.

4.3 Implementation of the TCLP(ℚ) interface

Figure 9 shows the interface for Holzbaur's CLP(ℚ) solver (Holzbaur 1995) as an example of integration of a constraint solver with Mod TCLP. This CLP(ℚ) implementation already provides most of the interface actually acts as a bridge to existing predicates.

A Mod TCLP constraint interface starts with the declaration `:- active_tclp.` It makes the compiler adjust the program transformation according to the available interface predicates and instructs the tabling engine to activate the TCLP framework. The functionality required by the interface is implemented as follows:

- `store_projection(+Vars,-st(F,Proj))` calls the CLP(ℚ) predicate `clpqr_dump_constraints(+Vars,-F,-Proj)` to perform the projection. It returns in `Proj` the projection of the current store onto the list of variables `Vars`. The variables in `Proj` are fresh and are contained in the list `F`, following the same order as those in `Vars`. `F` is used to restore the association between the variables in `Vars` and the constraints in `Proj`, as mentioned in Section 4.1.
- `call_entail(+st(F,Proj),+st(FGen,ProjGen))` calls the auxiliary predicate `check_entailment(F, FGen, Proj, ProjGen)` which success if `Proj` $\sqsubseteq$ `ProjGen`. First, `check_entailment/4` unifies `F` and `FGen`, respectively, the variables of the projection of the current store and the variables of the generator's projection. Then, the CLP(ℚ) predicate `clpq_meta(+Proj)` makes `Proj` part of the current constraint store by executing it. This does not interact with the current store because the variables in `F` and `FGen` are fresh. And finally, `clpq_entailed(+ProjGen)` success if `ProjGen` is entailed by the current constraint store (i.e., `Proj` $\sqsubseteq$ `ProjGen`).

https://doi.org/10.1017/S1471068418000571 Published online by Cambridge University Press

- `answer_compare(+st(F,Proj),+st(FAns,ProjAns),Res)` calls the predicate `check_entailment(F, FAns, Proj, ProjAns)` to check if `Proj` $\sqsubseteq$ `ProjAns`. If it is the case, `answer_compare/3` returns `=<` in `Res`. Otherwise, it calls `check_entailment(FAns, F, ProjAns, Proj)` to check if `ProjAns` $\sqsubset$ `Proj`. If it is the case, it returns `>` in `Res`, otherwise it fails (i.e., there is no entailment in any direction).
- `apply_answer(+Vars,+st(FAns,ProjAns))` unifies `FAns`, the variables of `ProjAns` with `Vars`, those in the pattern of the resumed call. Then, it uses the CLP($\mathbb{Q}$) predicate `clpq_meta(+ProjAns)` to add the answer constraint store `ProjAns` to the current constraint store. If the resulting constraint store is consistent, execution continues, and it fails otherwise.

The TCLP interface for CLP($\mathbb{R}$) is similar to that of CLP($\mathbb{Q}$). CLP($\mathbb{R}$) uses floating-point numbers, and its performance is better than that of CLP($\mathbb{Q}$), which uses exact fractions. However, floating-point rounding errors make CLP($\mathbb{R}$) (and TCLP($\mathbb{R}$)) inappropriate for some applications, as entailment is unsound and therefore termination can be compromised.

4.4 Two-Step projection

The design we have presented strives for simplicity. There is however an improvement that can be used to obtain more performance/reduce memory usage, at the cost of a slightly more complex design. We present it now, with the understanding that it does not change the general ideas we have presented so far.

`store_projection/2` is usually the most expensive operation in the TCLP interface, but it is only mandatory when a call is a generator, which we can determine from entailment checking.[9] We have however placed `store_projection/2` before entailment checking because constraint solvers can often use the projection operation to compute some information needed by the entailment check. Instead of recomputing this information, the projection is divided in two parts: an initial operation `early_call_projection(+Vars, -EarlyProj)`, executed before the entailment phase, that returns in `EarlyProj` the information needed to check entailment, and a second operation `final_call_projection(+Vars, +EarlyProj, -ProjStore)` that is executed after the entailment phase if the entailment check fails (and the call would then be a generator). If it is executed, this operation returns the projected constraint store in `ProjStore` using the information in `EarlyProj`.

For symmetry, a similar mechanism is used with answers. Instead of using `store_projection/2` (step 10 in Figure 8), two specialized versions are used: `early_ans_projection/2` and `final_ans_projection/3`, respectively, called before and after the answer entailment check

Example 5

The TCLP($\mathbb{Q}$) interface in Figure 9 uses `store_projection/2` to project the constraint store of every new call. But, since `clpq_entailed/1` does not need the projection of the

[9] For efficiency, we can check entailment using the current constraint store A instead of its projection onto a set of variables S because $A \sqsubseteq B \iff Proj(S, A) \sqsubseteq B$, where $S = vars(B)$, as in our case.

https://doi.org/10.1017/S1471068418000571 Published online by Cambridge University Press

current constraint store to check entailment w.r.t. the projected constraint store of a previous generator, the execution of the projection can be delayed:

```
early_call_projection(Vars, st(Vars,_)).
call_entail(st(Vars,_), st(FGen,ProjGen)) :-
                              Vars = FGen, clpq_entailed(ProjGen).
final_call_projection(_,st(Vars,_), st(F,Proj)) :-
                            clpqr_dump_constraints(Vars, F, Proj).
```

The performance impact of implementing the *Two-Step* projection is evaluated in Section 6.4, using the TCLP($\mathbb{Q}$) interface.

5 Other TCLP interfaces

The design we presented brings more flexibility to a system with tabled constraints at a reasonable cost in implementation effort. To support this claim, we present the implementation of the TCLP interface for a couple of additional solvers: a constraint solver for difference constraints (Section 5.1) completely written in C and ported from (Chico de Guzmán *et al.* 2012) and a solver for constraints over finite lattices (Section 5.2).

5.1 Difference constraints

Difference constraints CLP($\mathbb{D}_{\leq}$) is a simple but relatively powerful constraint system whose constraints are generated from the set $\mathcal{C}_{\mathbb{D}_{\leq}} = \{X - Y \leq d : X, Y, d \in \mathbb{Z}\}$, where X and Y are variables, and d is a constant.

A system of difference constraints can be modeled with a weighted graph, and it is satisfiable if there are no cycles with negative weight. A solver for this constraint system can be based on shortest-path algorithms (Frigioni *et al.* 1998) where the constraint store is represented as an $n \times n$ matrix `A` of distances. The projection of a constraint store `A` onto a set of variables `V` extracts a submatrix `A'` containing all pairs (v_1, v_2) s.t. $v_1, v_2 \in V$. For efficiency, a projection can be 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 implementation uses attributed variables to map Prolog variables onto their representation in the matrix by having as attribute the index of each variable in the matrix. Therefore, calculating projection is fast.

The TCLP($\mathbb{D}_{\leq}$) interface showcases that, as we mentioned in Section 4.1, the representation of the projected constraint store depends on the constraint solver. In this case, the projected constraint store is represented by a triple `st(Id,Ln,Proj)` where `Id` is the memory address of the vector with the indexes of the constrained variables of the call/answer, `Ln` is its length (the number of constrained variables), and `Proj` is the memory address of a copy of the submatrix which represents the projected constraint store. The indexes of the vector `Id` follow the same order as the variables in `Vars` and are used to restore the association between `Vars` and `Proj` when they have to be compared or applied.

https://doi.org/10.1017/S1471068418000571 Published online by Cambridge University Press

CLP($\mathbb{D}_{\leq}$) checks entailment using `clpdiff_entailed((Id,Ln),ProjGen)` and `clpqdiff_entails((Id,Ln),ProjGen)`, where `Id` and `Ln` identify the position of the variables in the matrix `A` (the current constraint store), and `ProjGen` is the memory address of a submatrix which represents the projection of a previous generator. Note that the indexes of the submatrix from 1 to n follow the order of the indexes in `Id`, that is, the kth column/row of the submatrix corresponds to the variable identified by the kth index in `Id`.

Therefore, the TCLP($\mathbb{D}_{\leq}$) interface increases performance and reduces memory footprint using the *Two-Step* projection (Section 4.4) because it only makes a copy of the submatrix `Proj` when the entailment phase fails and the current call/answer becomes a generator/new answer. See below the implementation of the projection and answer comparison operations using the *Two-Step* projection in the answer entailment check:

```
1 early_ans_projection(Vars, st(Id,Ln,_) ) :-
2     diff_project_index(Vars,(Id,Ln)).
3 answer_compare(st(Id,Ln,_), st(_,_,ProjAns), =<) :-
4     diff_entailed((Id,Ln),ProjAns), !.
5 answer_compare(st(Id,Ln,_), st(_,_,ProjAns),  >) :-
6     diff_entails((Id,Ln),ProjAns).
7 final_ans_projection(_, st(Id,Ln,_), st(Id,Ln,Proj) ) :-
8     diff_projection((Id,Ln),Proj).
```

5.2 Constraints over finite lattices

A lattice is a triple $(\mathbb{S}, \sqcup, \sqcap)$ 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 that $\forall\, a \in \mathbb{S} \,.\, \bot \sqsubseteq a \sqsubseteq \top$.

In the system of constraints over finite lattices CLP($\mathbb{L}at$), the constraints between points in the lattice arise from (1) the topological relationship of the lattice elements and (2) any additional operations between the elements in the lattice. These two classes of constraints are handled by two different layers.

The external layer is concerned with the lattice topology and implements the constraint $Y \sqsubseteq X$ with $X, Y \in \mathbb{S}$ and the projection operation for variable elimination using Fourier's algorithm (Marriott and Stuckey 1998): the projection of $X \sqsubseteq d \wedge Y \sqsubseteq X$ onto Y is $Y \sqsubseteq d$. This layer provides entailment checking and the operation to add a projected constraint store to the current constraint store.

Further constraints on variables can be imposed by relationships derived from internal operations other than those in the lattice. Compare, for example, $Y \sqsubseteq X$ with $Y \sqsubseteq X \wedge Y = X \oplus X$ for some operation $\oplus$ among elements of the lattice: the additional information can be helpful to simplify (or prove inconsistent) the constraint store. In the lattice solver, a second layer implements these additional operations (if they exist) and communicates with the topology-related layer.

We have used this solver to implement a constraint tabling-based abstract interpreter (Section 6.5), where the points of the lattice are the elements of the abstract domain. The lattice implementation provides at least the operators $\sqcup$ and $\sqcap$ and the operations

https://doi.org/10.1017/S1471068418000571 Published online by Cambridge University Press

among the elements of the lattice, which are the counterparts of the operations in the concrete domain, as described above.

6 Experimental evaluation

In this section, we evaluate the performance of our framework using the four constraint systems and interfaces we have summarily described ($\mathbb{Q}$, $\mathbb{R}$, $\mathbb{D}_{\leq}$, and $\mathbb{L}at$).

In Section 6.1, we quantify the performance benefits of TCLP vs. LP, tabling, and CLP using the `dist/3` (Figure 1) program presented in Section 2 with the TCLP($\mathbb{Q}$) interface. Then, we explore the impact and advantages of a more flexible modular framework. On the one hand, in Section 6.2, we evaluate the performance impact of the increased overhead w.r.t. previous implementations with less flexibility (i.e., the previous TCLP implementation of Chico de Guzmán *et al.* (2012)), and, on the other hand, in Section 6.3, we evaluate the benefits of a more comprehensive answer management strategy.

In Section 6.4, we evaluate the performance benefits of the *Two-Step* projection using TCLP($\mathbb{Q}$). These benefits are due to the reduction in the number of projections executed during the evaluation. In Section 6.5, we use tabling and TCLP($\mathbb{L}at$) to implement and benchmark a simple abstract interpreter.

In Appendix C of the Supplementary Material accompanying the paper at the TPLP archive, we compare the expressiveness and performance of the TCLP($\mathbb{D}_{\leq}$), TCLP($\mathbb{R}$), and TCLP($\mathbb{Q}$) interfaces. In this case, the expressiveness of CLP($\mathbb{R}/\mathbb{Q}$) comes with an overhead (which is higher in CLP($\mathbb{Q}$) due to its higher precision), but in certain problems, this expressiveness can bring great benefits using TCLP (additionally, in some problems, the precision could be determinant).

The Mod TCLP framework presented in this paper is implemented in Ciao Prolog. The benchmarks and a Ciao Prolog distribution including the libraries and interfaces presented in this paper are available at `http://www.cliplab.org/papers/tplp2018-tclp/`.[10] All the experiments were performed on a Mac OS-X 10.9.5 machine with a 2.66 GHz Intel Core 2 Duo processor. Times are given in milliseconds.

6.1 Absolute performance of TCLP vs. LP vs. tabling vs. CLP

Let us recall Table 1, where we used the `dist/3` program (Figure 1) to support the use of TCLP due to its better termination properties. We now want to check whether, for those cases where LP or CLP also terminates, the performance of TCLP is competitive, and, for those cases where only TCLP terminates, whether its performance is reasonable. We have used a graph of 35 nodes without cycles (775 edges) and a graph of 49 nodes with cycles (785 edges) and timed the results – see Table 2.

As we already saw in Table 1, TCLP not only terminates in all cases but it is also faster than the rest of the frameworks due to the combination of tabling (which avoids entering loops and caches intermediate results) and constraint solving. It also suggests, in line with the experience in tabling, that left-recursive implementations are usually faster

[10] Stable versions of Ciao Prolog are available at `http://www.ciao-lang.org`. However, the libraries and interfaces are still in development, and they are not fully available yet in the stable versions.

https://doi.org/10.1017/S1471068418000571 Published online by Cambridge University Press

Table 2: Run time (ms) for `dist/3`. '–' means no termination.

	LP	CLP(ℚ)	Tab	Mod TCLP(ℚ)	Graph
Left recursion	–	–	2311	**1286**	Without cycles
Right recursion	> 5 min	5136	3672	2237	
Left recursion	–	–	–	**742**	With cycles
Right recursion	–	10,992	–	1776	

Table 3: Performance comparison (ms) of CLP vs. original TCLP vs. Mod TCLP using $\mathbb{D}_{\leq}$ for `truckload/4` and `step_bound/4`. '–' means no termination.

	CLP($\mathbb{D}_{\leq}$)	Orig. TCLP($\mathbb{D}_{\leq}$)	Mod TCLP($\mathbb{D}_{\leq}$)
`truckload(300)`	40,452	**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**

and preferable, as they avoid work by "suspending first" and reusing answers when they are ready.

6.2 The cost of modularity: Mod TCLP vs. original TCLP

The original TCLP implementation (Chico de Guzmán *et al.* 2012) was deeply intertwined with the tabling engine and had a comparatively low overhead. Since it was done on the same platform as ours (Ciao Prolog) and shares several components and low-level implementation decisions, it seems a fair and adequate baseline to evaluate the performance cost of the added modularity. We will evaluate both the frameworks using exactly the same implementation of difference constraints (Section 5.1) and two benchmarks:

`truckload(P, Load, Dest, Time)` (Cui and Warren 2000; Schrijvers *et al.* 2008)**:** it solves a shipment problem given a maximum `Load` for a truck, a destination `Dest`, and a list of packages to ship (1 to `P`.) We set `P = 30, Dest = chicago,` and use `Load` as parameter to vary its complexity. `truckload/4` does not need tabling, but tabling speeds it up.

`step_bound(Init, Dest, Steps, Limit)`: it is a left-recursive graph reachability program similar to `dist/3` that constrains the total number (`Limit`) of edge traversals. `step_bound/4` needs tabling in the case of graphs with cycles, as it is the case of the graph we will use in this evaluation.

Table 3 shows that `truckload/4` incurs a nearly three-fold increase in execution time with respect to the initial non-modular TCLP($\mathbb{D}_{\leq}$) implementation. This is mainly due to the overhead of the control flow. In the original implementation, execution did not leave the level of C, as the tabling engine called directly the constraint solver, also written in C. However, in Mod TCLP, the tabling engine (in C) calls the interface level (written in Prolog), which calls back the constraint solver (in C). The additional overhead is the

https://doi.org/10.1017/S1471068418000571 Published online by Cambridge University Press

Table 4: Run time (ms) comparison of answer management strategies using Mod TCLP($\mathbb{D}_{\leq}$) for `truckload/4` and `step_bound/4`.

	Mod TCLP($\mathbb{D}_{\leq}$)			
	$\emptyset$	$\leftarrow$	$\rightarrow$	$\leftrightarrow$
`truckload(300)`	742,039	7806	7780	**7268**
`truckload(200)`	11,785	2314	2354	**2239**
`truckload(100)`	300	263	263	**259**
`step_bound(30)`	–	8450	–	**1469**
`step_bound(20)`	–	6859	38,107	**1267**
`step_bound(10)`	–	2846	8879	**845**

price we pay to make it much easier to plug in additional constraint solvers, which in the original TCLP needed *ad-hoc*, low-level wiring.

However, `step_bound/4` is less efficient in the original TCLP($\mathbb{D}_{\leq}$) implementation than in Mod TCLP and cannot be executed in CLP($\mathbb{D}_{\leq}$) due to the cycles in the graph. The reason behind this improvement is the enhanced answer management strategy whose implementation was made possible by our modular design. We will explore this point in the next section.

6.3 Improved answer management strategies

The modular design of Mod TCLP makes it possible to implement alternatives for internal operations more easily. In particular, the solver interface can include the `answer_compare/3` operation which determines whether a new answer entails, is entailed by, or none of them, some previous answer. This can be used to decide whether to add or not a new answer and remove or not an existing answer. This is undoubtedly expensive in general, but as advanced in Section 4.1, it holds promise for improving performance. To validate this intuition, we executed again `truckload/4` and `step_bound/4` with TCLP($\mathbb{D}_{\leq}$) under four different answer management strategies:

- $\emptyset$ all the answers are stored.
- $\leftarrow$ checks if new answers entail previous answers. If so, the new answer is discarded. That is the strategy used in the original TCLP framework.
- $\rightarrow$ checks if new answers are entailed by previous answers. If so, the previous answers are flagged as removed and ignored, and the new answer is stored.
- $\leftrightarrow$ checks entailment in both directions, discarding new answers and removing more particular answers.

The results in Table 4 confirm that, in the examples studied, and despite the cost of these strategies, the computation time is reduced. The '$\leftrightarrow$' strategy proves to be the best one, although by a small margin in some cases.

On the other hand, the worst strategy is '$\emptyset$,' which for the `truckload/4` program increases the execution time several orders of magnitude for large cases, while for the `step_bound/4` program, the execution does not terminate because it runs out of memory when trying to generate infinitely many repeated answers. While `truckload/4` behaves similarly for the other strategies, `step_bound/4` varies drastically (i.e., '$\rightarrow$' runs out of memory for the largest case).

https://doi.org/10.1017/S1471068418000571 Published online by Cambridge University Press

Table 5: Number of answers: saved (*Sav.*), discarded (*Dis.*) , removed (*Rem.*), and returned to the query (*Ret.*) for each answer management strategy.

Answer strategy		# Sav.	# Dis.	# Rem.	# Ret.
$\emptyset$	`truckload(300)`	448,538	0	0	14,999
	`truckload(200)`	52,349	0	0	1520
	`truckload(100)`	2464	0	0	58
$\leftarrow$	`truckload(300)`	67,503	9971	0	41
	`truckload(200)`	16,456	1325	0	23
	`truckload(100)`	1525	52	0	6
	`step_bound(30)`	44,549	716,826	0	252
	`step_bound(20)`	37,548	599,259	0	242
	`step_bound(10)`	15,625	242,351	0	165
$\rightarrow$	`truckload(300)`	75,272	0	9460	30
	`truckload(200)`	17,568	0	1298	18
	`truckload(100)`	1490	0	49	9
	`step_bound(30)`	>1,145,690	0	>1,074,071	–
	`step_bound(20)`	946,309	0	891,078	441
	`step_bound(10)`	294,728	0	276,867	221
$\leftrightarrow$	`truckload(300)`	48,524	6596	1740	5
	`truckload(200)`	13,550	1046	240	5
	`truckload(100)`	1343	45	10	3
	`step_bound(30)`	9697	74,528	4571	25
	`step_bound(20)`	9352	71,658	4371	25
	`step_bound(10)`	6650	56,935	3019	25

Part of the reasons for these differences can be inferred from Table 5, where, for each benchmark and strategy, we show how many of the generated answers were saved, discarded before being inserted, or removed after insertion. Note that these results are independent of the constraint solver used (i.e., executing the same programs using CLP($\mathbb{Q}$) or CLP($\mathbb{R}$) instead of CLP($\mathbb{D}_{\leq}$) generates the same answers).

For `truckload/4`, the '$\rightarrow$' and the '$\leftarrow$' strategies generate, discard/remove, and return a similar number of answers, which means that their 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, '$\leftrightarrow$.' For `step_bound/4`, '$\rightarrow$' generates many more candidate answers than either of the other two – in excess of 1 million for `step_bound(30)` — but '$\leftarrow$' also generates one order of magnitude more candidates answers than '$\leftrightarrow$.' Note that the number of generated answers is not always the same since, as discussed before, fewer saved answers wake up fewer consumers.

As an additional example of the usefulness of obtaining the most general correct answer, Figure 10 shows a graph and the program `sd/3`, used in Cui and Warren (2000) to calculate the "shortest distance" between the nodes in the graph. For a query such as `?- sd(X,Y,Dist),` the system reported in Cui and Warren (2000) returns a sequence of n answers of the form `Dist #>=` $\mathtt{N}_k$. Each $\mathtt{N}_k$ is the current achievable shortest distance from `X` to `Y`, such as $\mathtt{N}_1 > \cdots > \mathtt{N}_n$, and the later $\mathtt{N}_n$ is the shortest distance from `X` to `Y`. That is, for the query `?- sd(a,c,Dist),` it returns `Dist #>= 6.0` and `Dist #>= 3.0`. While

https://doi.org/10.1017/S1471068418000571 Published online by Cambridge University Press

```
1 :- table sd/3.
2
3 sd(X,Y,D) :-
4       edge(X,Y,D0),
5       D #>= D0.
6 sd(X,Y,D) :-
7       sd(X,Z,D1),
8       edge(Z,Y,D2),
9       D #>= D1+D2.
```

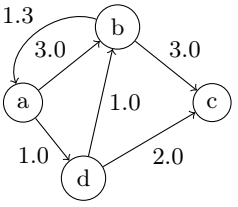

Fig. 10: `sd/3` – a shortest-distance program.

using Mod TCLP under the '↔' strategy, the evaluation of the query `?- sd(a,c,Dist)` only returns the answer `Dist #>= 3.0` (the most general) which corresponds to the tightest bound for the shortest distance between the nodes `a` and `c`.

6.4 Improved Two-Step projection

The design of Mod TCLP makes it possible to postpone the projection during the call/answer entailment phase using the *Two-Step* projection. As we advanced in Section 4.4, it holds promise for performance improvements. To validate this intuition, we use two benchmarks:

`fib(N, F)` the doubly recursive Fibonacci program run *backwards*. It is well known that tabling reduces `fib/2` complexity from exponential to linear. In addition, CLP makes it possible to run exactly the same program *backwards* to find the index of some Fibonacci number by generating a system of equations whose solution is the index of the given Fibonacci number (e.g., for the query `?- fib(N, 89)`, the answer is `N = 11`). Under CLP, the size of this system of equations grows exponentially with the index of the Fibonacci number. However, under TCLP, entailment makes redundant equations not to be added and solving them becomes less expensive. Additionally, entailment makes it possible to terminate (with failure) even when the query does not contain a non Fibonacci number, that is, `fib(N,`10^{314}`)`.

`dist(X, Y, D)` the program already used in Section 6.1.

We executed each of them with Mod TCLP($\mathbb{Q}$) and the two designs for the call projection we discussed earlier:

***One-Step*:** The projection of the call is executed before the call entailment phase (Figure 9). Note that CLP($\mathbb{Q}$) does not need this projection to check entailment of the current call constraint store w.r.t. another constraint store.

***Two-Step*:** The projection of the call is executed using `final_call_projection/3` and, therefore, it is only executed when the call turns out to be a generator.

The results in Table 6 (top) confirm that, in the examples studied, the *Two-Step* design reduces the computation time, although only by a small margin in the case of `dist/3` with left recursion. That is because, as we see in Table 6 (bottom), using the *Two-Step* projection, `dist/3` with left recursion executes the projection of a call only once; while using *One-Step,* it executes the projection twice, and, therefore, we only save

https://doi.org/10.1017/S1471068418000571 Published online by Cambridge University Press

Table 6: Run time in ms (top) and number of call projections (bottom) for each projection design, *One-Step* and *Two-Step*.

		Mod TCLP(ℚ)		
Run-time (ms)		One-Step	Two-Step	Ratio
`fib(N,F`$_{1500}$`)`		206,963	**126,461**	1.63
`fib(N,F`$_{1000}$`)`		89,974	**55,183**	1.63
`fib(N,F`$_{500}$`)`		22,133	**13,612**	1.63
`fib(N,`10^{314}`)`		205,638	**125,670**	1.63
`dist/3 right rec.`	Without cycles	2855	**2506**	1.14
	With cycles	2399	**1850**	1.30
`dist/3 left rec.`	Without cycles	1436	**1428**	1.01
	With cycles	776	**772**	1.01

		Mod TCLP(ℚ)		
# call projections		One-Step	Two-Step	Ratio
`fib(N,F`$_{1500}$`)`		1,129,497	**565,500**	2.00
`fib(N,F`$_{1000}$`)`		502,997	**252,000**	2.00
`fib(N,F`$_{500}$`)`		126,497	**63,500**	1.99
`fib(N,`10^{314}`)`		1,126,499	**563,252**	2.00
`dist/3 right rec.`	Without cycles	1563	**181**	8.64
	With cycles	2144	**443**	4.84
`dist/3 left rec.`	Without cycles	2	**1**	2.00
	With cycles	2	**1**	2.00

Note: F_n is the nth Fibonacci number, and 10^{314} is not a Fibonacci number.

the execution of one projection. Since they are executed early in the evaluation, the constraint store is small, and their execution is faster than in the case of `dist/3` with right recursion. Note that using the *Two-Step* projection, `dist/3` with right recursion executes up to eight times fewer call projections, and as consequence, its execution has better performance.

On the other hand, `fib/2` reduces drastically the computation time using *Two-Step* projection because, during the execution, call entailment is checked many times (although the ratio of *useless* projections is similar to that of `dist/3` with left recursion). Note that the Fibonacci number F_{1500} and 10^{314} have the same size (315 digits), and the run time/number of projections for `fib(N,F`$_{1500}$`)` and for `fib(N,`10^{314}`)` are similar. That is because the work needed to find the index of a Fibonacci number is similar to the work needed to confirm whether a number is or is not a Fibonacci number.

6.5 Abstract interpretation: Tabling vs. TCLP(𝕃at)

We compare here tabling and TCLP using two versions of a simple abstract interpreter (Cousot and Cousot 1977). The interpreter executes the programs to be analyzed on an abstract domain, collecting the possible values at every point until a fixpoint is reached. The result of the execution is a safe approximation of the run-time values of the

https://doi.org/10.1017/S1471068418000571 Published online by Cambridge University Press

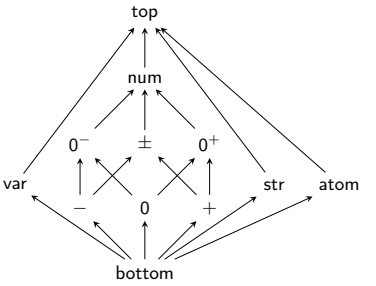

Fig. 11: *Signs* abstract domain.

variables in the concrete domain. The abstract domain we have used in this example is the *signs* abstract domain (Figure 11).

The two versions of the abstract interpreter we have used are as follows:

Tabling This version is a simple abstract interpreter written using tabling. This ensures termination, as the abstract domain is finite.

TCLP This version is based on the previous abstract interpreter, but it uses the TCLP($\mathbb{L}at$) constraint solver interface (Section 5.2) to operate on the abstract domain and setup constraints over the variables. The main differences with the *tabling* version are that TCLP uses constraint entailment instead of variant checking for loop detection, and, therefore, it can also use the answers to more general goals to avoid computing more particular goals.

We applied our abstract interpreter to two programs:

`takeuchi/m` (Figure 12): a Prolog implementation of the n-dimensional generalization of the Takeuchi function (Knuth 1991). The program is parametric on the number of input arguments n and it returns the result in its last argument.

`sentinel/m` (Figure 13): a variant of a synthetic program presented in Genaim *et al.* (2001). It receives as input its first argument (the `Sentinel`), and the next n arguments $\mathtt{A}_1, \ldots, \mathtt{A}_n$ are a ring-ordered[11] series of numbers. The outputs are the arguments $\mathtt{B}_1, \ldots, \mathtt{B}_n$, which correspond to a circular shift of $\mathtt{A}_1, \ldots, \mathtt{A}_n$, such that on success $\mathtt{B}_i < \mathtt{B}_{i+1}$ for all $i < n$, and if $\mathtt{Sentinel} = 0$, the first half of $\mathtt{B}_i$ are negative and the second half are positive; if $\mathtt{Sentinel} < 0$, $\mathtt{B}_i < \mathtt{Sentinel}$ for all i; and if $\mathtt{Sentinel} > 0$, $\mathtt{B}_i > \mathtt{Sentinel}$ for all i.

Table 7 shows the run-time results of analyzing `takeuchi/m` parameterized by the dimension of the function, n ($m = n + 1$), and `sentinel/m` parameterized by n, the length of the ring ($m = 2n + 1$). In both examples, the analysis with the TCLP version of the interpreter is faster than the analysis with the 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`, let us write $\mathtt{A}^{abs}$ to represent $\mathtt{A} \sqsubseteq abs$. On the one hand, when an initial goal $\mathtt{ring}(\mathtt{A}_1{}^{top}, \ldots, \mathtt{A}_n{}^{top}, \mathtt{B}_1{}^{top}, \ldots, \mathtt{B}_n{}^{top})$ is

[11] That is, there is a j such that $\mathtt{A}_j < \mathtt{A}_{j+1}$, $\mathtt{A}_{j+1} < \mathtt{A}_{j+2}$, $\ldots$, $\mathtt{A}_n < \mathtt{A}_1$, $\mathtt{A}_1 < \mathtt{A}_2$, $\ldots$, $\mathtt{A}_{j-2} < \mathtt{A}_{j-1}$.

https://doi.org/10.1017/S1471068418000571 Published online by Cambridge University Press

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

```
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).
```

Fig. 12: n-dimensional Takeuchi function and its 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).
```

Fig. 13: `sentinel/m` program.

interpreted by the TCLP analyzer, the first clause of `ring/2n` produces the first answer. Then, the interpreter continues with the second clause, interprets the goal $\mathtt{A}_1 > \mathtt{A}_2$, and starts the evaluation of $\mathtt{ring}(\mathtt{A}_2{}^{num}, \ldots, \mathtt{A}_n{}^{top}, \mathtt{A}_1{}^{num}, \mathtt{B}_1{}^{top}, \ldots, \mathtt{B}_n{}^{top})$. Since $num \sqsubseteq top$, this new call entails the previous one and TCLP suspends this execution. Then, the interpreter continues with the third clause, which starts the evaluation of $\mathtt{ring}(\mathtt{A}_n{}^{top}, \mathtt{A}_1{}^{top}, \ldots, \mathtt{A}_{n-1}{}^{top}, \mathtt{B}_1{}^{top}, \ldots, \mathtt{B}_n{}^{top})$, and TCLP also suspends the execution. Since the generator does not have 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, when the initial goal is evaluated in the tabling interpreter with $\mathtt{A}_1{}^{top}, \ldots, \mathtt{A}_n{}^{top}, \mathtt{B}_1{}^{top}, \ldots, \mathtt{B}_n{}^{top}$ as entry substitution, the first answer is also produced. Then the interpreter continues with the second clause, interprets the goal $\mathtt{A}_1 > \mathtt{A}_2$, and starts the evaluation of the recursive call with the entry substitution $\mathtt{A}_1{}^{num}, \ldots, \mathtt{A}_n{}^{top}, \mathtt{B}_1{}^{num}, \ldots, \mathtt{B}_n{}^{top}$. However, tabling does not suspend the execution because it is not a *variant* call of the previous one, which results in increased computation time.

https://doi.org/10.1017/S1471068418000571 Published online by Cambridge University Press

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

		Tabling	Mod TCLP($\mathbb{L}at$)
`takeuchi/m` ($m = n + 1$)	$n = 8$	31.44	**8.09**
	$n = 6$	13.75	**5.85**
	$n = 3$	**2.42**	3.12
`sentinel/m` ($m = 2n + 1$)	$n = 8$	1375.13	**9.23**
	$n = 6$	218.93	**6.53**
	$n = 4$	30.99	**4.56**

Table 8: Run time (ms) for `analyze(sentinel/m)`.

		Tabling		Mod TCLP($\mathbb{L}at$)	
		constraints before call	unconstrained call	constraints before call	unconstrained call
`sentinel/m` ($m = 2n+1$)	$n = 8$	**749.38**	1375.13	**5.29**	9.23
	$n = 6$	**98.80**	218.93	**3.31**	6.53
	$n = 4$	**6.53**	30.99	**2.85**	4.56

Table 8 shows the results of analyzing `sentinel/m` in two different scenarios: without any constraints in the abstract substitution of the variables or adding the constraint `Sentinel` $\sqsubseteq +$ before the analysis. Adding that domain restriction reduces analysis times by approximately the same ratio in both cases. Note that this compares two scenarios which are possible both for TCLP and for tabling without constraints. Additionally, the TCLP-based analyzer would be able to take into account constraints *among* variables, which would not be directly possible using tabling without constraints.

7 Related work

The initial ideas of tabling and constraints originate in Kanellakis *et al.* (1995), where a variant of Datalog featuring constraints was proposed. The time and space properties associated with the bottom-up evaluation of Datalog were studied in Toman (1997b), where a top-down evaluation strategy featuring tabling was proposed.

XSB (Swift and Warren 2012) was the first logic programming system that provided tabled CLP as a generic feature, instead of resorting to *ad-hoc* adaptations. This was done by extending XSB with attributed variables (Cui and Warren 2000), 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 (even for goals with constraints), instead of entailment checking of calls/answers. This makes programs terminate in fewer cases than using entailment and takes longer in other cases. This is similar to what happens in tabled logic programs with and without subsumption (Swift and Warren 2010). From the point of view of interfacing/adding additional CLP solvers to existing systems, the framework in Cui and Warren (2000) requires the constraint solver to provide the predicates `projection/1` and `entail/2`, which are used to discard more particular answers, but only in one direction. It also requires the implementation of the predicate

https://doi.org/10.1017/S1471068418000571 Published online by Cambridge University Press

`abstract/3`, which has to take care of the call abstraction. However, it is not clear if this predicate is part of the constraint solver or of the user program.

A general framework for Constraint Handling Rules (CHR) under tabled evaluation is described in Schrijvers *et al.* (2008). It takes advantage of 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 that use entailment. Besides, the need to change the representation between CHR and Herbrand terms takes a toll in performance. From the interface point of view, the framework provides interesting hooks: `projection(PredName)` specifies that predicate `PredName/1` determines how projection is to be performed, which makes it possible to, for example, ignore arguments; `canonical_form(PredName)` modifies the answer store to a canonical form as defined by `PredName/2` so that identical answers can be detected (e.g., using `sort/2` the constraints `[leq(1,X),leq(X,3)]` and `[leq(X,3),leq(1,X)]` are reduced to the same canonical form), and `answer_combination(PredName)`, if specified, applies `PredName/3` in such a way that two answers can be merged into one.

Failure Tabled CLP (Gange *et al.* 2013) 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 counters to implement a mechanism akin to iterative deepening.

Last, the original TCLP proposal (Chico de Guzmán *et al.* 2012) features entailment checking for calls and (partially) for answers, executes calls with all the constraints, and has good performance. However, from the interface point of view, it did not clearly state which operations must be present in the constraint solver, which made it difficult to extend, and was not focused on a modular design which, for example, made implementing specific answer management strategies cumbersome.

8 Conclusions and further work

We have presented an approach to include constraint solvers in logic programming systems with tabling. Our main goal is making the addition of new constraint solvers easier while taking full advantage of entailment between constraint stores. In order to achieve this, we determined the services that a constraint solver should provide to a tabling engine. This interface has been designed to give the constraint solver freedom to implement them. To validate our design, we have interfaced one solver previously written in C (CLP($\mathbb{D}_{\leq}$)), two existing classical solvers (CLP($\mathbb{Q}$ /$\mathbb{R}$)), and a new solver (CLP($\mathbb{L}at$)), and we have found the integration to be easy – certainly easier than with other designs – validating the usefulness of the capabilities that our system provides.

We evaluated 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 in the current implementation, we gave more importance to the cleanliness of the code and the design.

The facilities that our framework provides to integrate constraint solvers with tabling pave the way to new research directions:

https://doi.org/10.1017/S1471068418000571 Published online by Cambridge University Press

- Explore richer, faster, and more flexible implementations of abstract interpretation-based analyzers.
- Evolve CLP($\mathbb{L}at$) into a lattice domain that can capture reasoning in ontologies and explore its usage to implement constraint tabled-based reasoning systems featuring automatic reuse of more general concepts and combinations of answers into more general concepts.
- Implement a TCLP interface for a constraint solver over finite domains, CLP($\mathbb{FD}$) (Van Hentenryck 1989; Dincbas *et al.* 1988; Díaz and Codognet 1993). CLP($\mathbb{FD}$) is widely used to model discrete problems such as scheduling, planning, packing, and timetabling. The implementation is not straightforward due to the difficulty of expressing projection and entailment inside CLP($\mathbb{FD}$) (Carlson *et al.* 1994).

Supplementary material

To view supplementary material for this article, please visit https://doi.org/10.1017/S1471068418000571.

References

Carlson, B., Carlsson, M., and Diaz, D. 1994. Entailment of finite domain constraints. In *International Conference on Logic Programming (ICLP'94)*, Santa Marherita Ligure, Italy, 13–18 Jun. 1994, P. Van Hentenryck Ed. The MIT Press.

Charatonik, W., Mukhopadhyay, S. and Podelski, A. 2002. Constraint-based infinite model checking and tabulation for stratified CLP. In *ICLP'02*, Copenhagen, Denmark, 29 Jul.–1 Aug. 2002, P. J. Stuckey, Ed. Lecture Notes in Computer Science, vol. 2401. Springer, 115–129.

Chen, W. and Warren, D. S. 1996. Tabled evaluation with delaying for general logic programs. *Journal of the ACM 43,* 1 (January), 20–74.

Chico de Guzmán, P., Carro, M., Hermenegildo, M. V. and Stuckey, P. 2012. A general implementation framework for tabled CLP. In *FLOPS'12*, Kobe, Japan, 23–25 May 2012, T. Schrijvers and P. Thiemann, Eds. Lecture Notes in Computer Science, vol. 7294. Springer Verlag, 104–119.

Cousot, P. and Cousot, R. 1977. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *ACM Symposium on Principles of Programming Languages (POPL'77)*, Los Angeles, CA. Jan. 1977, R. M. Graham, M. A. Harrison, and R. Sethi, Eds. ACM Press, 238–252.

Cui, B. and Warren, D. S. 2000. A system for Tabled Constraint Logic Programming. In *International Conference on Computational Logic*, London, UK, 24–28 Jul. 2000, J. W. Lloyd, V. Dahl, U. Furbach, M. Kerber, K.-K. Lau, C. Palamidessi, L. Moniz Pereira, Y. Sagiv, and P. J. Stuckey, Eds. Lecture Notes in Computer Science, vol. 1861. Springer Verlag, 478–492.

Dawson, S., Ramakrishnan, C. R. and Warren, D. S. 1996. Practical program analysis using general purpose logic programming systems – A case study. In *Proc. of the ACM SIGPLAN'96 Conference on Programming Language Design and Implementation*, Philadephia, PA, 21–24 May, 1996, C. N. Fischer, Ed. ACM Press, New York, USA, 117–126.

Díaz, D. and Codognet, P. 1993. A minimal extension of the WAM for `clp(fd)`. In *Proc. of the Tenth International Conference on Logic Programming*, Budapest, Hungary, 21–25 Jun. 1993, D. S. Warren, Ed. MIT Press, 774–790.

Dincbas, M., Hentenryck, P. V., Simonis, H. and Aggoun, A. 1988. The constraint logic programming language CHIP. In *Proc. of the 2nd International Conference on Fifth Generation Computer Systems*, Tokyo, Japan, 28 Nov.–2 Dec. 1988, ICOT Ed. 249–264.

Falaschi, M., Levi, G., Martelli, M. and Palamidessi, C. 1989. Declarative modeling of the operational behaviour of logic programs. *Theoretical Computer Science 69*, 289–318.

Frigioni, D., Marchetti-Spaccamela, A. and Nanni, U. 1998. Fully dynamic shortest paths and negative cycles detection on digraphs with arbitrary arc weights. In *ESA*, Venice, Italy, 24–26 Aug. 1998, G. Bilardi, G. F. Italiano, A. Pietracaprina and G. Pucci, Eds. Springer, 320–331.

Gange, G., Navas, J. A., Schachte, P., Søndergaard, H. and Stuckey, P. J. 2013. Failure tabled constraint logic programming by interpolation. *TPLP 13,* 4–5, 593–607.

Genaim, S., Codish, M. and Howe, J. 2001. Worst-case groundness analysis using definite Boolean functions. *Theory and Practice of Logic Programming 1,* 5, 611–615.

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. *Theory and Practice of Logic Programming 12,* 1–2 (January), 219–252. URL: http://arxiv.org/abs/1102.5497. [Accessed on December 28, 2018].

Holzbaur, C. 1995. *OFAI CLP(Q,R) Manual*, Edition 1.3.3. Tech. Rep. TR-95-09, Austrian Research Institute for Artificial Intelligence, OFAI, Vienna, Austria.

Jaffar, J. and Maher, M. 1994. Constraint logic programming: A survey. *Journal of Logic Programming 19/20*, 503–581.

Kanellakis, P. C., Kuper, G. M. and Revesz, P. Z. 1995. Constraint query languages. *Journal of Computer and System Sciences 51,* 1, 26–52.

Knuth, D. E. 1991. Textbook examples of recursion. In *Artificial Intelligence and Mathematical Theory of Computation: Papers in Honor of John McCarthy*, 207–230.

Marriott, K. and Stuckey, P. J. 1998. *Programming with Constraints: An Introduction.* MIT Press.

Ramakrishna, Y., Ramakrishnan, C., Ramakrishnan, I., Smolka, S., Swift, T. and Warren, D. 1997. Efficient model checking using tabled resolution. In *Computer Aided Verification*, Haifa, Israel, 22–25 Jun. 1997, Y. S. Ramakrishna, C. R. Ramakrishnan, I. V. Ramakrishnan, S. A. Smolka, T. Swift and D. S. Warren, Eds. Lecture Notes in Computer Science, vol. 1254. Springer Verlag, 143–154.

Ramakrishnan, I., Rao, P., Sagonas, K., Swift, T. and Warren, D. 1995. Efficient tabling mechanisms for logic programs. In *ICLP'95*, Tokyo, Japan, 13–16 Jun. 1995, L. Sterling, Ed. MIT Press, 697–711.

Schrijvers, T., Demoen, B. and Warren, D. S. 2008. TCHR: A framework for tabled CLP. *Theory and Practice of Logic Programming* 4, July, 491–526.

Swift, T. and Warren, D. S. 2010. Tabling with answer subsumption: Implementation, applications and performance. In *Logics in Artificial Intelligence*, vol. 6341, Helsinki, Finland, 13–15 Sep. 2010, T. Janhunen and I. Niemelä, Eds. Springer Verlag, 300–312.

Swift, T. and Warren, D. S. 2012. XSB: Extending prolog with tabled logic programming. *Theory and Practice of Logic Programming* 1–2, January, 157–187.

Tamaki, H. and Sato, M. 1986. OLD resolution with tabulation. In *Third International Conference on Logic Programming*, 14–18 Jul. 1986, E. Shapiro, Ed. Lecture Notes in Computer Science, Springer-Verlag, London, 84–98.

Toman, D. 1997a. Constraint databases and program analysis using abstract interpretation. In *Constraint Databases and Their Applications*, Cambridge, MA, 19 Aug. 1996, V. Gaede, A. Brodsky, O. Günther, D. Srivastava, V. Vianu and M. Wallace, Eds. Lecture Notes in Computer Science, vol. 1191, 246–262.

Toman, D. 1997b. Memoing evaluation for constraint extensions of datalog. *Constraints 2,* 3/4, 337–359.

Van Hentenryck, P. 1989. *Constraint Satisfaction in Logic Programming*. MIT Press.

Warren, D. S. 1992. Memoing for logic programs. *Communications of the ACM 35,* 3, 93–111.

Warren, R., Hermenegildo, M. and Debray, S. K. 1988. On the practicality of global flow analysis of logic programs. In *Fifth International Conference and Symposium on Logic Programming*, Seattle, Washington, 15–19 Aug. 1988, R. A. Kowalski and K. A. Bowen, Eds. MIT Press, 684–699.

Zou, Y., Finin, T. and Chen, H. 2005. F-OWL: An inference engine for semantic web. In *Formal Approaches to Agent-Based Systems*, Greenbelt, MD, 26–27 Apr. 2004, M. G. Hinchey, J. L. Rash, W. Truszkowski and C. A. Rouff, Eds. Lecture Notes in Computer Science, vol. 3228. Springer Verlag, 238–248.