TPLP **19** (5–6): 1107–1123, 2019. © Cambridge University Press 2019
doi:10.1017/S1471068419000383

*Evaluation of the Implementation of an Abstract Interpretation Algorithm using Tabled CLP**

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

submitted 31 July 2019; accepted 2 Aug 2019

Abstract

CiaoPP is an analyzer and optimizer for logic programs, part of the Ciao Prolog system. It includes PLAI, a fixpoint algorithm for the abstract interpretation of logic programs which we adapt to use *tabled constraint logic programming*. In this adaptation, the tabling engine drives the fixpoint computation, while the constraint solver handles the LUB of the abstract substitutions of different clauses. That simplifies the code and improves performance, since termination, dependencies, and some crucial operations (e.g., branch switching and resumption) are directly handled by the tabling engine. Determining whether the fixpoint has been reached uses *semantic equivalence*, which can decide that two syntactically different abstract substitutions represent the same element in the abstract domain. Therefore, the tabling analyzer can reuse answers in more cases than an analyzer using syntactical equality. This helps achieve better performance, even taking into account the additional cost associated to these checks. Our implementation is based on the TCLP framework available in Ciao Prolog and is one-third the size of the initial fixpoint implementation in CiaoPP. Its performance has been evaluated by analyzing several programs using different abstract domains.

KEYWORDS: Abstract Interpretation, Constraints, Tabling, Prolog, PLAI, CiaoPP.

1 Introduction

Tabling (Tamaki and Sato 1986; Warren 1992) is an execution strategy for logic programs that suspends repeated calls which could cause infinite loops. Answers from non-looping branches are used to resume suspended calls which can, in turn, generate more answers and resume other suspended calls. 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 (i.e., they can only generate terms with a fixed finite depth) 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.

Constraint Logic Programming (CLP) (Jaffar and Maher 1994) extends Logic Programming (LP) with variables that can belong to arbitrary constraint domains and the

* Work partially supported by EIT Digital (https://eitdigital.eu), MINECO project TIN2015-67522-C3-1-R (TRACES), and Comunidad de Madrid project S2018/TCS-4339 BLOQUES-CM co-funded by EIE Funds of the European Union.

ability to incrementally solve equations involving these variables. CLP brings additional expressive power to LP, since constraints can very concisely capture complex relationships. Also, shifting from "generate-and-test" to "constraint-and-generate" patterns reduces the search tree and therefore brings additional performance, even if constraint solving is in general more expensive than unification.

The integration of tabling and constraint solvers makes it possible to exploit their synergy in several application fields: abstract interpretation (Swift and Warren 2010), reasoning on ontologies, and constraint-based verification (Gange et al. 2013). In this paper we use Mod TCLP (Arias and Carro 2019a) to adapt PLAI, the fixpoint algorithm implemented in the program analysis, optimization, and transformation tool CiaoPP (Hermenegildo et al. 2012; Hermenegildo et al. 2005). The re-implementation of PLAI uses tabling to reach the fixpoint (following ideas similar to (Kanamori and Kawamura 1993, Janssens and Sagonas 1998)), incremental aggregation techniques (Guo and Gupta 2008; Zhou et al. 2010; Swift and Warren 2010; Arias and Carro 2019b) to join the answers, by discarding the more particular ones, and call entailment checks (Chico de Guzmán et al. 2012; Arias and Carro 2019a) to detect repeated calls (in order to suspend execution to reuse answers from previous calls), thereby speeding up convergence. The resulting code space is reduced to one third and, consequently, increases the maintainability of the abstract interpreter.

2 Related Work

Abstract interpretation has always been seen as one of the most clear applications of tabled logic programming. It requires a fixpoint procedure, often implemented using memo tables and dependency tracking, which play a role very similar to the internal data structures that tabling engines need to detect repeated calls, store and reuse answers, and check for termination.

The relationship between abstract interpretation and tabling was recognized very early. *Extension tables* (Dietrich 1987) were proposed to record results from the execution of predicates and turn intensional definitions into extensional definitions. Their applications included "improving the termination and completeness characteristics of depth-first evaluation strategies in the presence of recursion". The idea of extension tables were applied as the embryo of SLG resolution and the XSB system. At the same time, abstract interpretation was then viewed as inefficient, and as part of the efforts to make it a practical technique to implement analyzers, tables, but also other ideas such as dependency tracking, were used (Warren et al. 1988), thus making it clear that a common underlying technology could be used in both types of systems.

The next step was to use these components, independently available in tabling systems, to explore how they could be used to build abstract interpreters. Earlier work (Kanamori and Kawamura 1993) explored the possibilities offered by OLDT (Tamaki and Sato 1986) to implement abstract interpretation. Using type inference as the guiding example, it suggests certain changes to OLDT and concludes that it is feasible to do abstract interpretation with OLDT. The paper neither describes an implementation nor reports performance, but it states that the abstract interpreter was implemented and was available. In (Warren 1999) an abstract interpreter written in XSB is presented as one of the applications of tabled Prolog.

https://doi.org/10.1017/S1471068419000383 Published online by Cambridge University Press

However, surprisingly few examples of abstract interpreters implemented using tabling have been presented and evaluated w.r.t. implementations without tabling. One of them is a framework (Janssens and Sagonas 1998) based on abstract compilation that executes the abstract version of the program under analysis, together with domain-dependent abstract operations, which is evaluated using the tabling system XSB and compared with the AMAI and PLAI systems (Janssens et al. 1995; Muthukumar and Hermenegildo 1992). Both systems use abstract interpreters written in Prolog without tabling, but they rely on very different underlying technologies, and with different representations for the abstract domains. From that evaluation, the paper concludes that tabling is a viable infrastructure for abstract interpretation, but concedes that the PLAI fixpoint algorithm was the most efficient abstract interpreter for logic programming available at the moment. The very different underlying infrastructure makes it difficult to use these results to draw meaningful conclusions.

On the other hand, abstract interpretation has been used as a benchmark to compare different implementations and/or scheduling strategies of tabling (Demoen and Sagonas 1998, Freire et al. 2001). Advanced tabled systems and techniques have been proposed to implement more efficient abstract interpreters by using the *least upper bound* operator (Schrijvers et al. 2008) to combine answers, numeric constraint solvers (Chico de Guzmán et al. 2012) to implement the Octagon domain, and the *partial order answer subsumption with abstraction* (Swift and Warren 2010) for cases where, e.g., the program computed does not have a finite model. However, none of them reports performance evaluation against other frameworks.

In this paper we started with PLAI, the state-of-the-art abstract interpreter used by CiaoPP, and re-implemented its fixpoint procedure in Tabled CLP preserving the interface with the rest of the system. Therefore, we can compare some indicators of code complexity (e.g., comparing lines of code, with the assumption that the tabled version is essentially a subset of the original version) and performance on a completely equal footing. This is, to our best knowledge, the first comparison that has these characteristics.

3 Background

In this section we briefly describe Mod TCLP (Arias and Carro 2019a), a generic interface that facilitates the integration of constraint solvers with the tabling engine in Ciao, Aggregate-TCLP (Arias and Carro 2019b), a framework implemented on top of Mod TCLP to incrementally compute lattice-based aggregates, and PLAI, the fixpoint algorithm used by CiaoPP.

3.1 The Mod TCLP framework

Tabled Logic Programming with Constraints (TCLP) (Arias and Carro 2019a, Schrijvers et al. 2008; Cui and Warren 2000) improves program expressiveness and, in many cases, efficiency and termination properties. Let us consider a program to compute distances between nodes in a graph written using tabling (Fig. 1, left). The query `?- dist(a,Y,D),D<K.` would loop under SLD due to the left-recursive rule, while it would terminate under tabling for acyclic graphs.

https://doi.org/10.1017/S1471068419000383 Published online by Cambridge University Press

```
:- table dist/3.

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

(a) Tabling

```
:- 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).
```

(b) TCLP

```
:- table dist(_,_,min).

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

(c) Aggregate-TCLP

Fig. 1: Distance traversal in a graph. Note: The symbols `#>` and `#=` are (in)equalities in CLP.

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 is found to be equal, modulo variable renaming, to a previous generator, the execution of the call is suspended and it is flagged 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. Upon suspension, execution switches to evaluating another untried branch. A branch which does not suspend can generate answers for the initial goal. When a generator finitely finishes exploring all the clauses and all answers are collected, the consumers that depend on it are resumed and fed with the answers of the generator. This may make generators produce new answers which can in turn resume more consumers. This process finishes when no new answers can be generated — i.e., a fixpoint has been reached. Tabling is sound and, for programs with a finite Herbrand model, complete (and, therefore, it always finishes in these cases).

However, in a cyclic graph, `dist/3` has an infinite Herbrand model: every cycle can be traversed repeatedly and create paths of increasing length. Therefore, the previous query `?- dist(a,Y,D), D < K` will not terminate under variant tabling, although the query as a whole has a finite model.

On the other hand, if the integration of tabling and CLP (Fig. 1, center) uses *constraint entailment* (Chico de Guzmán et al. 2012), calls to `dist/3` will suspend if there are previous similar calls that are more general, and only the most general answers will be kept. The query `?- D#<K, dist(a,Y,D)` terminates under TCLP because by placing the constraint `D#<K` before `dist(a,Y,D)`, the search is pruned when the values in `D` are larger than or equal to `K`.

This illustrates the main idea underlying the use of entailment ($\sqsubseteq$) in TCLP: 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 will represent a TCLP goal as $\langle \mathtt{g}, \mathtt{c_g} \rangle$ where g is the call (a literal) and c_g is the projection of the current constraint store onto the variables of the call. For example, $\langle \mathtt{dist(a,Y,D)}, \mathtt{D} > 0 \wedge \mathtt{D} < 75 \rangle$ entails the goal $\langle \mathtt{dist(a,Y,D)}, \mathtt{D} < 150 \rangle$ because $(\mathtt{D} > 0 \wedge \mathtt{D} < 75) \sqsubseteq \mathtt{D} < 150$. The latter is therefore more general (i.e., it is a generator) than the former (a consumer). All the solutions of a consumer are solutions for its generator, since the space of solutions of the consumer is a subset of that of the generator. However, not all answers from a generator are valid for its consumers. For example $\mathtt{Y} = \mathtt{b} \wedge \mathtt{D} > 125 \wedge \mathtt{D} < 135$ 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.

https://doi.org/10.1017/S1471068419000383 Published online by Cambridge University Press

Therefore, the tabling engine has to filter, via the constraint solver, the answers from the generator that are consistent w.r.t. the constraint store of the consumer.

Additionally, the Mod TCLP framework (Arias and Carro 2019a) has been used to implement in Ciao a framework, called Aggregate-TCLP (Arias and Carro 2019b), that incrementally computes aggregates for elements in a lattice. The Aggregate-TCLP framework uses the entailment and join relations in a lattice to define and compute aggregates, and to decide whether some atom is compatible with (i.e., entails) the aggregate. For example, the directive `:- table dist(_,_,min)` (Fig. 1, right), specifies the (aggregate) mode `min` for the third argument. The query `?- dist(a,Y,D)` will in this case terminate because only the shortest distance between two nodes found at every moment is kept, and it will be returned in `D` as a result of the evaluation of the initial call. Other tabling engines implement *answer subsumption* (Swift and Warren 2010) or a restricted form of it via *mode-directed tabling* (Guo and Gupta 2008; Zhou et al. 2010; Wielemaker et al. 2012; Santos Costa et al. 2012), that can be used to compute aggregates. However, answer subsumption, as implemented in XSB, assumes answers to be safe (i.e., ground) and works on non-ground answers only in some cases, so it would in principle not be applicable when answers are constraints. Answer subsumption also performs subsumption only on answers, while Aggregate-TCLP can in addition check entailment for calls. In the case of the TCLP implementation of the abstract interpreter, this makes it possible to reuse answers obtained from calls semantically equivalent (i.e., calls whose associated abstract substitutions differ, but that still represent the same object in the lattice) and/or more general (i.e., that represent an element higher in the lattice hierarchy). Note that in our benchmarks we are using semantic equivalence, since using entailment to detect more general calls would cause a loss of precision as the domains we are using are non-relational. Last, answer subsumption does not provide the freedom to be used with aggregates that cannot be expressed in terms of a lattice, such as `sum/3`, which (Arias and Carro 2019b) can work around.

3.2 The PLAI algorithm

We assume that the reader is familiar with the basic principles of abstract interpretation (Cousot and Cousot 1977; Bruynooghe 1991; Nielson et al. 2005). The PLAI algorithm used by the abstract interpreter of CiaoPP for static analysis extends the fixpoint algorithms proposed by (Bruynooghe 1991) with the optimizations described in (Muthukumar and Hermenegildo 1990). In logic programming, all possible concrete substitutions in the program (i.e., terms to which the variables in that program will be bound at run-time for a given query) can be infinite, which gives rise to an infinite execution tree. The core idea of PLAI is to represent this infinite execution tree by an abstract and-or tree using abstract substitutions to finitely represent the possibly infinite sets of substitutions in the concrete domain. The set of all possible abstract substitutions that a variable can be bound to is the *abstract domain* which is usually a complete lattice (or a complete partial order of finite height).

Domains in PLAI PLAI is domain-independent: new abstract domains can be easily implemented and integrated by using a common interface. The operations required by the domain interface are:

https://doi.org/10.1017/S1471068419000383 Published online by Cambridge University Press

- $\lambda' \sqcup \lambda''$, which gives the LUB of the abstract substitutions λ' and λ''. The LUB operation is defined in terms of the $\sqsubseteq$ relation of the abstract domain.
- `call_to_entry(p(`$\vec{u}$`),C,`λ`)`, where `C` is a clause and `p(`$\vec{u}$`)` is a call. It gives an abstract substitution describing the effects on `vars(C)` of unifying `p(`$\vec{u}$`)` with `head(C)` given an abstract substitution λ for the variables in $\vec{u}$.
- `exit_to_success(`λ`, p(`$\vec{u}$`), C, `β`)` which returns an abstract substitution describing the effect of execution `p(`$\vec{u}$`)` against clause `C`. For this, the variables of the abstract substitution β are renamed taking into account the unification with the terms in `head(C)` and the variables in `p(`$\vec{u}$`)`, and a new abstract substitution is returned updating λ with the new information.
- `extend(`λ`,`λ'`)` which extends abstract substitution λ to incorporate the information in λ' in a way that it is still consistent.
- `project_in(`$\vec{u}$`,`λ`)` which extends the abstract substitution λ so that it refers to all the variables in $\vec{u}$.
- `project_out(`$\vec{u}$`,`λ`)` which restricts the abstract substitution λ to refer only to the variables in $\vec{u}$.

For additional examples of abstract domains integrated in CiaoPP, we refer the reader to (Bueno et al. 2004; Muthukumar and Hermenegildo 1989; Vaucheret and Bueno 2002; Hermenegildo et al. 2012).

And-Or trees and substitutions In PLAI, the abstract and-or tree is constructed using a top-down driven strategy (instead of a bottom-up computation) so that the computation is restricted to what is required for the given query. In the resulting and-or tree, an *and-node* is a clause head `h` whose children are the literals in its body, $\mathtt{p}_1,\ldots,\mathtt{p}_n$, and an *or-node* is a literal, $\mathtt{p}_i$, whose children are the heads $\mathtt{h}_1,\ldots,\mathtt{h}_m$ of the clauses that unify with $\mathtt{p}_i$. Its construction starts with the abstract call substitution for the query. Then, abstract substitutions at all points of the abstract and-or tree are computed and finally, the success substitution for the query is computed.

Inside a clause, abstract substitutions at every point are denoted depending on their position among its literals. Given a clause $\mathtt{h}\texttt{:-}\mathtt{p}_1,\ldots,\mathtt{p}_n$, let λ_i and λ_{i+1} be the abstract substitutions to the left and right of the subgoal p_i, $1 \leq i \leq n$. Then, λ_i and λ_{i+1} are, respectively, the *abstract call substitution* and the *abstract success substitution* for the subgoal p_i. The projection of λ_1 on $vars(\mathtt{h})$ is the *abstract entry substitution*, β_{entry}, of the given clause, and, similarly, the projection of λ_{n+1} on $vars(\mathtt{h})$ is its *abstract exit substitution*, β_{exit}. The abstract substitutions for a clause are computed as follows:

- Exit substitution from the entry substitution (Algorithm 1): Given a clause $\mathtt{h}\texttt{:-}\mathtt{p}_1,\ldots,\mathtt{p}_n$ and an entry substitution β_{entry} for the clause head h, the call substitution λ_1 for p_1 is computed by simply adding to β_{entry} an abstraction for the variables in the clause that do not appear in the head. The success substitution for p_1 is λ_2, and it is computed as explained below (essentially, by repeating this same process for the clauses which unify with p_1). $\lambda_3,\ldots,\lambda_{n+1}$ are computed similarly. The exit substitution β_{exit} for this clause is the projection of λ_{n+1} onto $\vec{u}$, the variables in h.

https://doi.org/10.1017/S1471068419000383 Published online by Cambridge University Press

Algorithm 1: `entry_to_exit`: Compute exit substitution from entry substitution.

Data: A clause C of the form $\mathtt{h}(\vec{u})\mathtt{:-p}_1(\vec{u}_1)\mathtt{,}\ldots\mathtt{,p}_m(\vec{u}_m)$; an entry substitution β_{entry}
Result: An exit substitution β_{exit}
$\lambda_1 := \mathtt{project\_in}(vars(C), \beta_{entry})$;
for i := 1 to m **do**
  $\lambda_{i+1} := \mathtt{call\_to\_success}(\mathtt{p}_i(\vec{u}_i), \lambda_i)$;
return $\mathtt{project\_out}(\vec{u},\ \lambda_{m+1})$;

- Success substitution from the call substitution (Algorithm 2): Given a call substitution λ_{call} for a subgoal p, let $h_1, \ldots, h_m$ be the heads of clauses that unify with p. Compute the entry substitutions $\beta 1_{entry}, \ldots, \beta m_{entry}$ for these clauses. Compute their exit substitutions $\beta 1_{exit}, \ldots, \beta m_{exit}$ as explained above. Compute the success substitutions $\lambda 1_{success}, \ldots, \lambda m_{success}$ from the exit substitutions corresponding to these clauses. At this point, all different success substitutions can be considered for the rest of the analysis, or a single success substitution $\lambda_{success}$ for subgoal p computed by means of an aggregation operation for $\lambda 1_{success}, \ldots, \lambda m_{success}$. This aggregate is the least upper bound (LUB), denoted by $\sqcup$, of the abstract domain.

Note that these two procedures are mutually recursive and would not finish in case of mutually recursive calls. They merely describe how abstract substitutions are generated for the case of literals in a body (by carrying success abstract substitutions to call abstract substitutions) and how entry and exit substitutions of several clauses are composed together. For the general case of recursive predicates, where repeated calls and termination have to be detected, PLAI implements a fixpoint algorithm that we sketch below.

PLAI's fix point algorithm The core idea of PLAI's fixpoint algorithm (Muthukumar and Hermenegildo 1990) is that the subtree corresponding to the abstract interpretation of a node with a recursive predicate p should be finite. If the abstract domain is finite, a predicate p can only have a finite number of distinct call substitutions and therefore the subtree can only have a finite number of occurrences of nodes that have a variant of p and which themselves have subtrees. In addition to that, all other nodes in the subtree with the same predicate name p and with the same call substitutions (modulo variable renaming) use the approximate value of the success substitution computed previously for the root node of the subtree labeled with p, and hence they do not have any descendent nodes.

Based on this idea, the fixpoint algorithm iteratively refines the approximate values of the success substitution of the recursive predicate p as follows:

- First, it computes an approximate value of the projected success substitution using the LUB of the projected success substitutions corresponding to the **non-recursive** clauses of p. This provides an initial, hopefully non-empty, abstract substitution that is fast to compute (it does not need to check for repeated calls or termination) and accelerates the convergence of the fixpoint algorithm. In practice, it can be delegated to a specialized version of Algorithms 1 and 2 restricted to non-recursive calls / clauses. These can be determined beforehand by a reachability analysis based on strongly connected components.

https://doi.org/10.1017/S1471068419000383 Published online by Cambridge University Press

Algorithm 2: `call_to_success`: Compute success substitution from call substitution.

Data: A goal $p(\vec{u})$; an abstract call substitution λ_{call}
Result: A success substitution $\lambda_{success}$
λ_{proj} := `project_out(`$\vec{u}$`,`λ_{call}`)`;
$\lambda' := \bot$;
for each clause C which unifies with $p(\vec{u})$ **do**
  β_{exit} := `entry_to_exit(C,call_to_entry(p(`$\vec{u}$`),C,`λ_{proj}`))`;
  $\lambda' := \lambda' \sqcup$ `exit_to_success(`λ_{proj}`,p(`$\vec{u}$`),C,`β_{exit}`)`;
return `extend(`λ_{call}`,` λ'`)`;

- Then, it traverses the (finite) subtree corresponding to p in a depth-first fashion. When an entry-exit combination is needed for a call to p having the same call substitution (modulo variable renaming), the existing approximation is used. For a call to p with a different call substitution, a new (nested) fixpoint computation is started. When the analysis returns to the root of the subtree, the success substitution for p is updated as the LUB of the previous value and the value just computed from the recursive clauses of p.
- If there is a change in the success substitution for p, the depth-first traversal is restarted using the new success substitution, which is used for the subtree nodes corresponding to p that have a compatible call substitution. These depth-first traversal iterations can take place only a bounded number of times, since the LUB operation is monotonic and the abstract substitutions form a lattice of finite height.[1] Therefore, a fixpoint will be reached in a finite number of steps.
- If there is no change in the success substitution for the root node of the subtree of p for a given call substitution, then the analysis of that subtree is complete (for that call substitution) and the fixpoint computation of the predicate p terminates.

For recursive predicates called from within recursive predicates, the dependencies between nested calls have to be recorded to restart the traversal of the subtrees containing predicate calls whose success substitution has been updated.

4 Implementations of the PLAI Algorithm: Prolog vs. Tabling

We will now describe more in depth how the PLAI algorithm is implemented in CiaoPP[2] and highlight the differences w.r.t. the version that uses Tabled CLP.

4.1 PLAI in CiaoPP

The implementation of `call_to_success` is the entry point, as it relates the entry and exit substitutions of a call (in particular, of the top-level call). During the analysis of a goal $p(\vec{u})$, and for each clause that unifies with $p(\vec{u})$, the predicate `call_to_success`

[1] While it is true that abstract domains can be infinite, if convergence is not reached after some time, a widening operation changes the representation of the abstract substitutions to a coarser domain that has more chances to converge (or is sure to converge, if it is finite).
[2] The code is available at www.ciao-lang.org. For the reader convenience, we sketch it in Appendix B of the supplementary material accompanying the paper at the TPLP archive.

https://doi.org/10.1017/S1471068419000383 Published online by Cambridge University Press

invokes `entry_to_exit` which, for each subgoal in the body of the clause, invokes again `call_to_success`. The abstract interpreter is able to stop the evaluation of a part of the program and move to another part to evaluate calls to other predicates. The implementation of PLAI is optimized to accelerate the convergence of the fixpoint and reduce the computation by reusing previous results, among other techniques.

The PLAI algorithm is based on the construction of an and-or tree, described in Section 3.2, with the nodes representing the predicate calls visited during the analysis. To construct this tree, `call_to_success` identifies each goal with its corresponding and/or node and with the specialized version of its father (i.e., the version of the literal that originated the call) and carries around a list with the nodes on which the current goal depends. The analysis starts with a query (a goal) and a call substitution. With this information, `call_to_success` creates the root node of the tree and the list of clauses that unify with the goal. If the goal corresponds to a non-recursive predicate, it computes the success substitution which is asserted in a memo-table to reuse the result later on. Otherwise, the goal corresponds to a recursive predicate and it is dealt with by the fixpoint algorithm: first, it evaluates the non-recursive clauses obtaining an approximation of the success substitution and, after this, it starts the fixpoint computation.

During the fixpoint computation, for a goal with a given call substitution:

- If complete information has been already inferred and saved, `call_to_success` reuses it, to avoid re-computations.
- If it is already inside a fixpoint computation (some parent started a fixpoint with the same call), `call_to_success` reuses the approximation stored for this call, to avoid entering loops.
- If an analyzed call depends on other nodes whose fixpoint are not completed yet, two cases are treated:
  - — If the information on which the predicate depends is updated, a local fixpoint computation is started.
  - — Otherwise, nothing is done.

  To decide whether updated information for a node is available, the information inferred for it has a version number:
  - — When the information on a node is updated, its version number is increased by one.
  - — When a node uses information from another node, it stores the version of that information in the list of nodes on which it depends.

  Version numbers are used to detect updates of the information on which a node analysis depend. If the version number of the last information used from a node does not match its current version number, there has been an update that needs to be propagated.

When the fixpoint computation finishes and the list of dependent nodes is empty, the current information for this call is asserted. Otherwise, if this list is not empty, the information remains flagged as an approximation and the fixpoint restarts. As it can easily be seen, while the algorithm can be conceptually not too complex, its implementation is

https://doi.org/10.1017/S1471068419000383 Published online by Cambridge University Press

cumbersome and at points costly, since many interactions are done through the database using identifiers for program points.

4.2 The PLAI Algorithm in TCLP

The PLAI code using tabling is a simplification of the corresponding Prolog implementation. The main points that were changed are:

- The handling of dependencies among nodes and the detection of termination in the fixpoint computation, that were explicit in the Prolog version, are now transferred to the underlying fixpoint of the tabling engine.
- The calculation of the LUB of the abstract substitutions generated by different clauses unifying with a call is done via lattice-based constraint aggregation (which is in turn built upon tabling).

4.2.1 Internal Database and Dependencies

In the Prolog implementation, the information related to the abstract substitutions is kept in a dynamic database relating code, program points, entry/exit substitutions, and dependencies. This makes it globally accessible and allows it to survive across backtracking and calls, so that it does not need to be carried around the program and be rebuilt every time there is a change in the substitution at a program point.

However, making the abstract interpreter update that information, switch among calls, and re-analyze calls needs accessing and updating this database, which is costly and mixes declarative and imperative styles. On top of that, the CiaoPP implementation has been fine-tuned during many years to avoid unnecessary (re-)analyses and minimize the overhead of accessing the database. All of these optimizations cause the code to have to deal with specific cases for the sake of performance, hence adding to its complexity. But despite the involved implementation, this machinery mimics, at Prolog level, an infrastructure similar to a tabling engine, but specialized for a given program —the abstract interpreter— and with optimizations specific for the task at hand.

This bookkeeping becomes unnecessary when using a tabling-based implementation. An abstract interpreter written using tabling and equipped with the capability to detect when two syntactically different substitutions represent the same object, can automatically take care of termination, suspend analysis when repeated calls are detected, and resume them when new information is available — all of it as part of the normal execution of a tabled program, without having to explicitly update and check dependencies.

That makes the code much simpler (no dependencies, lists of pending goals, resuming, etc. need to be explicitly coded) and shorter (we have obtained a threefold reduction in code size). On the other hand, the tabling engine is generic and cannot decide which suspension and/or resumption policy is better for a particular application. We on purpose chose to (a) keep the TCLP code simple and not include any specific heuristic in the code, (b) not to reimplement an analyzer from scratch, but simplify existing code, and (c) keep exactly the same interfaces (both those offered to the rest of CiaoPP and those required by the fixpoint code) so that the TCLP-based abstract interpreter can interoperate with the rest of the CiaoPP machinery as a drop–in replacement with close to zero effort. For

https://doi.org/10.1017/S1471068419000383 Published online by Cambridge University Press

```
call_to_success(SgKey,Call,Proj,Sg,Sv,AbsInt,Succ) :-
        call_to_success_fixpoint(SgKey,Sg, st(Sv,Call,Proj,AbsInt,Prime) ),
        each_extend(Sg,Prime,AbsInt,Sv,Call,Succ).

:- use_package(tclp_aggregate).
:- table call_to_success_fixpoint(_,_,abst_lub).
call_to_success_fixpoint(SgKey,Sg, st(Sv,Call,Proj,AbsInt,Prime) ) :-
        trans_clause(SgKey,_,Clause),
        do_nr_cl(Clause,Sg,Sv,Call,Proj,AbsInt,Prime).
call_to_success_fixpoint(SgKey,Sg, st(Sv,_Call,Proj,AbsInt,Prime) ) :-
        \+ trans_clause(SgKey,_,_),
        apply_trusted0(Proj,SgKey,Sg,Sv,AbsInt,_ClId,Prime).
```

Fig. 2: Implementation of `call_to_success/7` under the TCLP framework.

these and other reasons, our performance figures (Section 5) are a lower bound of what could be achieved.

As an example, the implementation of `call_to_success/13` in Prolog checks several cases: if the call being analyzed is complete, under evaluation in a fixpoint, a call to a recursive predicate, a call to a non-recursive predicate, etc. to update information accordingly. It eventually invokes `proj_to_prime_nr/9`, which starts the fixpoint computation itself, and which recursively calls `call_to_success/13`. `call_to_success/13` has eight clauses and `proj_to_prime_nr/9` has six clauses (see Appendix B of the supplementary material accompanying the paper at the TPLP archive or the corresponding file at http://www.cliplab.org/papers/tclp-plai-iclp2019).

In the tabling implementation, the underlying engine and the calls to the abstract domain operations through the constraint solver interface take care of these cases and dependencies. This makes the implementation of `call_to_success` have just **one** clause (Fig. 2). The counterpart to `proj_to_prime_nr/9` (which we renamed `call_to_success_fixpoint/3` for clarity) has just two clauses: one for user predicates and another one for library and builtin predicates.

Additionally, the use of tabling makes it unnecessary to save explicitly all the intermediate substitutions, database identifiers for calls and program points, dependencies among goals, etc. This reduces the number of arguments, and `call_to_success` went from thirteen used in Prolog:

```
call_to_success(RFlag,SgKey,Call,Proj,Sg,Sv,AbsInt,ClId,Succ,List,F,N,Id)
```

to seven in the tabling-based implementation:

```
call_to_success(SgKey,Call,Proj,Sg,Sv,AbsInt,Succ)
```

4.2.2 Deciding Termination and Computing the LUB

In the PLAI algorithm, the different exit substitutions obtained from the clauses that unify with a given call are combined using the LUB operator of the abstract domain (Algorithm 2): exit substitutions $\beta_{i\,exit}$, for every clause C_i are joined to return the success substitution $\lambda_{success}$.

The CiaoPP implementation uses `bagof/3` to collect all the clauses in a list and then traverses it and analyzes every clause to create another list of abstract substitutions that are joined with the LUB. This processing is conceptually simple, but its implementation

https://doi.org/10.1017/S1471068419000383 Published online by Cambridge University Press

```
1 call_entail(abst_lub, st(Sv,_,ProjA,AbsInt,_), st(Sv,_,ProjB,AbsInt,_) ) :-
2         identical_abstract(AbsInt,ProjA,ProjB).
3 answer_entail(abst_lub, st(Sv,_,_,AbsInt,PrimeA), st(Sv,_,_,AbsInt,PrimeB) ) :-
4         less_or_equal(AbsInt,PrimeA,PrimeB).
5 answer_join(abst_lub,st(Sv,_,_,Abs, A), st(Sv,_,_,Abs, B), st(Sv,_,_,Abs,New) ) :-
6         compute_lub(Abs,[A,B],New).
7 apply_answer(abst_lub, st(Sv,_,_,AbsInt,Prime), st(Sv,_,_,AbsInt,Prime) ).
```

Fig. 3: Code of the operator `abst_lub` under the TCLP framework.

obscures the code with low-level operations, does not match the idea of having an interpreter executing on an abstract domain, and requires database accesses to retrieve the substitution applicable at that point.

In our implementation, the use of lattice-based aggregates with the tabling engine (Arias and Carro 2019b) simplifies the code. The `abst_lub` identifier in line 6 of Fig. 2 is the name of an interface that has several missions: determine suspension of calls, detect termination of the fixpoint, and perform aggregation of abstract substitutions. In the same line, the underscores state that the corresponding arguments are to be checked for equality (necessary to decide whether a fixpoint has been reached) using the *variant* policy, i.e., syntactical equality modulo variable renaming.

The implementation of the interface named `abst_lub` in Fig. 3 tells the tabling engine how to treat the argument selected previously with this identifier. In particular, the tabling engine checks the corresponding arguments for equality by calling `call_entail/3`. In our case, two abstract substitutions are termed equal if the abstract domain implementation (`identical_abstract/3`) decides so. This makes it possible to detect that two different representations correspond to the same object in the lattice and, if so, suspend a call or retrieve saved answers for it.

The code in Fig. 3 also aggregates the results returned in the third argument (the abstract substitutions) by joining them with the LUB of the lattice. The tabling engine calls `answer_entail/3` to decide whether a new answer (a substitution) is or not more general than an existing answer (`less_or_equal/3`). If its not comparable, `answer_join/4` (which in turn invokes `compute_lub/3`) is called to compute the LUB of a previous answer and the new one. With these definitions, lines 7 to 12 in Fig. 2 contain **all** the code necessary to return the exit substitution of a call w.r.t. all its matching clauses. The implementation of the LUB operation (`abs_lub`, Fig. 3) is based on the operations provided by the abstract domain implementation.

This code also performs an incremental computation of the LUB as follows: upon success, the first answer, corresponding to the exit substitution $\beta 1_{exit}$, is stored in the answer table of the tabled predicate. Let us call this stored answer β_{exit}. For the subsequent exit substitutions $\beta\, i_{exit}, i > 1$, there are two possible cases: if the saved substitution is more general ($\beta\, i_{exit} \sqsubseteq \beta_{exit}$), then $\beta\, i_{exit}$ is discarded; otherwise we make $\beta_{exit} = \beta_{exit} \sqcup \beta\, i_{exit}$.

4.2.3 Connecting Abstract Substitutions with Lattice-Based Aggregates

The TCLP system handles entailment, aggregation, etc. by delegating operations to an underlying constraint solver using a fixed interface (Arias and Carro 2019a). Since we purposely did not change the representation of the CiaoPP abstract domains (they are

https://doi.org/10.1017/S1471068419000383 Published online by Cambridge University Press

used in other parts of the system), we constructed a bridge between these domains and the interface that TCLP expects.

The original entry point of the fixpoint, `proj_to_prime_nr/9` (renamed as `call_to_success_fixpoint/3` in the TCLP implementation), now tabled, is automatically rewritten (by the package `tclp_aggregate`) to call an auxiliary predicate that, at run time, substitutes the arguments carrying abstract substitutions by attributed variables (Holzbaur 1992) that simulate having a constrained variable. Their attributes are tuples that contain (a) the identifier (`abst_lub`, in our example) that determines the interface to be used and (b) the abstract substitution and ancillary information necessary by the abstract interpreter.

When one operation of the tabling engine involves a call with attributed variables, the engine checks if it has an attribute with contents it recognizes. If so, it calls the corresponding predicate from the interface that, in our case, operates on the substitution stored in the attributes.

5 Evaluation

Besides simplifying code, the implementation of PLAI using TCLP gives performance advantages in many cases. These come mainly because part of the bookkeeping related to dependencies, saving the analysis state when restarting the analysis of a dependent call, checking for termination, etc. are handled at a lower level. On the other hand, the implementation currently in CiaoPP, as commented before, has been fine-tuned and specialized during many years to minimize the overhead of the fixpoint implementation, so that a large proportion of the analysis time is spent in domain-related operations. On top of that, the CiaoPP domain representation and domain operations are designed to work well with its current architecture and coding decisions (e.g. saving and retrieving from the dynamic databases) and are suboptimal for a tabling-based implementation: for example, redundant data is manipulated and/or stored. As commented earlier, we did not change any of these so the TCLP fixpoint can seamlessly interact with the rest of the CiaoPP tool, exposing and using exactly the same interfaces.

Even with these constraints, we observed speedups when analyzing most programs from a benchmark set. We used the *Groundness* and *Sharing+Freeness* (Muthukumar and Hermenegildo 1991) domains due to their relevance (e.g., for program optimization and correctness of parallelization). *Groundness* (see Table 1 for performance results) determines if some program variable will be bound to a ground term. This is useful to derive modes, optimize unification, and improve the precision of the *Sharing+Freeness* analysis, among others.

Sharing+Freeness (see Table 2) determines if two (or more) program variables may be bound to terms sharing a common variable. It is useful to determine, for example, whether running two goals in parallel may try to bind the same variable, thus causing races and compromising correctness. The benchmarks used are standard programs that have been previously used to evaluate CiaoPP.

All the experiments in this paper were performed on a Linux 5.0.0-13-generic machine with an Intel Core i7 at 1.80GHz with 16Gb of memory and using `gcc 8.3.0` to compile the abstract machine of Ciao Prolog. In all cases, every program was analyzed 40 times and the 10 worst times were discarded, both when using the tabling and the Prolog

https://doi.org/10.1017/S1471068419000383 Published online by Cambridge University Press

Table 1: Performance comparison: CiaoPP fixpoint in Prolog and TCLP (*Groundness* domain).

	Speedup	TCLP (ms)	CiaoPP (ms)
fibf_alt	1.60	**0.29**	0.46
aiakl	1.56	**2.45**	3.82
boyer	1.50	**7.31**	10.97
pv_queen	1.46	**0.74**	1.07
subst	1.41	**0.25**	0.35
pv_gabriel	1.37	**3.65**	4.99
rdtok	1.32	**7.03**	9.25
mmatf	1.24	**0.31**	0.39
hanoi	1.22	**0.53**	0.65
revf_lin	1.20	**0.27**	0.32
append	1.20	**0.17**	0.20
rev_lin	1.19	**0.26**	0.31
prefix	1.16	**0.27**	0.31
revf	1.15	**0.32**	0.37
pv_plan	1.15	**1.94**	2.23
sublist_app	1.14	**0.24**	0.27
reverse	1.14	**0.38**	0.43
flatten	1.13	**0.55**	0.62
palindro	1.12	**0.34**	0.38
fact	1.08	**0.25**	0.27
rotate	1.06	**0.46**	0.49
maxtree	0.98	0.63	**0.61**
zebra	0.92	1.38	**1.26**
browse	0.89	1.76	**1.57**
AVG	1.31	31.78	41.59

implementation, to try to minimize the effect of spurious interruptions, O.S. scheduling, etc. that can introduce noise in the execution. The remaining times were averaged. All the code and the system under evaluation is available at http://www.cliplab.org/papers/tclp-plai-iclp2019.

The average speedups in each table were calculated by adding up the (averaged) execution times for all the benchmarks and dividing the *CiaoPP* time by the *TCLP* time. This shows that, on average, the analysis with the *Groundness* domain speeds up a bit more than 30%, while the analysis with the *Sharing+Freeness* has experienced, on average, a slight slowdown (about 3%).

By looking at every benchmark in isolation, we can observe that the speedups differ greatly among them. We have sorted the benchmarks according to the speedup to appreciate better the differences. In both cases, only a small part of the benchmarks (three) experienced a slowdown, and even in these cases, the maximum slowdown was about 10%. In the case of *Sharing+Freeness*, the slowest analysis corresponded as well to the largest execution time (larger than the rest of the benchmarks combined). We want to note that this benchmark (zebra) is probably not a representative of a typical program, as it is a combinatorial problem with many free variables in a single clause, some of which are aliased with each other.

https://doi.org/10.1017/S1471068419000383 Published online by Cambridge University Press

Table 2: Performance comparison: CiaoPP fixpoint in Prolog and TCLP (*Sh+Fr* domain).

	Speedup	TCLP (ms)	CiaoPP (ms)
fact	1.30	**0.26**	0.33
pv_queen	1.23	**1.21**	1.49
mmatf	1.17	**0.51**	0.60
mmatrix	1.15	**0.53**	0.61
prefix	1.14	**0.46**	0.52
revf	1.12	**0.47**	0.53
revf_lin	1.10	**0.39**	0.43
reverse	1.10	**0.39**	0.43
rev_lin	1.10	**0.38**	0.42
rotate	1.06	**0.72**	0.76
pv_pg	1.01	**2.67**	2.70
append	0.98	1.11	**1.09**
sublist_app	0.96	0.87	**0.84**
zebra	0.91	16.34	**14.80**
AVG	0.97	26.31	25.55

The source of the speed difference is not easy to determine. A profile of the number of fixpoint calls in CiaoPP vs. fixpoint calls, entailment checks, joins, etc. in the TCLP version does not seem to show a correlation with the observed speedups. We therefore conjecture that the shape and size of the abstract substitution, and the relative cost of checking entailment, has to be explored to have a better explanation of the differences observed.

6 Conclusions and Future Work

We have presented a re-implementation of PLAI, a fixpoint computation algorithm for abstract interpretation, using tabled constraint logic programming. The resulting code is considerably shorter than the current Prolog implementation of PLAI in CiaoPP (one-third of its size) and much simpler: all the bookkeeping necessary to keep track of dependencies between predicates, analysis restarting, etc. is in charge of the tabling engine, which increases the maintainability of the implementation of PLAI.

We have evaluated its performance using several benchmarks and abstract domains, and compared it with the original implementation in CiaoPP. In most cases, the TCLP implementation showed improved performance, sometimes with a speedup of 60%. In a few cases there was a small slowdown, which we think is a reasonable price to pay for the added code clarity, especially taking into account that there is room for improvement in the current implementation.

Among the immediate future plans, we want to experiment re-implementing the abstract domains with an optimized representation of the abstract substitutions, and also use constraint logic programming techniques to propagate the effects of updates. We also expect that, using constraints, we will be able to define widening heuristics independently of the fixpoint algorithm thereby increasing the resulting flexibility, precision and performance w.r.t. the state of the art.

https://doi.org/10.1017/S1471068419000383 Published online by Cambridge University Press

Acknowledgements

We would like to thank Maximiliano Klemen, who helped us understand the intricacies of the CiaoPP implementation of PLAI. Thanks are also due to Manuel Hermenegildo, who gave us very valuable feedback on the paper manuscript and also a historical account on the early relationship between tabling and efficient abstract interpretation implementations.

Supplementary material

To view supplementary material for this article, please visit https://doi.org/10.1017/S1471068419000383.

References

Arias, J. and Carro, M. 2019a. Description, Implementation, and Evaluation of a Generic Design for Tabled CLP. *Theory and Practice of Logic Programming 19,* 3, 412–448.

Arias, J. and Carro, M. 2019b. Incremental evaluation of lattice-based aggregates in logic programming using modular TCLP. In *Practical Aspects of Declarative Languages - 21st International Symposium (PADL 2019)*, J. J. Alferes and M. Johansson, Eds. Lecture Notes in Computer Science, vol. 11372. Springer, 98–114.

Bruynooghe, M. 1991. A Practical Framework for the Abstract Interpretation of Logic Programs. *Journal of Logic Programming 10*, 91–124.

Bueno, F., Lopez-Garcia, P., and Hermenegildo, M. V. 2004. Multivariant Non-Failure Analysis via Standard Abstract Interpretation. In *FLOPS'04*. Number 2998 in LNCS. Springer-Verlag, 100–116.

Chico de Guzmán, P., Carro, M., Hermenegildo, M. V., and Stuckey, P. 2012. A General Implementation Framework for Tabled CLP. In *Int'l. Symposium on Functional and Logic Programming (FLOPS'12)*. Number 7294 in LNCS. 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)*. ACM Press, 238–252.

Cui, B. and Warren, D. S. 2000. A system for Tabled Constraint Logic Programming. In *Int'l. Conference on Computational Logic*. LNCS, vol. 1861. 478–492.

Demoen, B. and Sagonas, K. 1998. CAT: The Copying Approach to Tabling. In *Programming Language Implementation and Logic Programming*. Lecture Notes in Computer Science, vol. 1490. Springer-Verlag, 21–35.

Dietrich, S. W. 1987. Extension Tables: Memo Relations in Logic Programming. In *Fourth IEEE Symposium on Logic Programming*. 264–272.

Freire, J., Swift, T., and Warren, D. S. 2001. Beyond Depth-First: Improving Tabled Logic Programs through Alternative Scheduling Strategies. In *International Symposium on Programming Language Implementation and Logic Programming*. Number 1140 in LNCS. Springer-Verlag, 243–258.

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.

Guo, H.-F. and Gupta, G. 2008. Simplifying Dynamic Programming via Mode-directed Tabling. *Software: Practice and Experience* 1 (Jan), 75–94.

Hermenegildo, M. V., Bueno, F., Carro, M., Lopez-Garcia, 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. http://arxiv.org/abs/1102.5497.

Hermenegildo, M. V., Puebla, G., Bueno, F., and Lopez-Garcia, P. 2005. Integrated Program Debugging, Verification, and Optimization Using Abstract Interpretation (and The Ciao System Preprocessor). *Science of Computer Programming 58,* 1–2 (October), 115–140.

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

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

Janssens, G., Bruynooghe, M., and Dumortier, V. 1995. A blueprint for an abstract machine for abstract interpretation of (constraint) logic programs. In *ILPS*. 336–350.

Janssens, G. and Sagonas, K. 1998. On the Use of Tabling for Abstract Interpretation: An Experiment with Abstract Equation Systems. In *Tabulation in Parsing and Deduction*.

Kanamori, T. and Kawamura, T. 1993. Abstract Interpretation Based on OLDT Resolution. *Journal of Logic Programming 15*, 1–30.

Muthukumar, K. and Hermenegildo, M. 1989. Determination of Variable Dependence Information at Compile-Time Through Abstract Interpretation. In *1989 North American Conference on Logic Programming*. MIT Press, 166–189.

Muthukumar, K. and Hermenegildo, M. 1990. Deriving A Fixpoint Computation Algorithm for Top-down Abstract Interpretation of Logic Programs. Technical Report ACT-DC-153-90, Microelectronics and Computer Technology Corporation (MCC), Austin, TX 78759. April.

Muthukumar, K. and Hermenegildo, M. 1991. Combined Determination of Sharing and Freeness of Program Variables Through Abstract Interpretation. In *International Conference on Logic Programming (ICLP 1991)*. MIT Press, 49–63.

Muthukumar, K. and Hermenegildo, M. 1992. Compile-time Derivation of Variable Dependency Using Abstract Interpretation. *JLP 13,* 2/3 (July), 315–347.

Nielson, F., Nielson, H. R., and Hankin, C. 2005. *Principles of Program Analysis*. Springer. Second Ed.

Santos Costa, V., Rocha, R., and Damas, L. 2012. The YAP Prolog system. *Theory and Practice of Logic Programming 12,* 1-2, 5–34.

Schrijvers, T., Demoen, B., and Warren, D. S. 2008. TCHR: a Framework for Tabled CLP. *Theory and Practice of Logic Programming* 4 (Jul), 491–526.

Swift, T. and Warren, D. S. 2010. Tabling with answer subsumption: Implementation, applications and performance. In *Logics in Artificial Intelligence*. Vol. 6341. 300–312.

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

Vaucheret, C. and Bueno, F. 2002. More Precise yet Efficient Type Inference for Logic Programs. In *9th International Static Analysis Symposium (SAS'02)*. Lecture Notes in Computer Science, vol. 2477. Springer-Verlag, 102–116.

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

Warren, D. S. 1999. Programming in Tabled Prolog. https://www3.cs.stonybrook.edu/~warren/xsbbook/book.html. Unpublished manuscript. Accessed on May 15, 2019.

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*. MIT Press, 684–699.

Wielemaker, J., Schrijvers, T., Triska, M., and Lager, T. 2012. SWI-Prolog. *Theory and Practice of Logic Programming 12,* 1-2, 67–96.

Zhou, N.-F., Kameya, Y., and Sato, T. 2010. Mode-Directed Tabling for Dynamic Programming, Machine Learning, and Constraint Solving. In *Int'l. Conference on Tools with Artificial Intelligence*. Number 2. IEEE, 213–218.

https://doi.org/10.1017/S1471068419000383 Published online by Cambridge University Press