An Approach to the Abstract Interpretation of Goal-Directed Answer Set Programming *

DANIEL JURJO-RIVAS
Universidad Politécnica de Madrid (UPM) & IMDEA Software Institute, Madrid, Spain
daniel.jurjo@alumnos.upm.es

JOAQUÍN ARIAS
CETINIA, Universidad Rey Juan Carlos, Madrid, Spain
joaquin.arias@urjc.es

GOPAL GUPTA
University of Texas at Dallas, Richardson, USA
gupta@utdallas.edu

JOSE F. MORALES
Universidad Politécnica de Madrid (UPM) & IMDEA Software Institute, Madrid, Spain
josefrancisco.morales@upm.es

PEDRO LÓPEZ-GARCÍA
Spanish Council for Scientific Research (CSIC) & IMDEA Software Institute, Madrid, Spain
pedro.lopez@csic.es

MANUEL V. HERMENEGILDO
Universidad Politécnica de Madrid (UPM) & IMDEA Software Institute, Madrid, Spain
manuel.hermenegildo@upm.es

Abstract Interpretation infers and verifies program properties by over-approximating program semantics. It has been highly successful for (Constraint) Logic Programming, enabling the analysis of determinism, types, aliasing, and resource usage, as well as application in verification and program optimization. However, Abstract Interpretation has not yet been studied in the context of Goal Directed Answer Set Programming (ASP). In this work, we take a first step in this direction. We present a top-down algorithm based on the PLAI fixpoint, implemented in the abstract interpreter of the Ciao Prolog Preprocessor, to perform abstract interpretation of goal-directed ASP. We also introduce the Shared-Constraints abstract domain, designed to capture potential relations among variables induced by constraints. Finally, we study the practicality of the approach in s(CASP) through three applications: detection of false odd loops over negation, efficient *forall* evaluation enabled by the Shared-Constraints domain, and abstract specialization (including the simplification of required global constraints). Our results show that compile-time static analysis can improve the evaluation of goal-directed ASP programs.

1 Introduction

Abstract Interpretation [7] allows constructing sound static analysis tools that can extract properties of a program by safely approximating its semantics. Abstract interpretation-based analysis has been shown to be practical and effective in the context of (Constraint) Logic Programming ((C)LP) in both verification and program optimization. Classic abstract interpretation is based on fixpoint algorithms that infer semantic information by interpreting programs over abstract domains, such that the computed fixpoint represents a sound over-approximation of all possible concrete executions. Answer Set Programming

*Partially funded by MCIN/AEI 10.13039/501100011033 project COSASS (PID2021-123673OB-C32); MICIU/AEI 10.13039/501100011033 and FEDER/EU project EVASAI (PID2024-158227NB-C32); a research gift from Nexco Corp; MICIU/AEI/10.13039/501100011033 Grant CEX2024-001471-M; MICIU project CEX2024-001471-M *María de Maeztu*; and by the European Union GA 101154447 NEAT. We also thank the anonymous reviewers for their comments and suggestions for improvement.

```
1  q(X,Y) :-
2       X #> 5,
3       Y = a.
4  r(X) :-
5       X #< 1.
6  p(X) :-
7       not q(Y,X),
8       r(Y).
9  % DUAL
10 not q(X,Y) :-
11      not q_1(X,Y).
12 not q_1(X,Y) :-
13      Y \= a.
14 not q_1(X,Y) :-
15      Y = a,
16      X #=< 5.
17
18 not r(X) :-
19      not r_1(X).
20 not r_1(X) :-
21      X #>= 1.
22
23 not p(X) :-
24      not p_1(X).
25 not p_1(X) :-
26   forall(Y,not p_1(X,Y)).
27 not p_1(X,Y) :-
28      q(Y,X).
29 not p_1(X,Y) :-
30      not q(Y,X),
31      not r(Y).
```

Figure 1: Simple complete program, including its dual rules.

(ASP), based on the stable model semantics of [11], has attracted much attention due to its expressiveness, and its ability to incorporate non-monotonicity, represent knowledge, and model combinatorial problems. The s(CASP) system implements ASP with constraints using a goal-directed, top-down execution model that retains logical variables both during execution and in the answer sets. However, Abstract Interpretation has not yet been studied in the context of Answer Set Programming (see Section 6 for a discussion of related work). Since the goal-directed execution model of ASP is closer to that of (C)LP, it seems worthwhile to explore whether top-down analysis techniques via Abstract Interpretation, such as those developed for (C)LP, can be applied to ASP. In this paper, we present an abstract interpretation framework for goal-directed ASP which adapts the top-down abstract interpretation framework of PLAI for (C)LP. We also introduce the Shared-Constraints abstract domain, designed to capture potential relationships among variables induced by constraints. We illustrate the integration of the proposed framework with s(CASP) and evaluate how the information extracted at compile-time can be leveraged to improve execution performance. Finally, we discuss some related work and present our conclusions.

2 Preliminaries and Notation

Negation is encoded as default negation `not p` or as classical negation `-p`. While `not p` succeeds if the program cannot prove that `p` holds, `-p` succeeds if there is a rule that explicitly states how to deduce `-p`. A *constraint* is a conjunction of expressions built from predefined predicates whose arguments are constructed using predefined functions and variables, e.g., `X-Y #> 5`. An ASP program extended with constraints under a goal-directed execution is a set of clauses of the form:

$$\texttt{a :- } c_a\texttt{,}b_1\texttt{,...,}b_m\texttt{,not } b_{m+1}\texttt{,...,not } b_n\texttt{.}$$

where c_a is a conjunction of constraints, and $a, b_1, \ldots, b_n$ are atoms.

Definition 1 (Dual and Dual Program) *The dual of a predicate `p/n` is another predicate, say, `not p/n`, that returns $\vec{X}$ such that `p`$(\vec{X})$ is not true, i.e., it provides a constructive definition of `not p/n`. The dual of a logic program P is another logic program containing the dual of each predicate in P. We say a program is complete if for every `p` $\in P$, `not p` $\in P$.* ◁

To synthesize the dual of a logic program P, Clark's completion [5] is computed, and then De Morgan's laws are applied. A detailed discussion can be found in [3, 2]. The evaluation of dual rules requires a disequality constraint, `\=`, to handle the dual of equality constraints (e.g., unifications), and a predicate `forall/2`, to handle the dual of existential quantifiers. Fig. 1 shows a complete ASP program. Without loss of generality, we assume programs to be complete and predicates to be normalized. The goal-directed, top-down evaluation of an ASP program has similarities with SLD resolution. The computation is started by a *query*. A query is a literal `?-` $c_q\texttt{,}l_1\texttt{,...,}l_m$, where the l_i are (possibly negated) atoms and

c_q is a conjunction of constraints. The goal-directed, top-down evaluation also takes into account specific characteristics of ASP and the dual programs, such as the different kinds of loops that may appear.

Definition 2 (Loops and Loops Over Negation) *A loop is an execution trace of the form: $p(\vec{X}) \rightsquigarrow^* p(\vec{Y})$ such that solve$(p(\vec{X}) = p(\vec{Y}))$ succeeds, i.e., $p(\vec{X})$ entails/is entailed by $p(\vec{Y})$. If no negations are interleaved, it is called a positive loop, and execution backtracks to avoid non-termination. Otherwise, loops are classified according to the number of interleaving negations:*

- *Odd loop over negation: with an odd number of interleaving negations. In this case execution backtracks to avoid contradictions of the form $p \wedge \neg p$.*
- *Even loop over negation: when there is an even number of negations, as in* `p :- not q  q :- not p`*, multiple models are generated, such as* `{p}` *and* `{q}`*.*

◁

Abstract Interpretation. The main idea behind Abstract Interpretation [7] is to interpret the program over an abstract domain whose elements are finite representations of possibly infinite sets of actual states in the concrete program. We denote the concrete domain as D_γ, the abstract domain as D_α, and the functions that relate sets of concrete states with abstract states as the *abstraction* function $\alpha : D_\gamma \to D_\alpha$ and the *concretization* function $\gamma : D_\alpha \to D_\gamma$. The concrete domain is typically a complete lattice with the set inclusion order which induces an ordering relation in the abstract domain represented by $\sqsubseteq$. Under this relation the abstract domain is usually a complete lattice and $(D_\gamma, \alpha, D_\alpha, \gamma)$ is a Galois insertion/connection. *Top-down* analyses are a family of static analyses that build an *analysis graph* starting from a series of program *entry points*. This graph is a finite abstract object whose concretization approximates the (possibly infinite) set of possible executions of the concrete semantics.

3 Operational Semantics for Goal Directed ASP

In this work we focus on the execution model of Goal-Directed ASP [16, 3]. We represent the semantics of this model by a tree representation similar to that used for SLD-based resolution [15, 1, 13]. In this representation, the execution of a program is captured by a tree (an AND tree) which represents *resolvents*. A *resolvent* is represented by $(G_1, \ldots, G_n)\Sigma_i$, where G_i are literals and Σ_i is the accumulated state resulting from the composition of the states applied so far. A state is a tuple $\langle \mathscr{P}, \theta \rangle$, where θ represents the *current concrete constraint store* and $\mathscr{P}$ is a sequence of all calls encountered during execution (the *call path*). This path is checked prior to evaluating predicates to avoid inconsistencies and infinite loops.[1] The empty state is defined as $\varepsilon_s = \langle \varepsilon, \varnothing \rangle$ and is the initial resolvent. We assume a complete solver exists, and use *solve* to represent the combination of stores. Finally, given two states $\sigma_1 = \langle \mathscr{P}_1, \theta_1 \rangle$ and $\sigma_2 = \langle \mathscr{P}_2, \theta_2 \rangle$ their composition is defined as $\sigma_1\sigma_2 = \langle \mathscr{P}_1 \cdot \mathscr{P}_2, solve(\theta_1, \theta_2) \rangle$ where $\mathscr{P}_1 \cdot \mathscr{P}_2$ denotes path concatenation. To perform the computation, the leftmost literal G_1 is selected and the *immediate successor* of $(G_1, \ldots, G_n)\Sigma_i$ is computed depending on $G_1\Sigma_i$ (the *current goal*, denoted G for short). If G is a constraint, the solver computes the *immediate successor* $(G_2, \ldots, G_n)\Sigma_{i+1}$, with $\Sigma_{i+1} = \langle \sigma_i, \theta_{i+1} \rangle$ where θ_{i+1} is the constraint store θ after solving the constraints induced by G. If G is a universal quantification `forall(V,Goal)`, it is evaluated (for instance, by applying the forall algorithm [3]). On success, the *immediate successor* is $(G_2, \ldots, G_n)\Sigma_{i+1}$, with $\Sigma_{i+1} = \Sigma_i$.[2]

[1] This is one of the key differences w.r.t. SLD resolution.

[2] Note that `forall/2` does not update the concrete state upon success.

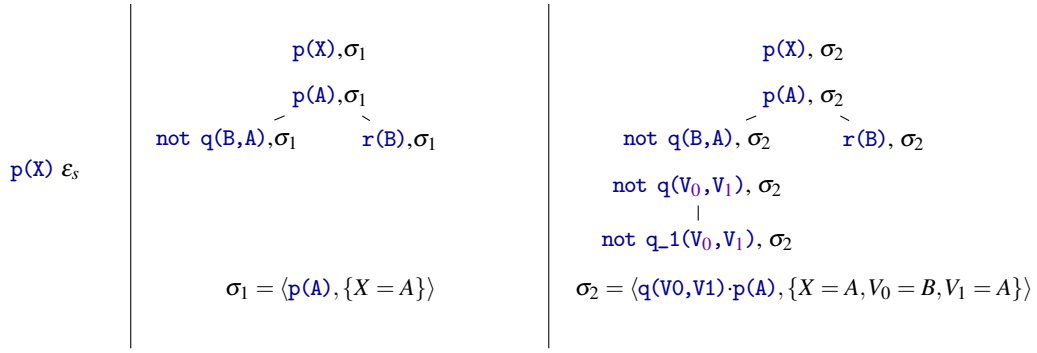

Figure 2: Three consecutive AND trees for the program in Fig. 1

Finally, if G is a predicate, the immediate successor is computed as follows:

$\varnothing$	If `loop`$(G, \mathscr{P}_i)$ returns `odd` or `positive`
$(G_2, \ldots, G_n)\Sigma_{i+1}$	If `loop`$(G, \mathscr{P}_i)$ returns `even` where $\Sigma_{i+1} = \langle \mathscr{P}_i \cdot G, \theta_i, \mathscr{C}_i \rangle$
$(B_1, \ldots, B_m, , G_2, \ldots, G_n)\Sigma_{i+1}$	Otherwise where $H \leftarrow B_1, \ldots, B_n$ is a clause, $H = G$ succeeds with θ_{i+1}, and $\Sigma_{i+1} = \langle \mathscr{P}_i \cdot G, \theta_i\theta_{i+1}, \mathscr{C}_i \rangle$

where the `loop` function inspects the call path ($\mathscr{P}$) in order to detect and classify loops. Concrete states are represented as AND trees, with literals in the state as leaves. A *sequence* $\ldots, t_i, t_j \ldots$ of AND trees is such that t_j is the immediate successor of t_i. The set of all AND trees which can originate from a set of queries completely specifies the procedural behavior of a program for that set of queries.

Fig. 2 shows some consecutive AND trees for the program in Fig. 1 with query `?- p(X)`. Notice that all new variables are propagated during computation.

3.1 A top-down algorithm to construct generalized AND trees

An AND tree contains more information than is necessary for analysis, potentially involving an unbounded number of variables. To address this in the context of (C)LP, Bruynooghe [4] proposes the notion of *generalized* AND *trees* to capture the computation state at each step relative to the current literal. Thus, each node is represented as a triplet $\langle A, \theta^c, \theta^s \rangle$, where A is a predicate, and θ^c and θ^s the call and success states over the predicate's variables. Generalized AND trees are constructed in a top-down manner starting from an initial query Q, annotated with an initial call state θ_0^c, whose domain is a subset of $\mathsf{vars}(Q)$. The rest of the tree is built by expanding the leaf nodes in the following way:
a) If A is a leaf adorned on the left with the call state θ_i^c, then:

i) If $A_k \leftarrow B_1, \ldots, B_n$ is a properly renamed clause, defining A, then the tree is expanded by performing $solve(A = A_k)$ and adding the calls $B_1, \ldots, B_n$ as children of A. B_1 is adorned on the left with the call state θ_{i+1}^c, whose domain is $\mathsf{vars}(A \leftarrow B_1, \ldots, B_n)$. This tree extension is named procedure entry.

ii) If A is a built-in, then the tree is expanded by adorning A on the right with a state θ_i^s, which is the success state of A. The domain of θ_i^s is vars(clause of A). If A is the last call of its clause, θ_i^s is also the success state of the body; otherwise it is the call state of the next call.

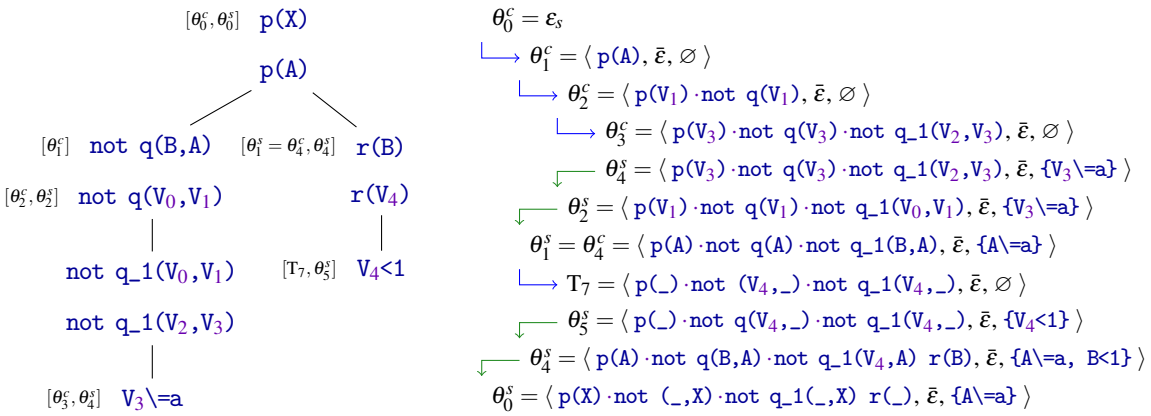

Figure 3: Complete generalized AND tree for the program in Fig. 1 and its construction steps.
NOTE: Blue arrows represent procedure entries while green arrows represent procedure exits.

b) If a node A is adorned on the left with a call state θ_j^c but not adorned with a state on the right, such that A is the parent of a clause body with success state θ_i^s, then the tree is expanded by adorning A on the right with a success state θ_j^s. The domain of θ_j^s is vars(clause of A). With A the last call of its clause (the query), A_j is also the success state of the body (the query); otherwise it is the call state of the next call. The operation extending the tree is named *procedure exit*.

Definition 3 (Operational Semantics $[\![P]\!]_Q$) *The operational semantics $[\![P]\!]_Q$ of a program P for a set of queries Q is the set of all generalized* AND *trees obtained from the execution of Q in P.* ◁

Given a sequence $t_1, \ldots, t_n$ of AND trees, a corresponding generalized AND tree can be built. The initial tree t_1, the query, corresponds to the root node. At each step, the state captured by t_j (restricted to the variables of the current literal) defines the state of the node added to the generalized AND tree. Conversely, given a generalized AND tree, a sequence of AND trees can be reconstructed. Starting from the root node, the initial tree t_1 is obtained, and each subsequent tree is generated by mimicking the steps of the generalized AND tree without projection, entry, or exit procedures.

Example 1 (generalized AND tree) *Fig. 3 presents a complete generalized* AND *tree for the program in Fig. 1 and the query* `?- p(X).` *Starting with call store $\theta_0^c = \varepsilon_s$ the execution succeeds with store $\theta_0^s = \langle$ `p(X)·not q(_,X)·not q_1(_,X)·r(_), ε̄, {X\=a}` $\rangle$. From the tree, it is possible to recover* `{ p(X/{X\=a}), not q(Y, X/{Y#<1,X\=a}), r(Y/{Y#<1}) }`*, the only answer set generated by this program for the given query. Notice how θ_0^c, θ_1^c and θ_2^c correspond to projecting $\varepsilon_s, \sigma_1, \sigma_2$ in the sequence of* AND *trees in Fig. 2.*

4 An Abstract Interpreter for Goal-Directed ASP

In Section 3 we presented the operational semantics for goal directed ASP. In this section we present a top-down approach for the analysis of goal-directed ASP programs. The objective of this analysis is to build an *analysis graph* starting from a series of program entry points.

Definition 4 (Analysis Graph) *An analysis graph is a (directed) call graph and a mapping function from predicate descriptors and call states to success states. A node is a tuple $\langle A, \lambda^c \rangle$, representing that a call to a predicate $\langle A, \lambda^c \rangle$ is possibly made in the concrete program execution and has an associated*

```
1:  function ANALYZE(P,Q^♯)
2:    G ← ∅
3:    for ⟨A,λ^c⟩ ∈ Q^♯ do
4:      G ← CALLTOSUCC(A,λ^c,G,∅)
5:    return G
6:  function CALLTOSUCC(A, λ^c,G,Tr)
7:    λ^pr ← proj^♯(vars(A), λ^c)
8:    Type ← TYPEOFLOOP(A,Tr)
9:    if (Type = even) then
10:     H ← FIRSTAPP(A,λ^pr,Tr)
11:     λ^en ← ENTRYPROC^#(λ^pr, A, H)
12:     λ^p ← EXITPROC^#(λ^en, H, A)
13:     λ^s ← EXTEND^#(λ^c,λ^p)
14:   else if (Type ∈ {noloop,PossOdd}) then
15:     Tr ← APPEND(A,λ^pr,Tr)
16:     λ^p ← ANALYZEPRED(λ^pr,⟨A,λ^c⟩,G,Tr)
17:     λ^s ← EXTEND^#(λ^c,λ^p)
18:     Trace ← REMOVEHEAD(Trace)
19:   else if (Type ∈ {odd,positive}) then
20:     λ^s ← ⊥
21:   else if (Type = PossEven) then
22:     H ← FIRSTAPP(A,λ^pr,Tr)
23:     λ^en ← ENTRYPROC^#(λ^pr, A, H)
24:     λ_1^p ← EXITPROC^#(λ^en,H,A)
25:     λ_2^p ← ANALYZEPRED(λ^pr,⟨A,λ^c⟩,G,Tr)
26:     λ^s ← EXTEND^#(λ^c,λ_1^p ⊔ λ_2^p)
27:   G ← update(G, ⟨A,λ^c⟩ ↦ λ^s)
28:   return λ^s,G
29: function ANALYZEPRED(λ^pr,⟨A,λ^c⟩,G,Tr)
30:   if ⟨A,λ^pr⟩ ↦^fx λ^p ∈ G then
31:     return λ^p
32:   else
33:     λ^p ← ⊥
34:     repeat
35:       λ_0^p ← λ^p
36:       for A_k :- {A_k,j} ∈ P do
37:         λ^en ← ENTRYPROC^#(λ^pr, A, A_k)
38:         λ^en ← AUGMENT^#(vars(A_k,j)\vars(A_k), λ^en)
39:         λ^ex, G ← ENTRYTOEXIT(λ^en,A_k,{A_k,j},⟨A,λ^c⟩,Tr,G)
40:         λ_1^p ← EXITPROC^#(proj(vars(A_k ), λ^ex), A_k, A)
41:         λ^p ← λ^p ⊔ λ_1^p
42:       G ← update(G, ⟨A,λ^pr⟩ ↦^fx λ^p)
43:     until λ_0^p = λ^p
44:     G ← del(G, ⟨A,λ^pr⟩ ↦^fx λ^p)
45:     return λ^p
46: function ENTRYTOEXIT(λ^en,A_k,{A_k,j},⟨A,λ^c⟩,Tr,G)
47:   λ^ex ← λ^en
48:   for A_k,i ∈ {A_k,j} do
49:     if A_k,i ∈ P then
50:       G ← update(G, ⟨A,λ^c⟩ →_k,i ⟨A_k,i,λ^ex⟩)
51:       λ^ex ← CALLTOSUCC(A_k,i, λ^ex,G,Tr)
52:     else if ISFORALL(A_k,j) then
53:       B_∀ ← GETFORALLGOAL(A_k,i)
54:       λ_∀^c ← TOPMOSTCALL(B_∀)
55:       if CALLTOSUCC(B_∀,λ_∀^c,∅,Tr) = ⊥ then
56:         λ^ex ← ⊥
57:       else
58:         λ^ex ← λ^c
59:     else
60:       λ^pr ← PROJ^♯(vars(A_k,i),λ^ex)
61:       λ^ex ← EXTEND^#(λ^ex, builtin^#(A_k,i,λ^pr))
62:   return λ^ex,G
```

Figure 4: A schematic description the analysis algorithm

success abstraction λ^s through the mapping $\langle A,\lambda^c\rangle \mapsto \lambda^s$. An edge in an analysis graph is of the form $\langle A,\lambda^c\rangle \rightarrow_{k,i} \langle B,\eta^c\rangle$. This represents that calling predicate A with calling pattern λ^c may cause B to be called (via the i-th literal in the k-th clause of A, $A_{i,k}$) with calling pattern η^c. ◁

The analysis graph is built to capture the behaviour of the original program by abstracting the set of generalized AND trees of P with a given set of queries $\mathcal{Q}$.

4.1 A PLAI-based abstract interpreter for goal-directed ASP

In order to construct the analysis graph, we modify the PLAI [19, 20, 10] top-down analysis framework, to adapt to the semantics of goal-directed ASP. The algorithm (which is parametric to a given abstract domain D_α) receives as input a program P and a set of initial *abstract queries*, $Q_\alpha = \langle A_i, \lambda_i^c\rangle$, where each A_i is an atom and $\lambda_i^c \in D_\alpha$. Algorithm 4 presents the analysis procedure. For each abstract query, the function CALLTOSUCC is invoked to compute the success abstraction of the corresponding abstract call pattern. First, it determines whether the current call corresponds to a loop via TYPEOFLOOP (Line 8). This function inspects the current trace and determines whether there will always occur a loop (`even`, `odd`, `positive`) or whether there may occur a loop. Since the abstract state over-approximates the concrete state, in most cases it cannot be determined whether a loop is `even` or `odd`.[3] Therefore, loop

[3] In fact, this is only possible if both literals in the trace are ground.

detection is performed syntactically: if loops are possible, the analysis has to assume the least precise (most general) option to happen. Once the case is decided, the analysis proceeds as follows:

- If there are `odd` or `positive` loops, the abstract success (λ^s) is assigned to $\bot$. This reflects that, for any corresponding concrete derivation, the execution will fail (Lines 19–20).
- It there are `even` loops the success abstraction is the result of abstractly joining the call with the looping variant (Lines 9–13).
- If there may be even loops, two prime abstractions are computed. One assuming the even case (by performing the abstract equalities), and another assuming that there is no even loop: continuing with the analysis. Then, the success abstraction is the result of extending with the least upper bound of both abstractions (Lines 21–26).
- If there are no loops (or there may be `odd` loops), analysis proceeds into AnalyzePred.

The function AnalyzePred (Lines 29–45) computes the abstract success of a given predicate. It iterates over *all* the clauses of the predicate,[4] computing for each a prime abstraction (the abstract state after executing the clause body) by calling EntryToExit. The prime abstractions of all the clauses are joined via $\sqcup$ and the result is stored in $\mathscr{G}$. This is repeated until a fixpoint is reached (Lines 34–43). During the computation of the fixpoint, auxiliary edges ($\cdot \overset{fx}{\mapsto} \cdot$) are temporarily added to the graph. Function EntryToExit traverses the body of each clause. For each predicate, an edge is added to $\mathscr{G}$ (Line 50) and then CallToSucc is invoked recursively. For a `forall` literal, the inner goal is analyzed independently with a top-most call abstraction, since the concrete execution is performed with new fresh variables. If the result is $\bot$, meaning that the `forall` invocation fails for any concrete execution, all the subsequent literals are unreachable and the abstraction is set to $\bot$ (Lines 52–58). Finally, for built-ins and constraints, the abstract state is updated by means of $\texttt{builtin}^{\#}$ (60–61).

4.2 Shared-Constraints: An Abstract Domain for ASP + Constraints

Similarly to the *set-sharing* domain [18, 12] for variable aliasing, we introduce the *Shared-Constraints* abstract domain which encodes which variables *may be* constrained at runtime by capturing them within the same *sharing set*. The lattice of the *Shared-Constraints* abstract domain is defined as $S_{ct} = \mathscr{P}(\mathscr{P}(PV))$, where PV is the set of program variables, equipped with the order relation: $Ss_1 \leq Ss_2 \Leftrightarrow \forall S_1 \in Ss_1, \exists S_2 \in Ss_2$ s.t. $S_1 \subseteq S_2$. Thus, given the constraint store $\{X > R, R > Y, Z \neq a\}$, its abstraction is $\{\{X,Y\},\{Z\}\}$. It captures that X may be constrained with Y, while Z is not constrained with either X or Y. We consider the constraint relation between variables to be transitive. For instance, if $\{X,Y\},\{Z,T\}$ are sharing sets, and Z is constrained with X, all variables become mutually constrained, resulting in the sharing set $\{X,Y,Z,T\}$. The abstraction of built-in literals is done by means of the $\texttt{builtin}^{\#}$ function defined as: $\texttt{builtin}^{\#}(L,\lambda) = \{S \in \lambda \mid S \cap \mathsf{vars}(L) = \varnothing\} \cup \{S \in \lambda \mid S \cap \mathsf{vars}(L) \neq \varnothing\}$. When encountering such literals, we conservatively assume that all their variables may become constrained and incorporate this information into the abstraction. For constraints, this definition yields a precise abstraction. For other literals, it provides a safe over-approximation. Precision can be improved by specializing the treatment of selected built-ins and operators. For example, for a call to $\texttt{length}(L,N)$, it can be safely inferred that L and N are not mutually constrained. The definitions of the $\texttt{entryProc}^{\#}$ and $\texttt{exitProc}^{\#}$ operations differ only in the order of equalities. In particular, $\texttt{entryProc}^{\#}$ is defined as $\texttt{entryProc}^{\#}(\lambda, T_1, T_2) = \texttt{asolve}(T_1 = T_2, \lambda)$ where the function `asolve` first computes the solved form of the equality $T_1 = T_2$ and, from this set of equations, invokes $\texttt{builtin}^{\#}$ iteratively until obtaining the resulting abstract constraint. The remaining operations are defined as: $\lambda_1 \sqcup \lambda_2 = \lambda_1 \cup \lambda_2$,

[4] Since given a literal we cannot know (in general) which clause will be executed.

$\texttt{proj}^\sharp(\lambda, Vs) = \{Ss \cap Vs \mid Ss \in \lambda\}$, $\texttt{extend}^\#(\lambda_1, \lambda_2) = \{Ss \cup Pr \mid Ss \in \lambda_1 \wedge Pr = \bigcup\{P \in \lambda_2 \mid P \cap Ss \neq \varnothing\}\}$, and $\texttt{augment}^\#(\lambda, Vs) = \lambda$. The $\texttt{augment}^\#$ operation requires its input variables to be fresh and unconstrained; therefore, the abstraction remains unchanged.

This abstract domain allows capturing dependencies between variables by recording which variables may become mutually constrained during goal evaluation. The resulting information is used in Section 5.1 to reorganize nested `forall` literals at compile time by grouping universally quantified variables that belong to the same dependency class.

5 Some Applications of the Abstract Interpretation of Goal-Directed ASP

This section evaluates the techniques presented and describes applications of abstract interpretation-based analyses in the context of goal-directed ASP. First, we propose an Abstract Interpretation-based transformation for efficient forall evaluations in s(CASP) using the Shared-Constraints abstract domain. Then, we discuss how abstract specialization of goal-directed ASP programs can be perfomed by using already mature abstract domains in the context of (C)LP. Finally, we use the information inferred by the analyzer to detect "false" odd loops over negation, enabling the compiler to skip generating the corresponding global constraints, and avoiding unnecessary consistency checks. Experiments were conducted on a MacBook Air M2 with 16 GB of RAM and 500 GB.

5.1 Efficient forall Evaluation enabled by the Shared-Constraints Domain

One of the main motivations for introducing abstract interpretation in the context of goal-directed ASP is the possibility of extracting compile-time information that can be exploited to improve the execution of the programs. One of the most computationally expensive steps in the execution of s(CASP) programs is the evaluation of forall predicates. The algorithm to handle `forall(V,G)`, initially proposed in [16] and extended in [3] to handle constraints, proceeds by iteratively narrowing the constraint store C_i under which the goal `G` is being evaluated (initially the constraint store has no restriction on the variable `V`, i.e., it is free). In each iteration, when an answer A is found, the subsequent constraint store C_{i+1} is $C_i \wedge A_V \wedge \neg A_{\overline{V}}$, where A_V is the projection of A on `V` and $A_{\overline{V}}$ is the projection of A onto the rest of variables on `G`. This procedure is computationally expensive, as an answer/constraint store may contain multiple constraints whose negation/dual must be considered. Concretely, given a store containing n constraints, there are up to $2^n - 1$ distinct combinations of dual constraints that must be explored during the evaluation. E.g., given `{A>5,A<10,B>=0}`, its 7 duals are: `{A=<5,A<10,B>=0}`, `{A>5,A>=10,B>=0}`, `{A>5,A<10,B<0}`, `{A=<5,A=>10,B>=0}`, etc. The initial algorithm was based on the assumption that there was only a single constraint domain (the Herbrand domain) and that constraints between free variables were not allowed. Under these assumptions, the `forall/2` predicate over multiple variables was evaluated in a nested manner.[5] This algorithm was easily adapted to handle constraints over the rationals/reals, provided that these constraints did not involve relations between free variables at the time the forall was evaluated. For this reason, the constraints had to be placed at the end of the rule body, leaving only one variable free. The extended algorithm for constraints (`c_forall`) was implemented by collecting all the variables in a single set, transforming the previous nested call into a form such as `forall([A,B], goal(A,C,B))`. This implementation improves upon the initial algorithm by supporting constraints among free variables over the rationals/reals. E.g., rules of the form `p(a,X,Y) :- X < Y` are allowed, despite both `X` and `Y` being unbound at the time the constraint is invoked during the forall evaluation. However, to

[5] In fact, the s(CASP) compiler still preserves this representation, i.e., `forall(A,forall(B, goal(A,C,B)))`.

Table 1: Runtime (in ms.) for Total / Answers evaluation and Abstract Interpretation.

	prev_forall		c_forall		cls_forall		A.I.
	Total	Answers	Total	Answers	Total	Answers	
light-1	280	48 (1)	850	274 (1)	350	59 (1)	9
light-2	310	81 (1)	1940	383 (1)	770	86 (1)	10
light-3	200	- (0)	190	- (0)	470	- (0)	10
light-4	420	160 (1)	wrong		1060	196 (1)	11
tap-1	680	43 (1)	263350	223841 (1)	1000	42 (1)	18
tap-2	240	28 (1)	1270	492 (1)	590	26 (1)	18
tap-3	280	60 (2)	2370	1007 (2)	720	61 (2)	19
graph-0	15939	14322 (30)	1400	1162 (3)	1910	1210 (3)	4
graph-1	8950	8096 (30)	790	569 (3)	8950	8646 (30)	8
hanoi	250	55 (1)	240	53 (1)	360	52 (1)	1

NOTE: In parenthesis the number of Answers.

avoid the combinatorial explosion due to the dual constraint mentioned earlier, `c_forall` assumes that the free variables are restricted among them in a unique cluster, i.e., rules of the form `p(X,Y,Z,T) :- X < Y, Z < T.` are not supported. Under this assumption the dual of a constraint store can be "optimized" to only generate n distinct dual constraints stores, for instance `{A>5,A<10,B>=A}` has only three dual stores: `{A=<5,A<10,B>=A}`, `{A>5,A>=10,B>=A}`, and `{A>5,A<10,B<A}`.

Exploiting the Shared-Constraints Abstract Domain. The information inferred by the abstract domain Shared-Constraints, described in Section 4.2, generates clusters of variables that may be constrained at execution time. This allows the compiler to reorganize nested `forall` predicates at compile time by grouping universally quantified variables that belong to the same dependency class. Then, at execution time, each cluster is handled in a nested manner, following the algorithm strategy proposed in [16], while still supporting constraints among free variables within each cluster as implemented in [3]. The compile-time procedure starts by selecting each (possibly nested) `forall` predicate in the program. Then, the innermost goal is analyzed starting from an empty abstraction. The usage of an empty abstraction is sound because universally quantified variables are unconstrained when the `forall` is invoked. Consider the previous rule, `p(X,Y,Z,T) :- X < Y, Z < T.`, and the nested forall predicate `forall(X,forall(Y,forall(T,p(X,Y,Z,Y))))`. The analysis invokes the abstract interpreter on the inner goal: $\textsc{Analyze}(P, \langle p(X,Y,Z,T), \varnothing \rangle)$. The resulting success abstraction for `p(X,Y,Z,T)` groups variables that may become mutually constrained, e.g., `[[X,Y], [T,Z]]`. Based on this abstraction, nested universal quantifiers are automatically reorganized by grouping quantified variables that belong to the same dependency class, obtaining `forall([X,Y], forall([T], p(X,Y,T,Z)))`, preserving the declarative semantics of the program and improving its evaluation.

Validation. We validate the benefits of using the Abstract Interpreter with the Shared-Constraints domain in the execution of the `c_forall` predicate. As a baseline, we present the execution times of the `prev_forall` predicate, which is the fastest since it does not consider any kind of relationships among variables. We selected a suite of benchmarks: (i) the `light-`j and `tap-`k examples describe properties of a time-varying light and tap flows, both modeled in [2] under Event Calculus, featuring complex `forall` predicates; (ii) the remaining examples include two implementations of the graph-coloring problem, and the Tower of Hanoi. These last examples do not involve complex `forall` invocations or constraints and are used to evaluate the overhead introduced by the abstract interpretation approach. Table 1 shows eval-

```
%%% Graph Coloring %%%
node(1).       node(2).
node(3).       node(4).
node(5).
edge(1,2).   edge(1,3).
edge(2,4).   edge(2,5).
edge(3,4).   edge(3,5).
color(red).
color(green).
color(yellow).

other_color(X,C) :-
    color(C), color(C2),
    C \= C2, color(X,C2).
color(X,C) :-
    node(X), color(C),
    not other_color(X,C).

:- edge(X,Y), color(X,C),
                    color(Y,C).
```

```
%%% Specialization %%%
other_color(1,green) :-
     color(1,red).
other_color(1,green) :-
     color(1,yellow).
other_color(1,red) :-
     color(1,green).
other_color(1,red) :-
     color(1,yellow).
...
% Some rules are omitted
...
other_color(5,red) :-
     color(5,yellow).
other_color(5,yellow) :-
     color(5,green).
other_color(5,yellow) :-
     color(5,red).
color(1,green) :-
     not other_color(1,green).
color(1,red) :-
     not other_color(1,red).
color(1,yellow) :-
     not other_color(1,yellow).
color(2,green) :-
     not other_color(2,green).
...
color(4,green) :-
     not other_color(4,green).
color(4,red) :-
     not other_color(4,red).
color(4,yellow) :-
     not other_color(4,yellow).
color(5,green) :-
     not other_color(5,green).
color(5,red) :-
     not other_color(5,red).
color(5,yellow) :-
     not other_color(5,yellow).
```

Figure 5: Implementation of the Graph Coloring Problem and its specialization using *eterms*.

uation times for these examples, comparing the three `forall` implementations. For each example, we report the total execution time (Total, in ms), the time spent computing answers (Answers), the number of answers (in parenthesis), and, for the `cls_forall` the overhead due to the static analysis and the forall rewriting (AI). The results show that detecting variable independence under constraints can significantly improve performance. In particular, the `tap-1` example exhibits a roughly **250× speed-up**. The program contains 12 `forall` predicates, 11 of which quantify over at least two variables, while only 3 involve variables that may become mutually constrained at run time, while the remaining can be treated independently. In the cases of graph-0, grouping universal variables leads to a different number of answers. All of these answers are correct, although in some cases they are redundant. A similar effect is observed when the `prev_forall` predicate is used, which suggests that this treatment of universally quantified variables is more inclined to generating redundant answers. Moreover, we observe that the optimization often brings execution times close to those obtained with `prev_forall`. This highlights that, while in some cases grouping variables has little apparent effect, in others even a small reduction in the number of `forall` invocations can lead to substantial performance improvements. The additional compilation effort is negligible, remaining below 20 milliseconds in all benchmarks.

5.2 Toward Abstract Specialization

Since the approach in Section 4 adapts the PLAI framework to analyze goal-directed, top-down ASP programs and does not require the underlying abstract domain to introduce additional operations, it allows for the direct use of domains designed for (C)LP. However, custom domains and algorithms could yield more precise abstractions. We focus on the *eterms* abstract domain [23], which implements a type abstraction with efficient widening, providing a precise over-approximation of regular types. This captures all possible Herbrand equalities in any successful answer set, enabling partial grounding tailored to specific queries. Compared to query-independent analyses, the top-down, goal-directed approach

```
%%% path_disc/2 %%%
initial_node(a).

edge(a,b).
edge(a,c).
edge(b,d).
edge(c,d).

path_disc(X,Y) :-
  edge(X,Z),
  not path_disc(Z,Y).
```

```
%%% Analysis wo. entry point %%%
:- true pred edge(_A,_B)
  : ( term(_A), term(_B) )
  => ( rt11(_A), rt10(_B) ).
:- true pred path_disc(X,Y)
  : ( term(X), term(Y) )
  => ( rt11(X), term(Y) ).

path_disc(X,Y) :-
  true((term(X),term(Y),term(Z))),
  edge(X,Z),
  true((rt11(X),term(Y),rt10(Z))),
  not_path_disc(Z,Y),
  true((rt11(X),term(Y),rt10(Z))).

:- regtype rt11/1.
   rt11(a). rt11(b). rt11(c).
:- regtype rt10/1.
   rt10(b). rt10(c). rt10(d).
```

```
%%% Analysis with entry point %%%
% ?- initial_node(X),path_disc(X,Y)
:- true pred path_disc(X,Y)
  : ( rt19(X), term(Y) )
  => ( rt0(X), term(Y) ).

path_disc(X,Y) :-
  true((rt19(X),term(Y),term(Z))),
  edge(X,Z),
  true((rt0(X),term(Y),rt5(Z))),
  not path_disc(Z,Y),
  true((rt0(X),term(Y),rt5(Z))).

:- regtype rt0/1. rt0(a).
:- regtype rt19/1. rt19(a). rt19(d).
:- regtype rt5/1. rt5(b). rt5(c).
```

Figure 6: *eterms* analysis with/without entry point for the program `path_disc/2`.

produces more concrete constraints, since it takes into account only the possible constraints from an initial set of queries. The procedure for performing specialization using this abstract information consists of two steps: (i) **Analysis:** the program is analyzed using the *eterms* domain. At success, each variable's Herbrand domain is over-approximated across all possible answer sets. (ii) **Specialization:** using this information, the program is specialized, generating rule instances suitable for abstract evaluation.

Validation. Fig. 5 shows, on the left, an instance of the graph coloring problem, and on the right the resulting program after applying abstract specialization using the information inferred by *eterms*. For the specialized program, 150 answers to `?- color(X,Y)` were computed in 65 seconds (including 225 ms precompilation), with a mean answer time of 433 ms. The default program required 80 seconds, with a mean answer time of 533 ms. This represents an 18% reduction in execution time.

5.3 Detection of False Odd Loops over Negation

During compilation, s(CASP) generates denials for OLON rules, which are then checked by the predicate `nmr_check` to prevent inconsistencies. For instance, the rule `p(X) :- q(X,Y), not p(Y).` is treated as an OLON rule when the possible values of `X` and `Y` are unknown. However, if the program only includes `q(a,b)` for `q/2`, no denial is required. We use type information from the *eterms* abstract domain to filter candidate NMR rules in four steps: (i) the program is analyzed without NMR checks; (ii) for each OLON rule, the types of its initial and final literals are intersected: a non-empty intersection indicates a potential loop, while an empty one means it cannot occur; (iii) to ensure safety, all reachable literals matching the initial literal are also checked; (iv) if all intersections are empty, the OLON loop can be safely omitted from inconsistency checks.

Consider `path_disc/2` in Fig. 6 (left), which determines whether two nodes are disconnected by paths longer than one edge. Fig. 6 (center) shows the analysis performed without query information, i.e., all predicates are analyzed with top-most calling patterns. The true assertions inferred by the analysis over-approximate both calls and success patterns and record the abstract states at each program point. This is

not sufficient to filter the OLON rule: the calling pattern `term(X)` intersects with any inferred type, and the success goal for `not path_disc/2` leaves `Y` over-approximated as `term(Y)`. This loss of precision is caused by the `forall` in the dual of `path_disc/2`. In contrast, Fig. 6 (right) shows that analyzing from `?- initial_node(X), path_disc(X,Y).`, yields more precise results. The possible calls to `path_disc/2` are restricted to `a` and `d`. Although a call with `X=d` may arise during analysis, it cannot succeed because after calling `edge/2`, the type of `X` is refined to `a`, since `edge(d,_)` succeeds. Thus, the OLON loop can be omitted from inconsistency checking, since the regular types of `X` and `Z` are disjoint, i.e., `[a,d]` and `[b,c]`, respectively.

6 Related work

Top-down approaches for abstract interpretation were first used in analyzers such as MA3 and Ms [24], and matured in the PLAI analyzer [19, 20], which follows the tree abstraction of [4], but implemented with an efficient fixpoint algorithm. PLAI was later applied to the analysis of CLP/CHCs in [10] and imperative programs [8, 17]. In the context of ASP different approaches have been proposed for compilation, static analysis, and debugging. In [6], a static analysis methodology was proposed for analyzing an ASP program's cycle structure and deriving consistency guarantees without computing the answer sets directly. The approach identifies cycles and their interdependencies and constructs a Cycle Graph, where nodes represent cycles and edges represent connections (handles) between them. By examining this graph, one can determine whether a program admits stable models as well as reason about some properties. Similarly, in [14], ASP programs were represented as Cycle Graphs and partial answer sets obtained from the resulting structures. Denecker et al. [9] proposed the use of Approximation Fixpoint Theory to define the stable and well-founded semantics of logic programs, in particular in the context of ASP. They provided non-monotonic semantics by means of an approximator operator. While a theoretical framework, and centered around classical ASP, it may be interesting to explore if similar ideas could be useful in the context of s(CASP). It has also been proposed to abstract ASP rules to sound over-approximations of a program's answer sets by reducing the domain [21]. These abstractions enable approximate reasoning at lower cost and support unsatisfiability explanation by identifying relevant program parts. While closely related to Abstract Interpretation ideas, this work focuses on over-approximating answer sets for reasoning and debugging within the ASP context, rather than capturing more general program properties. Finally, in [22] a non-monotonic program analysis technique was presented based on logical abduction. The approach bounds the non-monotonic search using a monotone sequence of checkpoints, ensuring overall convergence. The technique is applied to the analysis of Horn clauses and similar ideas could be explored to increase precision in the handling of the negated parts of programs.

7 Conclusions

We presented a first step toward Abstract Interpretation of goal-directed ASP by adapting the PLAI top-down fixpoint framework. Using the new Shared-Constraints abstract domain, we captured variable dependencies from constraints and showed how compile-time information can improve the execution of s(CASP) programs, including more efficient `forall` evaluation. We also explored preliminary applications in abstract specialization and the elimination of spurious odd-loop checks. Our results are encouraging in that they show that, despite the current simplifications in handling the dual rules, abstract interpretation can provide reusable program insights and enable optimizations difficult to achieve syntactically. This suggests opportunities for precision improvements in future work.

References

[1] K. R. Apt (1990): *Introduction to Logic Programming*. In J. van Leeuwen, editor: *Handbook of Theoretical Computer Science*, Elsevier, pp. 493–576, doi:10.1016/B978-0-444-88074-1.50015-9.

[2] Joaquín Arias, Manuel Carro, Zhuo Chen & Gopal Gupta (2022): *Modeling and Reasoning in Event Calculus using Goal-Directed Constraint Answer Set Programming*. *TPLP* 22(1), pp. 51–80, doi:10.1017/S1471068421000156.

[3] Joaquín Arias, Manuel Carro, Elmer Salazar, Kyle Marple & Gopal Gupta (2018): *Constraint Answer Set Programming without Grounding*. *TPLP* 18(3-4), pp. 337–354, doi:10.1017/S1471068418000285.

[4] M. Bruynooghe (1991): *A Practical Framework for the Abstract Interpretation of Logic Programs*. *Journal of Logic Programming* 10, pp. 91–124, doi:10.1016/0743-1066(91)80001-T.

[5] Keith L. Clark (1978): *Negation as Failure*. In H. Gallaire & J. Minker, editors: *Logic and Data Bases*, Springer, pp. 293–322, doi:10.1007/978-1-4684-3384-5_11.

[6] Stefania Costantini (2005): *Towards Static Analysis of Answer Set Programs*. Available at https://api.semanticscholar.org/CorpusID:15643923.

[7] P. Cousot & R. Cousot (1977): *Abstract Interpretation: a Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints*. In: *Proc. of POPL'77*, ACM Press, pp. 238–252, doi:10.1145/512950.512973.

[8] Emanuele De Angelis, Fabio Fioravanti, John P. Gallagher, Manuel V. Hermenegildo, Alberto Pettorossi & Maurizio Proietti (2021): *Analysis and Transformation of Constrained Horn Clauses for Program Verification*. *TPLP* 22(6), pp. 1–69, doi:10.1017/S1471068421000211.

[9] Marc Denecker, Maurice Bruynooghe & Joost Vennekens (2012): *Approximation Fixpoint Theory and the Semantics of Logic and Answers Set Programs*. In: *Correct Reasoning*, pp. 178–194, doi:10.1007/978-3-642-30743-0_13.

[10] M. García de la Banda, M.V. Hermenegildo, M. Bruynooghe, V. Dumortier, G. Janssens & W. Simoens (1996): *Global Analysis of Constraint Logic Programs*. *ACM Transactions on Programming Languages and Systems* 18(5), pp. 564–615, doi:10.1145/232706.232734.

[11] Michael Gelfond & Vladimir Lifschitz (1988): *The Stable Model Semantics for Logic Programming*. In: *ICLP'88*, pp. 1070–1080, doi:10.2307/2275201.

[12] D. Jacobs & A. Langen (1989): *Accurate and Efficient Approximation of Variable Aliasing in Logic Programs*. In: *North American Conference on Logic Programming*.

[13] Joxan Jaffar & Jean-Louis Lassez (1987): *Constraint Logic Programming*. In: *ACM Symposium on Principles of Programming Languages*, ACM, pp. 111–119, doi:10.1145/41625.41635.

[14] Fang Li, Huaduo Wang, Kinjal Basu, Elmer Salazar & Gopal Gupta (2021): *DiscASP: A Graph-based ASP System for Finding Relevant Consistent Concepts with Applications to Conversational Socialbots*. In: *ICLP Technical Communications*, doi:10.4204/EPTCS.345.35.

[15] J.W. Lloyd (1987): *Foundations of Logic Programming*. Springer, second, extended edition, doi:10.1007/978-3-642-83189-8.

[16] Kyle Marple, Elmer Salazar, Zhuo Chen & Gopal Gupta (2017): *The s(ASP) Predicate Answer Set Programming System*. *The Association for Logic Programming Newsletter*.

[17] M. Méndez-Lojo, J. Navas & M.V. Hermenegildo (2007): *A Flexible (C)LP-Based Approach to the Analysis of Object-Oriented Programs*. In: *LOPSTR*, *LNCS* 4915, Springer-Verlag, pp. 154–168, doi:10.1007/978-3-540-78769-3_11.

[18] K. Muthukumar & M.V. Hermenegildo (1989): *Determination of Variable Dependence Information at Compile-Time Through Abstract Interpretation*. In: *1989 North American Conference on Logic Programming*, MIT Press, pp. 166–189.

[19] K. Muthukumar & M.V. Hermenegildo (1990): *Deriving A Fixpoint Computation Algorithm for Top-down Abstract Interpretation of Logic Programs*. Technical Report ACT-DC-153-90, Microelectronics and Comp. Tech. Corp. (MCC). Available at https://cliplab.org/papers/mcctr-fixpt.pdf.

[20] K. Muthukumar & M.V. Hermenegildo (1992): *Compile-time Derivation of Variable Dependency Using Abstract Interpretation*. *Journal of Logic Programming* 13(2/3), pp. 315–347, doi:10.1016/0743-1066(92)90035-2.

[21] Zeynep G. Saribatur & Thomas Eiter (2021): *Omission-Based Abstraction for Answer Set Programs*. *Theory and Practice of Logic Programming* 21(2), p. 145–195, doi:10.1017/S1471068420000095.

[22] Daniel Schwartz-Narbonne, Philipp Rümmer, Martin Schäf, Ashish Tiwari & Thomas Wies (2015): *Non-monotonic program analysis*. In: *2nd HCVS@ETAPS*.

[23] C. Vaucheret & F. Bueno (2002): *More Precise yet Efficient Type Inference for Logic Programs*. In: *9th SAS*, *LNCS* 2477, pp. 102–116.

[24] R. Warren, M.V. Hermenegildo & S. K. Debray (1988): *On the Practicality of Global Flow Analysis of Logic Programs*. In: *JICSLP*, MIT Press, pp. 684–699.