Overcoming the Grounding Bottleneck Due to Constraints in ASP Solving: Constraints Become Propagators

Bernardo Cuteri¹, Carmine Dodaro¹, Francesco Ricca¹,⋆ and Peter Schüller²
¹University of Calabria, Italy
²TU Wien, Austria
{cuteri, dodaro, ricca}@mat.unical.it, ps@kr.tuwien.ac.at

Abstract
Answer Set Programming (ASP) is a well-known formalism for Knowledge Representation and Rea-
soning, successfully employed to solve many AI problems, also thanks to the availability of efficient
implementations. Traditionally, ASP systems are based on the ground&solve approach, where the
grounding transforms a general input program into its propositional counterpart, whose stable models
are then computed by the solver using the CDCL algorithm. This approach suffers an intrinsic lim-
itation: the grounding of one or few constraints may be unaffordable from a computational point of
view; a problem known as grounding bottleneck. In this paper, we develop an innovative approach for
evaluating ASP programs, where some of the constraints of the input program are not grounded
but automatically translated into propagators of the
CDCL algorithm that work on partial interpreta-
tions. We implemented the new approach on top of
the solver WASP and carried out an experimen-
tal analysis on different benchmarks. Results show
that our approach consistently outperforms state-
of-the-art ASP systems by overcoming the ground-
ing bottleneck.

1 Introduction
Answer Set Programming (ASP) [Brewka et al., 2011] is a
decorative formalism for knowledge representation and rea-
soning based on the stable model semantics [Gelfond and
Lifschitz, 1991]. Efficient implementations of ASP, such as
CLINGO [Gebser et al., 2016] and DLV [Alviano et al., 2017],
are available, which made possible the development of con-
crete applications. In the recent years, ASP has been widely
used for solving several problems in the context of artificial
intelligence, such as game theory [Amendola et al., 2016],
natural language processing [Schüller, 2016], natural lan-
guage understanding [Cuteri et al., 2019b], robotics [Erdem
and Patoglu, 2018], scheduling [Dodaro and Maratea, 2017],
and more [Erdem et al., 2016]. Therefore, the improve-
ment of ASP systems is an interesting research topic in arti-
ficial intelligence. Traditional ASP systems are based on the
ground&solve approach [Kaufmann et al., 2016], in which a
grounder module transforms the input program (containing
variables) in its propositional counterpart, whose stable mod-
els are subsequently computed by the solver module. ASP
solvers implement an extension the Conflict Driven Clause
Learning (CDCL) algorithm [Kaufmann et al., 2016].
Although the ASP implementations based on ground&solve
are known to be effective in many contexts [Erdem et al., 2016],
the traditional approach has an intrinsic limitation. In particu-
lar, there are classes of programs whose evaluation is not fea-
sible because of the combinatorial blowup of the grounding
of some rules. This issue is usually referred to as grounding
bottleneck. In many practical cases the grounding bottleneck
is due to one or few constraints that model the (non) admis-
sibility of problem solutions [Ostrowski and Schaub, 2012;
Calimeri et al., 2016].

In the literature, there are several attempts to solve the
grounding bottleneck problem [Gebser et al., 2018]. Some of
these are based on language extensions that hybridize ASP
with other formalisms (such as constraint programming [Ost-
rowski and Schaub, 2012; Balducci and Lierler, 2017], and
difference logic [Gebser et al., 2016; Susman and Lierler,
2016]) that can be used to express the hard-to-ground con-
straints. Hybrid formalism are efficiently evaluated by cou-
pling an ASP systems with a solver for the other theory, thus
circumventing the grounding bottleneck. There are also ap-
proaches that work on plain ASP, such as lazy grounding tech-
niques, that resulted in several promising systems, such as
GASP [Dal Palù et al., 2009], ASPERIX [Lefèvre and Nico-
las, 2009] and ALPHA [Weinzierl, 2017]. The idea of lazy
grounding is to instantiate a rule only when its body is satis-
fied. In this way, it is possible to prevent the grounding
of rules which are unnecessary during the search of an an-
swer set. Albeit lazy grounding techniques obtained good
preliminary results, their performance is still not competitive
with state-of-the-art systems [Gebser et al., 2018]. A differ-
cent approach was proposed in [Cuteri et al., 2017], where
problematic constraints are removed from the non-ground
input program and the resulting program is provided as input
to a modified version of a CDCL-based able to simulate the
presence of problematic constraints. In [Cuteri et al., 2017] the
authors compared two alternative strategies for extending
ASP solvers based on CDCL, namely lazy instantiation

⋆Contact Author
and propagators. In the lazy instantiation strategy, the solver computes a stable model of the program without problematic constraints. If this stable model satisfies also the omitted constraints, then it is also a stable model of the original program. Otherwise, the violated instances of these constraints are lazily instantiated, and the search continues. The other strategy relies on an extension of the propagation function by adding custom propagators, whose role is to perform the inferences of missing constraints during the search. However, both lazy instantiation and propagators were based on procedures written in an imperative language and that are specific for the problem at hand. This approach loosens the declarative nature of ASP, and is a time consuming task that can be carried out only by developers expert on system APIs. Recently, Cuteri et al. [Cuteri et al., 2019a] presented a strategy to translate (compile) non-ground constraints into dedicated C++ procedures, which is used by the system to lazily instantiate them in an automatic way. This approach keeps declarativity of ASP and is effective when the problematic constraints are likely to be satisfied by a candidate model (i.e., whenever lazy instantiation is effective cfr. [Cuteri et al., 2017]). However, a significant number of problems, especially hard combinatorial problems from ASP competitions [Calimeri et al., 2016] cannot be handled efficiently by systems relying on lazy instantiation [Cuteri et al., 2017].

In this paper, we push forward the idea of [Cuteri et al., 2017; Cuteri et al., 2019a], and we present a novel strategy for translating (compiling) non-ground constraints into dedicated C++ procedures that are used as propagators during the search of the CDCL algorithm. Differently from [Cuteri et al., 2019a], propagators operate on partial interpretations and require radically different algorithms that are more involved than methods on total interpretations. To assess the performance of our approach, we implemented it on top of WASP [ Alviano et al., 2015] and conducted an experimental analysis on different benchmarks proposed in the literature. Results show that our approach outperforms state-of-the-art ASP systems in all tested scenarios.

2 Preliminaries

2.1 Answer Set Programming

An ASP program $\pi$ is a finite set of rules of the form $h_1 \ldots h_n \leftarrow b_1, \ldots, b_m$, where $n, m \geq 0$, $n + m \neq 0$, $h_1, \ldots, h_n$ are atoms and represent the head of the rule, while $b_1, \ldots, b_m$ are literals and represent the body of the rule. We denote by $\text{body}(r)$ the set of literals appearing in the body of $r$. In particular, an atom is an expression of the form $p(t_1, \ldots, t_k)$, where $p$ is a predicate of arity $k$ and $t_1, \ldots, t_k$ are terms. Terms are alphanumeric strings and are either variables or constants. According to Prolog conventions, only variables start with uppercase letters.

A literal is an atom $a$ or its negation $\neg a$, where $\neg$ denotes the negation as failure. A literal is said to be positive if it is an atom and negative if it is the negation of an atom. For an atom $a$, the complement is $\neg a \leftarrow \neg a$, for a negated atom $\neg a$, the complement is $\neg \neg a = a$. For a literal $l$, $\text{trm}(l)$ denotes the list of terms in $l$, and $\text{pred}(l)$ is the name of the predicate of $l$. A rule is called a constraint if $n = 0$, and a fact if $n = 1$ and $m = 0$.

### Algorithm 1 ComputeStableModel

Input: A ground program $\mathcal{P}$
Output: A stable model for $\mathcal{P}$ or $\bot$

1. begin
2. $I := \emptyset$;
3. $I := \text{Propagate}(I)$;
4. if $I$ is inconsistent then
5. $r := \text{CreateConstraint}(I)$;
6. $I := \text{RestoreConsistency}(I)$;
7. if $I$ is consistent then $\mathcal{P} := \mathcal{P} \cup \{r\}$;
8. else return $\bot$;
9. else if $I$ total then return $I$;
10. else
11. $I := \text{RestartIfNeeded}(I)$;
12. $\mathcal{P} := \text{DeleteConstraintsIfNeeded}(\mathcal{P})$;
13. $I := I \cup \text{ChooseLiteral}(I)$;
14. goto 3;

### Function Propagate($I$)

1. $\mathcal{I} = I$;
2. for $\ell \in \mathcal{I}$ do $\mathcal{I} := \mathcal{I} \cup \text{Propagation}(\mathcal{I}, \ell)$;
3. return $\mathcal{I}$;

An object (atom, rule, etc.) is called ground or propositional, if it contains no variables. Rules are safe, that is each variable occurs in a positive literal of the body. Given a program $\pi$, let the Herbrand Universe $U_\pi$ be the set of all constants appearing in $\pi$ and the Herbrand Base $B_\pi$ be the set of all possible ground atoms which can be constructed from the predicate symbols appearing in $\pi$ with the constants of $U_\pi$. $B$ denotes $B_\pi \cup B_\pi$. Given a rule $r$, $\text{gnd}(r)$ denotes the set of rules obtained by applying all possible substitutions $\sigma$ from the variables in $r$ to elements of $U_\pi$. For a program $\pi$, the ground instantiation $\text{gnd}(\pi)$ of $\pi$ is the set $\bigcup_{r \in \pi} \text{gnd}(r)$. Stable models of a program $\pi$ are defined using its ground instantiation $\text{gnd}(\pi)$. An interpretation $I$ for $\pi$ is a set of literals. $I$ is total if $\forall a \in B_\pi$, either $a \in I$ or $\neg a \in I$ and $l \in I \implies \exists \ell \in I$. Given an interpretation $I$, $I^+$ denotes the set of positive literals in $I$ and $I^-$ denotes the set of negative literals in $I$. A ground literal $l$ is true w.r.t. $I$ if $l \in I$, otherwise it is false. A total interpretation $I$ is a model for $\pi$ if, for every $r \in \text{gnd}(\pi)$, at least one atom in the head of $r$ is true w.r.t. $I$ whenever all literals in the body of $r$ are true w.r.t. $I$. The reduct of a ground program $\pi$ w.r.t. a model $I$ is the ground program $\pi^I$, obtained from $\pi$ by (i) deleting all rules $r \in \pi$ whose negative body is false w.r.t. $I$ and (ii) deleting the negative body from the remaining rules. An interpretation $I$ is a stable model of a program $\pi$ if $I$ is a model of $\pi$, and there is no $J$ such that $I$ is a model of $\pi^I$ and $J^+ \subset I^+$. A program $\pi$ is coherent if it admits at least one stable model, incoherent otherwise.

2.2 Classical CDCL Evaluation

The standard solving approach for ASP is instantiation followed by a procedure similar to CDCL for SAT with exten-
sions specific to ASP [Kaufmann et al., 2016]. The basic algorithm ComputeStableModel(II) for finding a stable model of program II is shown in Algorithm 2. The Function 1 combines unit propagation (as in SAT) with some additional ASP-specific propagations, which ensures the model is stable (cf. [Kaufmann et al., 2016]). The algorithm calls for each additional propagator a procedure called Propagation in Function 1, which takes as input a true literal, and the interpretation, and returns as output the extended interpretation with literals that are inferred and their reason. Given a partial interpretation I consisting of literals, and a set of rules II, unit propagation infers a literal ℓ to be true if there is a rule r ∈ II such that r can be satisfied only by I ∪ {ℓ}. Given the nogood representation C(r) = \{a_1, ..., a_n, b_1, ..., b_j, ..., b_{m+1}, ..., b_m\} of a rule r, then the negation of a literal ℓ ∈ C(r) is unit propagated w.r.t. r and rule r iff C(r) \ {ℓ} ⊆ I. In the following we refer to C(r) \ {ℓ} as the reason for the inference of ℓ.

Pairs (ℓ, reason of ℓ) are stored during the execution of propagation, and will be used to perform conflict resolution, and more specifically during the UIP computation [Alviano et al., 2015]. We refer to the list of such pairs as the implication list. To ensure that models are supported, unit propagation is performed on the Clark completion of II or alternatively a support propagator is used [Alviano and Dodaro, 2016].

Example 1 Consider the following ground program Π1

\[
\begin{align*}
g_1 &: a \leftarrow \neg b \\
g_2 &: b \leftarrow \neg a \\
g_3 &: c \leftarrow \neg d \\
g_4 &: d \leftarrow \neg c
\end{align*}
\]

ComputeStableModel(Π1) starts with I = ∅ and does not propagate anything in line 3. I is partial and consistent, so the algorithm continues in line 11. Assume no restart and no deletion is performed, and assume ChooseLiteral returns \{a\}, i.e., I = \{a\}. Next, Propagate(I) is called which yields I = \{a, b, \neg b\} where \neg b comes from unit propagation on g_2 and b from unit propagation on g_6. I is inconsistent and an analysis yields the reason of the conflict, i.e., CreateConstraint(I) = \{g_7\} with g_7 := \neg a. Intuitively, the truth of a leads to an inconsistent interpretation, thus a must be false. Then, the consistency of I is restored (line 6), i.e., I = \{a\}, and g_3 is added to Π1. The algorithm again restarts at line 3 with I = \{a\} and propagates I = \{\neg a, b\}, where \neg a comes from unit propagation on g_7, and b from unit propagation on g_2. I is partial and consistent, therefore lines 11–13 are executed. Assume again that no restart and no constraint deletion happens, and that ChooseLiteral(I) = \{e\}. Therefore, the algorithm continues in line 3 with I = \{\neg a, b, c\}. Propagation yields I = \{\neg a, b, c, \neg d\} because \neg d is support-propagated w.r.t. g_3 and I (or unit-propagated w.r.t. the completion of g_3 and I). I is total and consistent, therefore the algorithm returns I as the first stable model.

3 Constraints as Propagators

In this section we present our strategy for evaluating constraints with propagators that are automatically generated by a compilation-based approach. In the following, we assume w.l.o.g. that the bodies of constraints never contain two literals with the same predicate name. Algorithms 3-4 present the pseudo-code of the compiler generating propagators from constraints. To ease readability, we write in red the code that is produced by the compiler, and in black the code (e.g., variables and references) that are in the scope of the compiler. Thus, red lines, written inside the symbols « and », represent the code that is printed by the compiler, and black references in gray lines denote the fact that the compiler is printing the value of such variables. For example, «case "pred(C)"» is equivalent to the C++ instruction printf("case \"%s\"", pred(C)). Algorithm 3 takes as input a non-ground ASP constraint C and prints the code of the propagator for C. It starts by declaring in the propagator code an empty implication list (line 2) which will be in charge of accumulating the result of the propagation of a literal l (the literal in input to the propagator), which we call starting literal. Depending on the predicate name of l, the propagator code must evaluate one of the \{body(C)\} possible join orders. To do so, it writes a switch on the predicate name of l, writing one case for each literal in the constraint. In each case, it prints a dedicated code that is able to propagate l calling algorithm 4. The evaluation is written as a nested join loop, which is a cascade of for and if blocks.

We now describe what is happening in algorithm 4, which receives a constraint C and a literal c ∈ C. First it builds a substitution σ that will be used and updated in the whole nested join. Initially, σ is set to the empty substitution ε (line 1). Then, from line 2 to 4 the compiler writes the code that adds to σ a mapping from the variables in c (known at compilation time, thus black in the algorithm) to the constants in l (ground literal known at execution time). Recall that trm(x) returns the list of terms of a literal x. The square notation, commonly used in C++ programming, denotes the access of a list to a specific (one-based) index (e.g. if the list is trm(c) = [X, Y, 2], then trm(c)[1] is X). At line 5, the algorithm reorders the body of C in a new list B where negative literals are always at the end of B; and c is not in B.

In the propagator code, u will be the complement of the literal that is unit propagated, and it is initialized to ⊥ to denote that it is not known at the beginning. The printing of the code of the nested join loop starts at line 8. Essentially, at each iteration i we either print a for loop if B[i] is positive, and an if statement if it is negative. Each subsequent for or if is nested inside the previous one. When B[i] is positive,
the propagator will first collect in the set \(T\) the true literals that match (read it as has a substitution to) \(\sigma(B[i])\), i.e., the current body literal \(B\), to which the building substitution \(\sigma\) is applied (line 10). Such true literals are iterating the current substitution, and in case of propagation, will build the reason of the propagation. Moreover, if the undefined literal is still \(\bot\) (line 12) it means the join must also collect the undefined literals that match \(\sigma(B[i])\), line 11. At line 14 the propagator iterates with a variable \(b\) the union of the true literals and eventually undefined literals just collected. When \(b\) is undefined, \(u\) becomes \(b\) (lines 15 - 16). At this point, \(\sigma\) is extended with the variables of the current body literal \(B[i]\), by mapping them to the constants in \(b\) (line 17), similarly to what was done before with \(c\) and \(l\). On the other hand, when \(B[i]\) is negative (line 20) the compiler iterates over a negative literal of the constraint. In such case, all variables have a mapping in \(\sigma\) (because of safety condition of ASP and negative literals are at the end of \(B\)). The propagator will determine the ground literal \(b_i = \sigma(B[i])\). At this point, the join can continue either if \(b_i\) is true, or the undefined \(u\) is still \(\bot\) and \(b_i\) is undefined (line 22). In the second case, \(u\) will be set to be equal to \(b_i\). Once the compiler completes the writing of the cascade of nested for and if blocks and it is in the most nested block, it can write the code that collects the successful match of the constraint as an additional pair in the implication list of the propagator (lines 25-29). The literal \(l\) is always part of the reason \(R\) (line 25), and the propagator will also add to \(R\) all the true literals in the join (all \(b_i\), beside \(b_i = u\)). The complement of \(u\) which is a propagated literal, together with its reason \(R\) are added to the implication list (29). Finally, from line 30 to line 34, the compiler writes the code that rolls back sigma to its previous state (at the end of each block), and eventually sets \(u\) back to \(\bot\) in case it is equal to \(b_i\) and closes the parenthesis of the nested for and if blocks. In rough terms, the compiler produces that code of a procedure that is able to find an instantiation of the constraint with a single undefined literal to be unit propagated.

3.1 Compilation Example

To better understand what the compiler actually prints, in algorithm 5 we provide an example of a generated propagator for a small constraint. The compiled constraint \(C\) in the example is \(b(X), c(X, Y), \sim d(X, 1)\). The input of the propagator is an interpretation \(I\) and a starting literal \(l\), and the output is the implication list containing literals to be propagated with their reasons.

First the propagator switches on the predicate name of the starting literal \(l\) (line 2). There is a case for each predicate name appearing in \(C\), i.e., “b”, “c” and “d”. For the sake of readability we present the single case for predicate name “b” (lines 4-36). The propagator creates the substitution \(\sigma\), where the variable \(X\) is mapped to the first (and only) constant of \(l\) (lines 4 and 5). Then the literal \(u\) is initialized to \(\bot\) and \(\sigma\) is stored in \(\sigma_i\) so that it can be rolled back at each iteration of the next loop. Subsequently, the propagator collects all the true atoms of \(c(X, Y)\), where \(X\) is first replaced by the constant in \(l\) by the application of \(\sigma\), as well as all the undefined atoms of \(c(X, Y)\) analogously. Then, the propagator iterates over true and undefined atoms at line 12, continuing the join.

**Algorithm 3 CompilePropagateConstraintWithStarter**

**Input**: A constraint \(C\), a literal \(c \in C\)

**Output**: Prints an algorithm that is able to perform the unit propagation of \(C\) starting from a ground literal whose predicate is the same of \(c\)

```plaintext
1 «\(\sigma = c\)»
2 forall the \(k = 1, \ldots, \text{trm}(c)\) do
3 if \(\text{trm}(c)[k]\) is variable then
4 «\(\sigma = \sigma \cup \{\text{trm}(c)[k] \mapsto \text{trm}(l)[k]\}»
5 \(B = \text{computeBodyOrdering}(C, c)\)
6 «\(u := \bot\)»
7 forall the \(i = 1, \ldots, |B|\) do
8 «\(\sigma_i = \sigma\)»
9 if \(B[i]\) is positive then
10 «\(T_i = \{t \in I^+ \mid \text{match}(\sigma(B[i]), t)\}»
11 «\(U_i = \emptyset\)»
12 «if \(u = \bot\)»
13 «\(U_i = \{p \in (B \setminus I)^+ \mid \text{match}(\sigma(B[i]), p)\}»
14 «for \(b_i \in (T_i \cup U_i)\) »
15 «if \(b_i \in U_i\)»
16 «\(u = b_i\)»
17 forall the \(k = 1, \ldots, |\text{trm}(B[i])|\) do
18 if \(\text{trm}(B[i])\) is variable then
19 «\(\sigma = \sigma \cup \{\text{trm}(B[i])[k] \mapsto \text{trm}(b_i)[k]\}»
20 else
21 «\(b_i = \sigma(B[i])\)»
22 «if \(b_i \in I \lor (u = \bot \land b_i \in (B \setminus I))\) »
23 «if \(u = \bot \land b_i \in (B \setminus I)\) »
24 «\(u = b_i\)»
25 «\(R = \{l\}»
26 forall the \(i = 1, \ldots, |B|\) do
27 «\(R = R \cup \{b_i\}\)»
28 «\(R = R \setminus \{u\}\)»
29 «\(I_i = I_i \cup (\neg, R)\)»
30 forall the \(i = |B|, \ldots, 1\) do
31 «\(\sigma = \sigma_i\)»
32 «if \(u = b_i\)»
33 «\(u = \bot\)»
34 «\}\)»
```

For each \(b_i\) in \(T_1 \cup U_1\), the propagator checks whether \(b_i\) is undefined and eventually updates \(u\), if no undefined has been found up to now. Then \(\sigma\) is extended by the mappings of \(X\) and \(Y\) respectively to the first and second constants in \(b_i\) (lines 16-17). At this point, in line 18, the propagator handles the negative literal \(\neg d(X, 1)\), which becomes a ground literal \(b_2\) after the substitution with \(\sigma\); and the propagator checks whether \(b_2\) is the first undefined (and updates \(u\)), or \(b_2\) is true and a complete substitution is found. In these cases the propagator execution would reach the innermost code (from line 24) where it adds a propagation to the output implication list. In particular, if \(u = b_1\), \(b_1\) is the propagated literal and the reason is \(R \cup \{b_2\}\), otherwise \((u = b_2)\), \(b_2\) is the propagated literal and the reason is \(R \cup \{b_1\}\). At lines 30 to 32 and 33 to
36, the propagator restores the state of $\sigma$ and eventually the undefined literal to $\bot$ and closes a nested code block.

### 3.2 Implementation

The implementation follows the execution presented in pseudo-code in algorithms 3 and 4. The compiler has been implemented in C++, and its output is also C++ code compilable to the WASP propagator interface, and is loaded in the ASP solver as a C++ dynamic library. Several optimizations have been implemented, some of them discussed in the following.

Nested loops are made efficient by using proper indexes on terms, for which we use hash-maps to access matching ground literals. This is possible since the order of evaluation is fixed and the predicates extensions can be indexed statically (index terms are known at compilation time). For example, if we evaluate :- a(X,Y), b(Y,Z) with a as starter, we can benefit from indexing the extension of the predicate b on the first term (variable Y). By using a predicate-wise split of the interpretation and indexes, we highly optimize the propagator code related to interpretation access (e.g. lines 10, 13 and 22 of algorithm 4). Moreover, even though, in the compiler pseudo-code, we assumed that there is no repetition of predicates names, we explicitly handle such cases in the implementation: basically, we detect when the same undefined literal is used more than once in the same nested join. The implementation also supports built-in arithmetic comparisons, and features a heuristic to employ a smart body join. The implementation also supports built-in arithmetic comparisons, and features a heuristic to employ a smart body join. The implementation follows the execution presented in algorithm 4.

### 4 Experiments

In order to empirically assess the impact of the proposed technique, we considered several benchmarks used in previous editions of the ASP Competitions [Calimeri et al., 2016] or proposed in the literature, namely Incremental Scheduling, Natural Language Processing (NLP) using three different objective functions (cardinality, coherency and weighted abduction), Packing, and Partner Units. All benchmarks contain at least one constraint whose grounding is expensive from a computational point of view. We used as reference for the state-of-the-art the traditional ASP system CLINGO [Gebser et al., 2016] and the lazy-grounding based ASP system AL-PHA [Weinzierl, 2017]. It is important to observe that some of the encodings include features that are not currently supported by ALPHA, e.g., weak constraints. For such benchmarks, the performance of ALPHA is not reported. Moreover, we included in the comparison the solver WASP [Alviano et al., 2015] and the version of WASP implementing the lazy propagator as described in [Cuteri et al., 2019a] referred to as WASP-LAZY. In the following, our implementation is referred to as WASP-EAGER. All versions of WASP use CLINGO as grounder (executed with the option -output=models). WASP-LAZY and WASP-EAGER compile as lazy and eager propagators, respectively, all constraints that do not contain aggregates in the encodings, and therefore such constraints are not grounded in advance. Note that compilation is done only once for each benchmark and its running time is negligible (less than 2 seconds). The experiments were run on an Intel Xeon CPU E7-8880 v4 @ 2.20GHz, time and memory were limited to 2 hours and 5 GB, respectively.

An overview of the obtained results is given in Table 1: we report the number of solved instances, N/A indicates that the

<table>
<thead>
<tr>
<th>Algorithm 4 Ex. of compiling :- b(X), c(X,Y), \sim (X,1)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Input</strong>: An interpretation $I$ and a literal $l$ to propagate</td>
</tr>
<tr>
<td><strong>Output</strong>: An implication list</td>
</tr>
<tr>
<td>1 $I_I = \emptyset$</td>
</tr>
<tr>
<td>2 switch $pred(l)$ do</td>
</tr>
<tr>
<td>3 case “b”</td>
</tr>
<tr>
<td>4 $\sigma = \epsilon$</td>
</tr>
<tr>
<td>5 $\sigma = \sigma \cup {X \mapsto trn(l)[1]}$</td>
</tr>
<tr>
<td>6 $u = \bot$</td>
</tr>
<tr>
<td>7 $\sigma_1 = \sigma$</td>
</tr>
<tr>
<td>8 $T_2 = {t \in I^+ \mid match(\sigma(c(X,Y),t))}$</td>
</tr>
<tr>
<td>9 $U_2 = \emptyset$</td>
</tr>
<tr>
<td>10 if $u = \bot$ then</td>
</tr>
<tr>
<td>11 $U_1 = {p \in (B \setminus I)^+ \mid match(\sigma(c(X,Y),p))}$</td>
</tr>
<tr>
<td>12 forall the $b_1 \in T_2 \cup U_1$ do</td>
</tr>
<tr>
<td>13 if $b_1 \in U_1$ then</td>
</tr>
<tr>
<td>14 $U_2 = {l}$</td>
</tr>
<tr>
<td>15 $R = R \cup {b_1}$</td>
</tr>
<tr>
<td>16 $R = R \cup {b_2}$</td>
</tr>
<tr>
<td>17 $R = R \setminus {u}$</td>
</tr>
<tr>
<td>18 $I_I = I_I \cup {T, R}$</td>
</tr>
<tr>
<td>19 $\sigma = \sigma_2$</td>
</tr>
<tr>
<td>20 if $u = b_2$ then</td>
</tr>
<tr>
<td>21 $u = \bot$</td>
</tr>
<tr>
<td>22 case “c”</td>
</tr>
</tbody>
</table>
| 23 .............................................................................. \
| 24 break |
| 25 case “d” |
| 26 .............................................................................. \
| 27 break |
| 28 return $I_I$ |
In this paper, we presented a novel approach for the automatic compilation of constraints into propagators and we implemented it on top of the ASP solver WASP. The performance of our tool has been empirically validated on benchmarks whose evaluation was not feasible due to the combinatorial blow-up of the grounding of some constraints. Results show that our solution outperforms state-of-the-art systems on benchmarks modeling AI problems. Concerning future work, we plan to extend our implementation for supporting also other ASP constructs, such as aggregates.

Acknowledgments
This work was partially supported by MISE under PON MAP4ID Prog. n. F/190138/01-03/X44.
References


