TPLP **19** (5–6): 857–873, 2019. © Cambridge University Press 2019
doi:10.1017/S1471068419000231

Partial Compilation of ASP Programs

BERNARDO CUTERI, CARMINE DODARO, FRANCESCO RICCA
DeMaCS, University of Calabria, Italy,
(*e-mails:* {cuteri,dodaro,ricca}@mat.unical.it)

PETER SCHÜLLER
Knowledge-based Systems Group, TU Wien, Austria,
(*e-mail:* peter.schueller@tuwien.ac.at)

submitted 26 July 2019; accepted 31 July 2019

Abstract

Answer Set Programming (ASP) is a well-known declarative formalism in logic programming. Efficient implementations made it possible to apply ASP in many scenarios, ranging from deductive databases applications to the solution of hard combinatorial problems. State-of-the-art ASP systems are based on the traditional ground&solve approach and are general-purpose implementations, i.e., they are essentially built once for any kind of input program. In this paper, we propose an extended architecture for ASP systems, in which parts of the input program are compiled into an ad-hoc evaluation algorithm (i.e., we obtain a specific binary for a given program), and might not be subject to the grounding step. To this end, we identify a condition that allows the compilation of a sub-program, and present the related partial compilation technique. Importantly, we have implemented the new approach on top of a well-known ASP solver and conducted an experimental analysis on publicly-available benchmarks. Results show that our compilation-based approach improves on the state of the art in various scenarios, including cases in which the input program is stratified or the grounding blow-up makes the evaluation unpractical with traditional ASP systems.

KEYWORDS: Answer set programming, Grounding bottleneck, Compilation

1 Introduction

Answer Set Programming (ASP) is a powerful formalism that has roots in Knowledge Representation and Reasoning and is based on the stable model semantics (Gelfond and Lifschitz 1991; Brewka et al. 2011). ASP is a viable solution for representing and solving many classes of problems thanks to its high expressive power and the availability of efficient systems (Gebser et al. 2018). Indeed, ASP has been successfully applied to several academic and industrial applications, such as data integration and consistent query answering in databases (Marileo and Bertossi 2010; Manna et al. 2015), ontological reasoning (Ianni et al. 2009; Leone et al. 2019), explanation of biomedical queries (Erdem and Öztok 2015), game theory (Amendola et al. 2016), product configuration (Kojo et al. 2003; Dodaro et al. 2016), decision support systems for space shuttle flight controllers (Nogueira et al. 2001), construction of phylogenetic supertrees (Koponen et al. 2015), reconfiguration systems (Aschinger et al. 2011), natural language understanding (Cuteri et al. 2019), and more (Erdem et al. 2016). A key feature of ASP consists of the capability to model hard combinatorial problems in a declarative and compact way. Albeit ASP is supported by efficient systems, the improvement of their performance is still an interesting research topic (Lierler et al. 2016).

The state-of-the-art approach for solving ASP programs has two steps: initially, variables are replaced with constants by the grounder, and the resulting equivalent variable-free program is evaluated by a propositional search-based solver computing the answer sets. This approach is usually referred to as the ground&solve approach (Gebser et al. 2018). Moreover, ASP implementations are general-purpose, i.e., they are essentially built once for any kind of input program.

In this paper, we propose an extended architecture for ASP systems, which allows for obtaining specific implementations for a given program and relaxes the traditional two-steps architecture by avoiding that the whole program has to be grounded upfront.

Specific implementations are obtained by introducing a technique that allows for *compiling* (parts of) ASP programs into dedicated implementations. As usual in computer science, by compilation we mean the translation of a program written in a high-level language into another programming language (usually a lower level language nearer to the machine code) to create an executable program. To this end, we identified a condition that allows the compilation of a non-ground ASP sub-program into a C++ procedure, which simulates the behavior of that subprogram during the evaluation. Since, in general, only parts of the input program are transformed into dedicated implementations, we name our technique *partial compilation of ASP programs*. To the best of our knowledge, *this is the first attempt of compiling ASP programs in the literature*.

Whenever an entire program can be compiled an ad-hoc specialized binary is generated (this is the case for the relevant fragment of stratified normal programs); otherwise a compilable subprogram P is packaged into a dynamic library that extends an existing ASP solver with an ad-hoc lazy propagator (Cuteri et al. 2017) that simulates the behavior of P during the computation of answer sets. Note that, as it will be clearer later, compiled sub-programs are never grounded. One of the weak spots of the pure ground&solve approach is that the grounding might generate a propositional program that is too big for solvers to tackle (this problem is often referred to as the grounding bottleneck) of ASP; our architecture alleviates this problem whenever the rules that are causing the bottleneck are compiled.

An important feature of our partial compilation approach is that it can be implemented by extending in a natural way existing ASP systems that support external propagators (Gebser et al. 2016; Dodaro and Ricca 2018). This allows for keeping the benefits of existing implementations and extend their applicability and overall performance. In particular, our partial compilation approach has been developed by extending the state-of-the-art ASP solver WASP (Alviano et al. 2015) to include propagators from dynamic libraries, and a compiler that processes a compilable sub-program and generates the corresponding source code in C++, which is finally transformed in executable code by a C++ compiler.

To assess the efficacy of our approach, we conducted an experimental analysis on publicly-available benchmarks. Results show that our compilation-based approach improves on the state of the art in various scenarios, including cases in which the input program is stratified or the grounding makes the evaluation less efficient with traditional ASP systems.

2 Preliminaries

We recall some preliminary notions that are used in the remainder of the paper.

https://doi.org/10.1017/S1471068419000231 Published online by Cambridge University Press

2.1 Answer set programming

An ASP program π is a finite set of rules of the form $h_1|\ldots|h_n$:- $b_1,\ldots,b_m$. 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. 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 $\sim a$, where $\sim$ 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, $\overline{a} = \sim a$, for a negated atom $\sim a$, $\overline{\sim a} = a$. A rule is called a *constraint* if $n = 0$, and a *fact* if $n = 1$ and $m = 0$.

An object (atom, rule, etc.) is called *ground* or *propositional*, if it contains no variables. Given a program π, let the *Herbrand Universe* U_π be the set of all constants appearing in π and the *Herbrand Base* B_π be the set of all possible ground atoms which can be constructed from the predicate symbols appearing in π with the constants of U_π. Given a rule r, $Ground(r)$ denotes the set of rules obtained by applying all possible substitutions σ from the variables in r to elements of U_π. For a program π, the *ground instantiation* $Ground(\pi)$ of π is the set $\bigcup_{r\in\pi} Ground(r)$. Stable models of a program π are defined using its ground instantiation $Ground(\pi)$. An interpretation I for π is a set of literals s.t. $\forall a \in B_\pi$, either $a \in I$ or $\sim a \in I$ and $l \in I \implies \bar{l} \notin I$. 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. An interpretation I is a *model* for π if, for every $r \in Ground(\pi)$, 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 π w.r.t. a model I is the ground program π^I, obtained from π 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 π if I is a model of π, and there is no J such that J is a model of π^I and $J^+ \subset I^+$. A program π is *coherent* if it admits at least one stable model, *incoherent* otherwise.

A sub-program of π is a set of rules $\lambda \subseteq \pi$. In what follows, we denote by $\mathcal{P}(X)$ the set of predicate names appearing in X where X is an ASP expression (rule, rule head, literal, program, etc.) and we denote by $\mathcal{L}(X)$ the set of literals appearing in X, where X is again an ASP expression. In the following, $head_r$ and $body_r$ denote the head and the body of a rule r, respectively, while $body_r^+$ and $body_r^-$ denote the positive and the negative body of r, respectively. Moreover, given a set of rules λ, let $heads(\lambda) = \{a \mid a \in head_r, r \in \lambda\}$.

2.2 Loop unrolling and dead code elimination

In our work, we will mention two well-known optimizations used by compilers: *loop unrolling* and *dead code elimination* (Muchnick 1997). Loop unrolling is a loop transformation technique that, in the simplest formulation, removes the loop control instructions and replicates the loop body a number of times equal to the number of iterations, adjusting variables accordingly so to obtain an equivalent code. Dead code elimination is the removal of instructions that would never be executed, such as the body of conditional statements that are known to be false. Such techniques are typically implemented by exploiting information that is know at compile time.

https://doi.org/10.1017/S1471068419000231 Published online by Cambridge University Press

```
for(int j=0;j<n;j++) {                    for(int j=0;j<n;j++) {
 for(int i=0;i<3;i++) {                     a[0] = b[0] + j;
  if(i<1) { a[i] = b[i] + j; }  ⟹           b[1] = a[1] + j;
  else    { b[i] = a[i] + j; }              b[2] = a[2] + j;
}}                                        }
```

Fig. 1: Exemplification of loop unrolling and dead code elimination. The statements outlined in blue (i.e. lines 2–4) on the snippet on the left-hand side are transformed resulting in the code reported on the right-hand side.

We exemplify the effect of applying loop unrolling on the snippet of C++ code reported in Figure 1. Looking at the inner `for` statement (outlined in blue in Figure 1), we note that the number of iterations is fixed (to 3) and is known at compile time; thus, this loop can be unrolled by a compiler by writing three instantiations of the inner block of code, one for each of the three possible values of the loop controlling variable `i`, i.e., 0,1, and 2. In the resulting code, the three instances of the inner `if` statement (outlined in blue in Figure 1) contain conditions that can be evaluated at compile time (since variable `i` is replaced by its actual value by loop unrolling); thus, we apply dead code elimination removing the `if` statement and the code in the branch that will be never activated. The result of applying both loop unrolling and dead code elimination to our example is reported on the right-hand side of Figure 1. Note that the number of iterations of the outermost `for` statement depends on a variable `n`, thus it cannot be subject to loop unrolling at compile time because the value of `n` will be known only at execution time.

The potential benefits of applying these techniques become clear by observing that, in the original code, for each iteration of the outermost `for` statement one has to perform three increments of variable `i` and three evaluations of the `if` statement that are not present in the equivalent transformed code. Loop unrolling might not always be beneficial because the program size (generally) increases, leading to potential issues such as cache misses. Nonetheless, as it will be clearer in the following, the loops that are subject to unrolling in our technique typically require very few iterations (since they are limited to the number of predicates in the program or the number of literals in rules bodies). We refer to (Muchnick 1997) for more details about compilation techniques.

3 Conditions for splitting and compiling

In this section, we describe the conditions under which we allow the partial compilation.

The conditions for a sub-program to be compilable under our compilation-based approach are based on the concept of labeled dependency graph of an ASP program.

Definition 1
Given an ASP program π, the dependency graph of π, denoted DG_π, is a labeled graph (V, E) where V is the set of predicate names appearing in some head of π, and E is the smallest subset of $V \times V \times \{+, -\}$ such that (i) $(V_1, V_2, +) \in E$ if $\exists r \mid V_1 \in \mathcal{P}(body_r^+) \wedge V_2 \in \mathcal{P}(head_r)$; (ii) $(V_1, V_2, -) \in E$ if $\exists r \mid V_1 \in \mathcal{P}(body_r^-) \wedge V_2 \in \mathcal{P}(head_r)$; and (iii) $(V_1, V_2, -) \in E$ if $\exists r \mid V_1, V_2 \in \mathcal{P}(head_r)$.

Intuitively, the dependency graph contains positive (resp., negative) arcs from positive (resp., negative) body literals to head atoms, and negative arcs between atoms in a disjunctive head.

https://doi.org/10.1017/S1471068419000231 Published online by Cambridge University Press

Definition 2
An ASP program π is stratified iff DG_π has no loop containing a negative edge.

Definitions provided above are classical definitions for ASP programs, and now we define when an ASP sub-program is compilable.

Definition 3
Given an ASP program π, an ASP sub-program $\lambda \subseteq \pi$ is *compilable with respect to* π if both the following condition hold: (i) λ is a stratified ASP program and (ii) for all $p \in \mathcal{P}(heads(\lambda))$ it holds that $p \notin \mathcal{P}(\pi \setminus \lambda)$.

Intuitively, a (sub-)program is compilable if it is stratified and does not define any predicate that appears elsewhere in the program. This condition often applies in practice. Indeed, ASP encodings are often structured according to guess-and-check programming methodology, where the checking part (typically stratified rules and constraints) is captured by the above definition.

Example 1
Consider the following program π_1:

```
(1) in(X) | out(X) :- v(X).
(2) r(X,Y) :- e(X,Y).
(3) r(X,Y) :- e(X,Z), r(Z,Y).
(4) :- in(X), in(Y), not r(X,Y).
```

where `v(X)` and `e(X,Y)` model the nodes and edges of a graph, respectively. Program π_1 contains two compilable sub-programs, one given by constraint (4) and one given by constraint (4) together with rules (2) and (3). △

Note that (sets of) constraints are always compilable; indeed, rules having no head cannot cause any cycle in the dependency graph and trivially satisfy condition (ii) of Definition 3.

The following result is fundamental to understand our evaluation strategy.

Theorem 1
Let π be an ASP program and $\lambda \subseteq \pi$ be a *compilable* subprogram. For all answer sets M_π of π there exists an answer set $M_{\pi\setminus\lambda}$ of $\pi \setminus \lambda$ such that M_π is the unique answer set of the program $\{f. \mid f \in M^+_{\pi\setminus\lambda}\} \cup \lambda$.

Proof
The thesis follows from the splitting theorem (Lifschitz and Turner 1994). Observe that the set $\mathcal{L}(\pi \setminus \lambda)$, i.e., the literals appearing in $\pi \setminus \lambda$, is trivially a splitting set for π, where λ is the *top program* of π w.r.t. the splitting set, and $\pi \setminus \lambda$ is the *bottom program*. Moreover, λ is stratified and possibly includes constraints, thus it admits at most one answer set (Ceri et al. 1990). □

Assuming that one can compile λ in a specialized implementation, the above result suggests that one can compute an answer set M_π of a program π by first computing an answer set $M_{\pi\setminus\lambda}$ of $\pi \setminus \lambda$ (by using a standard ASP system), and then extending $M_{\pi\setminus\lambda}$ to M_π by computing (resorting to the compiled implementation of λ) the answer set of the union of λ with all atoms of $M_{\pi\setminus\lambda}$ as facts. This sketched principle is elaborated in the following.

https://doi.org/10.1017/S1471068419000231 Published online by Cambridge University Press

Algorithm 1 Solving with a compiled program

Input: ASP program π', ASP compilable program λ
Output: An answer set of $\pi = \pi' \cup \lambda$ or $\bot$ if π is incoherent
1: λ^{eval} =compile(λ)
2: $M_{\pi'}$ = answer_set(π')
3: **while** $M_{\pi'} \neq \bot$ **do**
4: $(C, M_{ext}) = \lambda^{eval}(M_{\pi'})$
5: **if** $C \neq \emptyset$ **then**
6: $\pi' = \pi' \cup C$
7: **else**
8: **return** M_{ext}
9: $M_{\pi'}$ = answer_set(π')
10: **return** $\bot$

4 Architecture for Partial Compilation

The architecture for evaluating ASP programs with partial compilation is formalized in Algorithm 1. The algorithm takes as input two ASP programs π' and λ, where λ is compilable with respect to $\pi = \pi' \cup \lambda$, and computes one answer set of π if it exists, otherwise it returns $\bot$ to denote that the input is incoherent. In the following λ_R denotes the set of stratified rules with non-empty head in λ and λ_C the set of constraints in λ. First the program λ is compiled obtaining the procedure λ^{eval}. Then, procedure *answer_set* (i.e., a standard ASP system comprising grounder and solver) is called to compute an answer set $M_{\pi'}$ of π'. If π' is incoherent then *answer_set* returns $\bot$ and Algorithm 1 terminates returning $\bot$. Otherwise, $M_{\pi'}$ is provided as input to the compiled program λ^{eval}, which returns a pair (C, M_{ext}), where C is a set of ground constraints having in the body only literals from $B_{\pi'}$, and M_{ext} is an answer set for $\pi' \cup \lambda_R$. We use subscript *ext* to denote that it is the extension of the answer set of π' with the answer set of λ_R. Importantly, C models a sufficient condition for discarding $M_{\pi'}$, and possibly also other candidate answer sets $M'_{\pi'}$ of π' that cannot be extended to answer sets of π because $M'_{\pi'} \cup \lambda$ is incoherent. If $C = \emptyset$ then $\lambda \cup M_{\pi'}$ is coherent, Algorithm 1 terminates, returning M_{ext} (line 8) which is an answer set of π (by Theorem 1). Otherwise, if $C \neq \emptyset$, C is added to π', so that the subsequent call to *answer_set* searches for another answer set of π'. The execution continues until π' is detected to be incoherent (line 3), and $\bot$ is returned (line 10), or an answer set is found.

The correctness of this evaluation strategy follows trivially from Theorem 1, once we have correct algorithms for *answer_set*, and λ^{eval}. How to obtain *answer_set* is well-known, thus in the following we describe the way in which we obtain λ^{eval}.

5 Compilation of sub-programs

In this section, we describe our strategy for compiling a sub-program λ to obtain procedure λ^{eval}. In order to simplify the presentation, we first describe a general-purpose evaluation strategy that is valid for any compilable input program, and then we describe how this strategy can be instantiated by transforming λ into a λ-specific algorithm that evaluates λ w.r.t. an answer set $M_{\pi'}$ of π' by applying loop unrolling and dead code elimination (see Section 2.2). The general purpose strategy is essentially composed of two components: (*i*) a procedure for computing bottom-up an answer set of a compil-

https://doi.org/10.1017/S1471068419000231 Published online by Cambridge University Press

Algorithm 2 BottomupEvaluation()

Input: ASP program $\lambda = \lambda_R \cup \lambda_C$, an answer set $M_{\pi'}$ of π'
Output: A set of ground constraints C and an interpretation M_{ext}

1: $R = M_{\pi'}$
2: $DG = \text{dependency\_graph}(\lambda)$
3: $SCCs = \text{topological\_sort}(DG)$
4: **for all** $SCC \in SCCs$ **do**
5: **for all** predicate $P \in SCC$ **do**
6: **for all** exit rules $r \in \lambda_R$ with $P \in \mathcal{P}(head_r)$ **do**
7: $S = \text{starter\_atom}(r)$
8: **for all** $s \in R_S$ **do**
9: $R_P = R_P \cup \text{evaluate}(r, s, R)$
10: **for all** predicate $P \in SCC$ **do**
11: $W_P = R_P$
12: **while** $\exists W_P \in W \mid W_P \neq \emptyset$ **do**
13: **while** $W_P \neq \emptyset$ **do**
14: **for all** $r \in \lambda_R \mid \mathcal{P}(head_r) \in SCC, P \in \mathcal{P}(body_r^+)$ **do**
15: **for all** $s \in W_P$ **do**
16: $E = \text{evaluate}(r, s, R)$
17: $W_{\mathcal{P}(head_r)} = W_{\mathcal{P}(head_r)} \cup (E \setminus R_{\mathcal{P}(head_r)})$
18: $R_{\mathcal{P}(head_r)} = R_{\mathcal{P}(head_r)} \cup E$
19: $W_P = W_P \setminus \{s\}$
20: $K = \emptyset$
21: **for all** $r \in \lambda_C$ **do**
22: $S = \text{starter\_atom}(r)$
23: **for all** $s \in R_S$ **do**
24: $K = K \cup \text{ground}(\text{r}, \text{s}, \text{R})$
25: $M_{ext} = R$
26: $C = \emptyset$
27: **for all** $c \in K$ **do**
28: $C = C \cup \{BuildConstraint(c, M_{\pi'}, M_{ext}, \lambda_R)\}$
29: **return** (C, M_{ext})

able program and a set of facts, and in case there does not exists one, (*ii*) an algorithm computing a set of constraints that are violated by the input facts.

Generic Bottom-up Evaluation. Historically, bottom-up semi-naïve algorithms are the standard way to evaluate stratified programs (Ceri et al. 1990). We also adopt this algorithm, that we have refactored and exemplified in pseudo-code in Algorithm 2 to make more clear how compilation specializes it depending on the program in input. In the algorithm, $SCCs$ denotes the topologically ordered set of the strongly connected components of the dependency graph DG_λ; and given a set of literals X, X_P denotes the set of literals in X whose predicate is P, thus W_P and R_P denotes sets of literals w.r.t. predicate P and we call them the *working set* and the *result set* of predicate P, respectively.

The evaluation of λ starts with the computation of the dependency graph DG of λ. Once the dependency graph is computed, the evaluation considers one strongly connected component (SCC) at a time, following a topological sort of the dependency graph. The for loops at line 5 and 10 iterate over all predicate names in the current SCC. Rules are classified into *exit* and *recursive*. A rule r is an exit rule for an SCC S if all predicates in

https://doi.org/10.1017/S1471068419000231 Published online by Cambridge University Press

Algorithm 3 BuildConstraint()

Input: A constraint c, an interpretation $M_{\pi'}$ of π', an answer set M_{ext}, the program λ_R
Output: A ground constraint

```
1: R = ∅, S = ∅
2: while c ≠ ∅ do
3:    l = NextLiteral(c)
4:    S = S ∪ {l}
5:    if P(l) ∈ P(π') then
6:       R = R ∪ {l' ∈ M_π' | l' ≐ l}
7:    else if P(l) ∉ P(π') ∧ positive(l) then
8:       for all r ∈ λ_R | l ≐^σ head_r do
9:          for all b ∈ body_r do
10:            c = c ∪ {σ(b)}
11:   else if P(l) ∉ P(π') ∧ negative(l) then
12:      for all r ∈ λ_R | l ≐^σ ~head_r do
13:         for all b ∈ body_r do
14:            c = c ∪ {σ(b̄)}
15:   c = c \ S
16: return toConstraint(R)
```

$\mathcal{P}(body_r)$ belong to a component that precedes S in the topological sort. Otherwise, r is said to be recursive, i.e. there is some body predicate in the body of r that belongs to S. For each SCC, *exit rules* are evaluated first (line 6), while *recursive rules* are evaluated whenever all exit rules of the SCC have been evaluated (line 14).

Rules are evaluated as nested join loops (Ceri et al. 1990; Garcia-Molina et al. 2009) and the join starts with an atom, called *starter atom*. For exit rules and constraints, we have only a single join loop and the starter atom is selected among positive body atoms of the rule. For recursive rules, we might have several join loops, and each starter atom is selected among atoms whose predicate belongs to the recursive component. The reason is that exit rules do not produce new atoms in the same component while recursive rules produce new atoms that can trigger new joins. A nested join loop of a rule r and a starter atom s is implemented by function *evaluate*, which returns a set of atoms that belong to the predicate of the head of r. For the evaluation of recursive rules, the algorithm takes advantage of a set W, used as a working set to accumulate the atoms of recursive predicates in the evaluation. The computation of constraints that are returned is done at the end of the bottom-up evaluation (from line 20) and takes advantage of the algorithm *BuildConstraint* described in the following. Note that for constraints we use the function *ground* which extends *evaluate* to produce ground constraints C generated from λ w.r.t. $M_{\pi'}$.

Handling Failed Constraints. We now describe how the constraints to be added to π' are computed. A non-trivial issue is that the constraints in the compiled program might consist of literals that do not appear in π'. Algorithm 3 presents a simplified pseudo-code of the procedure that we adopt in our implementation. The idea is to build a result set R of literals step by step starting from a ground constraint c. Note that c is initially ground, but during the execution of the algorithm non-ground literals might be added to it. In the following, we use the standard concept of variable-substitution σ that represents a mapping from variables to either constants or variables. At each step, the algorithm

https://doi.org/10.1017/S1471068419000231 Published online by Cambridge University Press

selects one literal l in c (function *NextLiteral(c)*). If the predicate of l appears in π' we add all the literals l' in $M_{\pi'}$ that *unifies* (symbol $\doteq$) it, i.e. there is a variable-substitution σ such that $\sigma(l) = l'$. Otherwise, if the predicate of l does not appear in π' and l is a positive literal, we add $\sigma(b)$, where b is a body literal of a rule whose head unifies with substitution σ (symbol $\overset{\sigma}{=}$) with l. Finally, if the predicate of l does not appear in π' and l is a negative literal, we add $\sigma(\bar{b})$, where b is a body literal of a rule whose negated head (denoted as $\sim head_r$) unifies with substitution σ (symbol $\overset{\sigma}{=}$) with l. The process continues until c becomes empty. The set of literals S stores literals that have already been processed to prevent loops. Note that Algorithm 3 starts from c that is known to be not satisfied in M_{ext}, and traces back (like in a top-down evaluation of a query) the computation of c from λ to identify a set of literals from $M_{\pi'}$ that imply c. Indeed, steps 7–10 replace a positive literal $l \in c$ by the body of a rule that can infer l, whereas steps 11–14 replace a negative literal $\sim l \in c$ with the negation of the body of the rules that could infer l but did not, and 5–6 instantiate the remaining literals in c w.r.t $M_{\pi'}$. Thus, at the end of the process, R will contain some literals in $M_{\pi'}$ that caused the derivation of c from λ and $M_{\pi'}$. Termination is guaranteed, since the same literal is not processed twice (step 15) and steps 7–14 replace literals until no l can be further replaced.

Compilation. The general purpose bottom-up evaluation strategy described above constitutes the template that is instantiated by the compiler depending on the program in input. In particular, the parts of Algorithm 2 outlined in blue (i.e. lines 2–7, 10, 12–14, and 20–22) contain instructions that can be evaluated at compile-time because they depend on the syntactic structure of the input; and thus they are subject to loop unrolling and dead code elimination. Moreover, the dependency graph and its SCCs are computed at compile time and eliminated after unrolling the loops mentioning them. The parts of the algorithm in black cannot be simplified and are kept in the compiled version to be executed at runtime. Thus, the compiler given a compilable program λ produces an ad-hoc procedure obtained by applying the transformations mentioned above to Algorithm 2, and obtains λ^{eval} (see Algorithm 1). Note that the output of the compiler is a procedure that computes the same result of Algorithm 2 only for the given λ.

6 Implementation and Experiments

The strategy has been implemented within the WASP solver by exploiting its C++ APIs. The fact that the implementation is embedded into a state-of-the-art ASP solver makes partial compilation more appealing due to the possibility to rely on the consolidated performance of a CDCL solver. In particular, when the solver starts it calls our implemented compiler, which compiles the input compilable program into a C++ dynamic library that implements a lazy propagator. Candidate models are passed to the dynamic library that computes the extended model and checks the constraints. The implementation is available at https://bitbucket.org/bernardo_cuteri/lazy_wasp.

We experimented with partial compilation in four different settings:

(E1) Compilation of stratified programs;
(E2) Partial compilation of constraints;
(E3) Partial compilation of rules and constraints; and
(E4) Partial compilation of rules.

https://doi.org/10.1017/S1471068419000231 Published online by Cambridge University Press

Time and memory for each run are limited to 10 minutes CPU-time and 6GB, respectively. In all experiments, we compare our system against the best ASP systems for the benchmark at hand. Concerning experiment (E1), ASP solvers are not included since the programs are already evaluated by ASP grounders. Concerning experiments (E2), (E3) and (E4), CLASP and WASP are used as a reference. Moreover, CLASP, WASP, and compilation-based approach use GRINGO as grounder. In addition, it should be noted that, being based on WASP, the most relevant result is given by how the compilation-based implementation compares with plain WASP.

Compilation times are reported exactly once per domain (thus, only on one instance) because the system automatically avoids compiling twice the same program, using an MD5 hash on the compiled program. This fits real-world use-cases where the program is fixed and the instance changes. In general, compilation times are negligible (up to 2.6 seconds) since we are compiling few rules (up to 15), the only exception being the *wine* encoding in OpenRuleBench that consists of 999 rules and takes some minutes to compile.

For what concernes what parts of the input programs are compiled, we report that in experiment (E1) we compile the whole program, while in all the others we find experimentally some sub-programs that are hard to ground. Sub-programs selection, is in general non trivial, but in many practical cases one can try to incrementally remove parts of the input program, respecting the compilability condition, until grounding becomes acceptable (e.g. the grounding step terminates in an acceptable amount of time).

For all experimental settings, we selected pre-existing benchmarks wherever possible and considered two new benchmarks (*connected k-cut, min-cut with transitive closure*) in the cases where we could not find any. New benchmarks consist of classical computer science problems possibly extended to fit the experiment use-case, naively encoded in ASP.

The results are commented in the following in a separate paragraph for each setting. The benchmarks are available for download at https://bitbucket.org/bernardo_cuteri/lazy_wasp.

(E1) Evaluation of stratified programs. Stratified programs are a large subset of ASP programs that allows to model and solve deductive database applications (Eiter et al. 2009). To test our implementation, we considered the well-known benchmarks called *OpenRuleBench*, which is an open community benchmark designed to test rule engines. In particular, we run perfect model computation as done for comparing ASP implementations in (Calimeri et al. 2017). We compared our method with three state-of-the-art ASP systems: GRINGO (Gebser et al. 2016), DLV (Leone et al. 2006), and I-DLV (Calimeri et al. 2017). Plain WASP is not included in this benchmark since stratified programs are already solved by grounders. Results are reported in a cactus plot in Figure 2 and clearly show the performance benefits of the compilation-based approach. Indeed, it solves more instances than state-of-the-art approaches and has in general lower running time.

(E2) Partial compilation of constraints. In this experiment, we considered two benchmarks presented in (Cuteri et al. 2017), namely StableMarriage and Natural Language Understanding (NLU). As shown in (Cuteri et al. 2017), the encodings of such benchmarks include some constraints leading to a grounding bottleneck. Cuteri et al.

https://doi.org/10.1017/S1471068419000231 Published online by Cambridge University Press

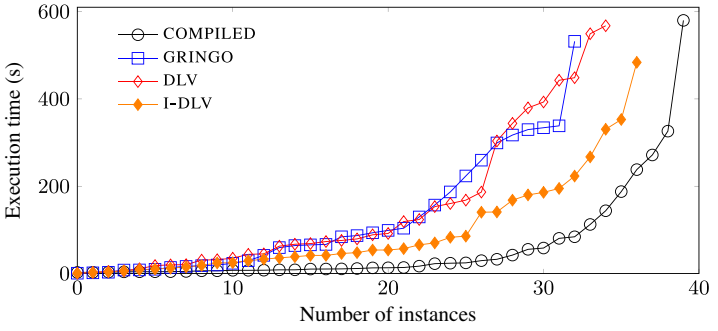

Fig. 2: (E1) OpenRuleBench benchmark

(2017) presented a strategy to lazily evaluate such constraints by means of custom Python scripts. Therefore, in the analysis, we compare our approach with these custom Python scripts.

The Stable Marriage benchmark is based on the well-known Stable Marriage problem where there are n men and m women, where each person has a preference order over the opposite sex and the problem consists in finding a marriage that is *stable* (i.e. there is no couple for which both partners would rather be married with each other than their current partner). Results are reported in Table 1. Each table row is associated to a different value of a parameter k of preferences, e.g. each man (resp. woman) gives the same preference to all the women (resp. men) but to $k\%$ of them a lower preference is given.

The NLU benchmark is about an application of ASP to Natural Language Understanding involving the computation of optimal solutions for First Order Horn Abduction

Table 1: (E2) Stable Marriage: Number of solved instances and average running time (in seconds).

Pref. (k%)	**CLASP**		**WASP**		**WASP PYTHON**		**COMPILED**	
	sol.	**avg t**	**sol.**	**avg t**	**sol.**	**avg t**	**sol.**	**avg t**
0	10	4.36	10	6.2	10	5.8	10	5.6
5	10	28.3	10	25.3	10	5.7	10	5.8
10	10	43.6	8	48.2	10	5.4	10	5.6
15	10	57.9	9	38.3	10	6.8	10	5.6
20	10	62.9	9	50	10	5.9	10	5.4
25	10	67.8	7	52.6	10	5.9	10	5.9
30	10	72.8	10	60.1	10	6	10	5.7
35	10	84.4	5	111.4	10	6.3	10	8.3
40	10	87.6	7	63.3	10	9.4	10	20
45	10	92.0	8	83.8	10	6.3	10	11.3
50	10	94.7	9	67.9	10	6.4	10	8.3
55	10	95.13	7	124.4	9	7.2	9	9.4
60	10	96.36	8	63.3	10	11.5	9	10.7
65	10	99.8	6	66.7	6	18.2	9	17.1
70	10	98.9	6	71	3	21.8	5	132.3
75	10	96.0	8	89.9	0	-	1	13.8
80	10	99.3	7	148.9	0	-	0	-
85	10	107.7	6	107.2	0	-	0	-
90	10	278.7	9	152.2	0	-	0	-
95	8	295.6	10	70.3	0	-	0	-
100	10	98.8	8	61.9	1	7.3	0	-
Tot solved	206		167		139		143	

https://doi.org/10.1017/S1471068419000231 Published online by Cambridge University Press

Table 2: (E2) NLU Benchmark: Number of solved instances and average running time (in seconds).

Obj. Func.	CLASP		WASP		WASP PYTHON		COMPILED	
	sol.	avg t	sol.	avg t	sol.	avg t	sol.	avg t
card	46	63.7	48	83.0	50	2.8	50	2.3
coh	45	68.6	48	83.0	50	26.8	49	18.3
wa	46	90.5	48	103.2	49	23.6	49	38.5

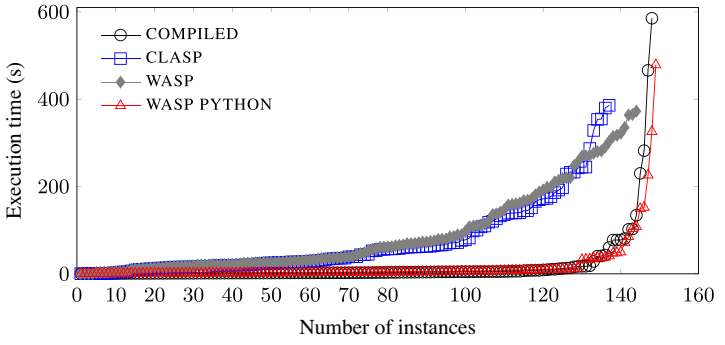

Fig. 3: (E2) NLU Benchmark: Cumulative results of all cost functions.

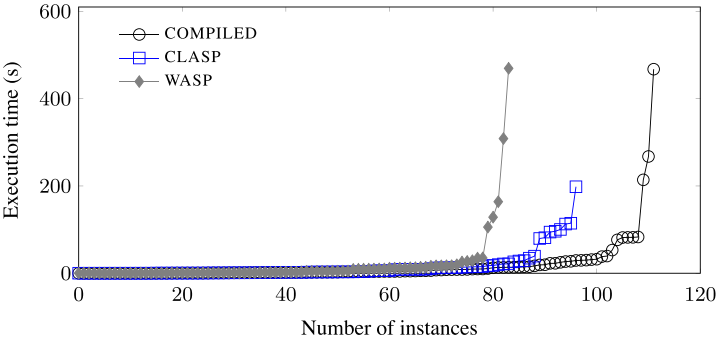

Fig. 4: (E3) Connected k-cut benchmark

problems under cost functions cardinality, cohesion, and weighted abduction. Results are reported in Table 2 and Figure 3. Each row in the table presents the result obtained for a specific cost function, while the figure presents the cumulative results for all cost functions.

It is possible to observe that our evaluation strategy works best in the same settings in which constraint lazy instantiators work (Cuteri et al. 2017), i.e., when the removed constraints are hard to ground, but easy to satisfy. The reason is that our evaluation follows the same execution pattern of lazy constraints, i.e., check the constraint on answer set candidates of the original input program without the lazy constraint. It is important to emphasize here that approaches from (Cuteri et al. 2017) are hand-written by experts, whereas our approach automatically generates the source code with no need of expertise in an imperative language and solver internals/APIs (i.e., the purely declarative solving approach is preserved).

https://doi.org/10.1017/S1471068419000231 Published online by Cambridge University Press

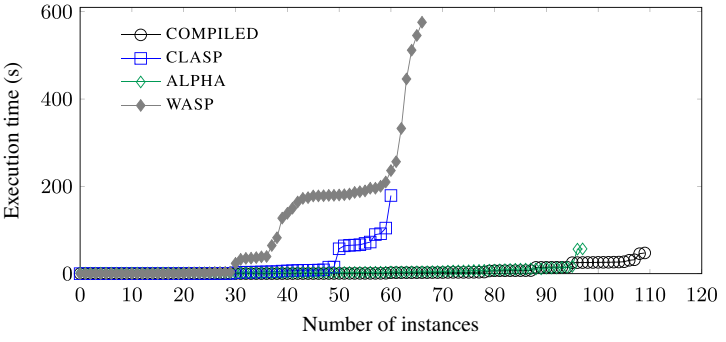

Fig. 5: (E3) Non-partition removal coloring benchmark

(E3) Partial compilation of rules and constraints. In this experiment, we consider two benchmarks: *connected k-cut* and *non-partition removal coloring*.

Connected k-cut is a graph problem where the goal is to find a cut of size at least k such that the two formed partitions are connected. Instances were randomly generated containing graphs with different numbers of nodes (from 200 to 800), different densities (from 0.001 to 0.25) and different cut sizes (from 50 to 800). Non-partition removal coloring is a benchmark inspired by a real-world configuration application (Gebser et al. 2015) and proposed by Bogaerts and Weinzierl (2018). The formulation of the problem is as follows: given a directed graph, the goal is to remove one vertex in such a way that the transitive closures of the original and of the resulting graph are equal on the remaining nodes and that the resulting graph is 3-colorable. Instances were taken from (Bogaerts and Weinzierl 2018).

Results are reported in Figures 4 and 5. Concerning *connected k-cut*, compilation-based approach solves 15 and 28 more instances than CLASP and WASP, respectively. Similar results can be observed also in the benchmark *non-partition removal coloring*. Indeed, compilation-based approach outperforms both CLASP and WASP, solving 49 and 43 more instances, respectively. For the sake of completeness, in this benchmark, we included in the analysis the lazy-solver ALPHA (Weinzierl 2017). Indeed, albeit ALPHA is not competitive in general with state-of-the-art solvers, in this benchmark it outperforms both CLASP and WASP. However, ALPHA cannot reach the performance of the compilation-based approach (which solves 12 instances more with similar average running times).

(E4) Partial compilation of rules. In this experiment, we consider the min-cut problem with transitive closure. Given a graph G the goal is to compute a minimum cost cut of G and to compute the transitive closures of the two resulting partitions. In order to analyze the performance of compilation-based approach on sub-programs without constraints, in this benchmark the compiled sub-program is only made of rules. Results are reported in Figure 6, where we observe that CLASP is much faster than WASP solving 15 more instances. Such a gap is partially filled by the compilation-based approach which is able to solve 8 more instances than plain WASP.

Summary of the results. Experiments show that the approach is particularly effective for solving stratified programs (E1) and for compiling grounding intensive sub-programs. For what concernes stratified programs, the evaluation is bottom-up as implemented in the other compared systems, but the compilation approach pays off due to its specificity. In

https://doi.org/10.1017/S1471068419000231 Published online by Cambridge University Press

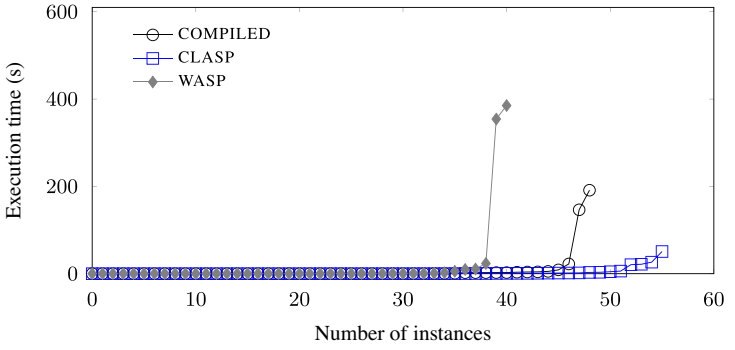

Fig. 6: (E4) Min-cut with transitive closure

experiment (E2), where only constraints are compiled, the approach works similarly w.r.t. the custom lazy instantiators implemented in (Cuteri et al. 2017): good performances when the constraint is easy to satisfy, but hard do ground. This behaviour has been already shown empirically in (Cuteri et al. 2017) and can easily be observed, for example, in the Stable Marriage results (small values of k). In (E3), the approach is effective also in presence of rules. In the *k-cut* benchmark WASP is originally slower than CLINGO, but the compiled approach is faster than CLINGO. Moreover, the compiled approach behaves well w.r.t. lazy grounding approaches as shown in the non-partition removal coloring benchmark. Finally, in (E4) the compiled is again able to improve on the performance of the base solver WASP when only rules (no constraints) are compiled.

7 Related Work

Traditional evaluation strategy of ASP systems is based on two steps, namely *grounding* and *solving*; for both phases, several efficient systems have been proposed. Concerning the grounding, state-of-the-art grounders are DLV (Faber et al. 2012), GRINGO (Gebser et al. 2011) and IDLV (Calimeri et al. 2017); which are all based on semi-naïve database evaluation techniques (Ullman 1988) for avoiding duplicate work during grounding. Concerning ASP solvers, the first generation, i.e., SMODELS (Simons et al. 2002) and DLV (Leone et al. 2006), was based on a DPLL-like algorithm extended with inference rules specific to ASP. Modern ASP solvers such as CLASP (Gebser et al. 2015) and WASP (Alviano et al. 2015) include mechanisms for conflict-driven clause learning and for non-chronological backtracking. Both solvers also offer an external interface to simplify the integration of custom solving strategies in the main search algorithm. In particular, we used the interface of WASP to implement the techniques described in the paper. Alternative approaches are based on the lazy grounding of the whole program, e.g., GASP (Dal Palù et al. 2009), ASPERIX (Lefevre et al. 2017), or ALPHA (Weinzierl 2017), where all rules are instantiated lazily; this makes the search less informed but might have a better memory footprint. These 'fully lazy' approaches have in common, that they instantiate even the non-stratified part of the program only when rule bodies of the respective rules are satisfied in the current assignment of the search process, as opposed to our approach where all guesses are instantiated upfront and only stratified parts depending on guesses (including constraints) are computed lazily. Our Algorithm 3 computes

https://doi.org/10.1017/S1471068419000231 Published online by Cambridge University Press

constraints that are related to Justifications (Bogaerts and Weinzierl 2018), with the difference that our approach needs ground constraints using only atoms from π', while the ALPHA solver uses nonground constraints computed from Justifications branches that are cut off at the first negated literal. CASP (Balduccini and Lierler 2017; Ostrowski and Schaub 2012) and ASPMT (Bartholomew and Lee 2014) can solve problems with large constraints, but extend the language with external theories. The compilable program definition is related to Rule Splitting Sets of HEX programs (Eiter et al. 2016), however, we here define them on the basis of predicates, not partially ground atoms. ASP Modules (Janhunen et al. 2009) are more permissive than compilable subprograms because they permit mutually cyclic (negative) dependencies among modules, which is not possible in compilable subprograms.

8 Conclusion

Compilation-based approaches are meant to speed up computation by exploiting information known at compilation time to create custom procedures that are specific to the problem at hand. In this paper, we presented what is, to the best of our knowledge, the first work on compilation-based techniques for ASP solving. In our approach, we allow compilation of ASP sub-programs and we define what a compilable sub-program is, i.e., we specify what are the conditions under which our approach can be adopted. The presented approach has been developed as a solver extension of WASP which is a state-of-the-art ASP solver. The evaluation strategy presented includes a bottom-up evaluation for computing the unique stable model of the compilable sub-program and a top-down evaluation for computing failed constraints in terms of literals that are known to the ASP solver. An experimental analysis shows the benefits that can be obtained in different use-cases by a compilation-based approach. The approach is particularly suited for solving stratified programs, and for compiling ground-intensive sub-programs where lazy instantiators are effective.

In the future, we are planning to extend the presented approach to allow eager/post propagation, i.e., the evaluation is performed also on partial interpretations every time a new literal is chosen (eager) or when unit propagation ends (post). Moreover, it is also interesting to investigate whether it is possible to automatically select a sub-program to be compiled that maximizes the performance of our technique.

Acknowledgments

This work has been partially supported by MIUR under PRIN 2017 project n. 2017M9C25L 001 (CUP H24I17000080001), and from the EU's Horizon 2020 research and innovation program under grant agreement No 825619 (AI4EU).

References

Alviano, M., Dodaro, C., Leone, N., and Ricca, F. 2015. Advances in WASP. In *LPNMR*. LNCS, vol. 9345. Springer, 40–54.

Amendola, G., Greco, G., Leone, N., and Veltri, P. 2016. Modeling and reasoning about NTU games via answer set programming. In *IJCAI*. IJCAI/AAAI Press, 38–45.

https://doi.org/10.1017/S1471068419000231 Published online by Cambridge University Press

Aschinger, M., Drescher, C., Friedrich, G., Gottlob, G., Jeavons, P., Ryabokon, A., and Thorstensen, E. 2011. Optimization methods for the partner units problem. In *CPAIOR*. 4–19.

Balduccini, M. and Lierler, Y. 2017. Constraint answer set solver EZCSP and why integration schemas matter. *TPLP 17,* 4, 462–515.

Bartholomew, M. and Lee, J. 2014. System aspmt2smt: Computing ASPMT theories by SMT solvers. In *JELIA*. Lecture Notes in Computer Science, vol. 8761. Springer, 529–542.

Bogaerts, B. and Weinzierl, A. 2018. Exploiting justifications for lazy grounding of answer set programs. In *IJCAI*. 1737–1745.

Brewka, G., Eiter, T., and Truszczynski, M. 2011. Answer set programming at a glance. *Commun. ACM 54,* 12, 92–103.

Calimeri, F., Fuscà, D., Perri, S., and Zangari, J. 2017. I-DLV: the new intelligent grounder of DLV. *Intelligenza Artificiale 11,* 1, 5–20.

Ceri, S., Gottlob, G., and Tanca, L. 1990. *Logic Programming and Databases*. Surveys in computer science. Springer.

Cuteri, B., Dodaro, C., Ricca, F., and Schüller, P. 2017. Constraints, lazy constraints, or propagators in ASP solving: An empirical analysis. *TPLP 17,* 5-6, 780–799.

Cuteri, B., Reale, K., and Ricca, F. 2019. A logic-based question answering system for cultural heritage. In *JELIA*. Lecture Notes in Computer Science, vol. 11468. Springer, 526–541.

Dal Palù, A., Dovier, A., Pontelli, E., and Rossi, G. 2009. GASP: answer set programming with lazy grounding. *Fundam. Inform. 96,* 3, 297–322.

Dodaro, C., Gasteiger, P., Leone, N., Musitsch, B., Ricca, F., and Schekotihin, K. 2016. Combining answer set programming and domain heuristics for solving hard industrial problems (application paper). *TPLP 16,* 5-6, 653–669.

Dodaro, C. and Ricca, F. 2018. The external interface for extending WASP. *TPLP in press CORR abs/1811.01692*.

Eiter, T., Fink, M., Ianni, G., Krennwallner, T., Redl, C., and Schüller, P. 2016. A model building framework for answer set programming with external computations. *TPLP 16,* 4, 418–464.

Eiter, T., Ianni, G., and Krennwallner, T. 2009. Answer set programming: A primer. In *Reasoning Web*. Lecture Notes in Computer Science, vol. 5689. Springer, 40–110.

Erdem, E., Gelfond, M., and Leone, N. 2016. Applications of answer set programming. *AI Magazine 37,* 3, 53–68.

Erdem, E. and Öztok, U. 2015. Generating explanations for biomedical queries. *TPLP 15,* 1, 35–78.

Faber, W., Leone, N., and Perri, S. 2012. The intelligent grounder of DLV. In *Correct Reasoning*. Lecture Notes in Computer Science, vol. 7265. Springer, 247–264.

Garcia-Molina, H., Ullman, J. D., and Widom, J. 2009. *Database systems - the complete book (2. ed.)*. Pearson Education.

Gebser, M., Kaminski, R., Kaufmann, B., Ostrowski, M., Schaub, T., and Wanko, P. 2016. Theory solving made easy with clingo 5. In *ICLP TCs*. OASICS, vol. 52. 2:1–2:15.

Gebser, M., Kaminski, R., Kaufmann, B., Romero, J., and Schaub, T. 2015. Progress in clasp series 3. In *LPNMR*. Lecture Notes in Computer Science, vol. 9345. Springer, 368–383.

Gebser, M., Kaminski, R., König, A., and Schaub, T. 2011. Advances in *gringo* series 3. In *LPNMR*. LNCS, vol. 6645. Springer, 345–351.

Gebser, M., Leone, N., Maratea, M., Perri, S., Ricca, F., and Schaub, T. 2018. Evaluation techniques and systems for answer set programming: a survey. In *IJCAI*. ijcai.org, 5450–5456.

Gebser, M., Ryabokon, A., and Schenner, G. 2015. Combining heuristics for configuration problems using answer set programming. In *LPNMR*. Springer, 384–397.

Gelfond, M. and Lifschitz, V. 1991. Classical negation in logic programs and disjunctive databases. *New Generation Comput. 9,* 3/4, 365–386.

Ianni, G., Martello, A., Panetta, C., and Terracina, G. 2009. Efficiently querying RDF(S) ontologies with answer set programming. *J. Log. Comput. 19,* 4, 671–695.

Janhunen, T., Oikarinen, E., Tompits, H., and Woltran, S. 2009. Modularity Aspects of Disjunctive Stable Models. *Journal Of Artificial Intelligence Research 35*, 813–857.

Kojo, T., Männistö, T., and Soininen, T. 2003. Towards intelligent support for managing evolution of configurable software product families. In *SCM*. LNCS, vol. 2649. Springer, 86–101.

Koponen, L., Oikarinen, E., Janhunen, T., and Säilä, L. 2015. Optimizing phylogenetic supertrees using answer set programming. *TPLP 15,* 4-5, 604–619.

Lefevre, C., Béatrix, C., Stéphan, I., and Garcia, L. 2017. Asperix, a first-order forward chaining approach for answer set computing. *Theory and Practice of Logic Programming 17,* 3, 266–310.

Leone, N., Allocca, C., Alviano, M., Calimeri, F., Civili, C., Costabile, R., Fiorentino, A., Fuscà, D., Germano, S., Laboccetta, G., Cuteri, B., Manna, M., Perri, S., Reale, K., Ricca, F., Veltri, P., and Zangari, J. 2019. Enhancing DLV for large-scale reasoning. In *LPNMR*. Lecture Notes in Computer Science, vol. 11481. Springer, 312–325.

Leone, N., Pfeifer, G., Faber, W., Eiter, T., Gottlob, G., Perri, S., and Scarcello, F. 2006. The DLV system for knowledge representation and reasoning. *ACM TOCL 7,* 3, 499–562.

Lierler, Y., Maratea, M., and Ricca, F. 2016. Systems, engineering environments, and competitions. *AI Magazine 37,* 3, 45–52.

Lifschitz, V. and Turner, H. 1994. Splitting a logic program. In *ICLP*. MIT Press, 23–37.

Manna, M., Ricca, F., and Terracina, G. 2015. Taming primary key violations to query large inconsistent data via ASP. *TPLP 15,* 4-5, 696–710.

Marileo, M. C. and Bertossi, L. E. 2010. The consistency extractor system: Answer set programs for consistent query answering in databases. *Data Knowl. Eng. 69,* 6, 545–572.

Muchnick, S. S. 1997. *Advanced Compiler Design and Implementation*. Morgan Kaufmann.

Nogueira, M., Balduccini, M., Gelfond, M., Watson, R., and Barry, M. 2001. An A Prolog decision support system for the space shuttle. In *Answer Set Programming*.

Ostrowski, M. and Schaub, T. 2012. ASP modulo CSP: the clingcon system. *TPLP 12,* 4-5, 485–503.

Simons, P., Niemelä, I., and Soininen, T. 2002. Extending and implementing the stable model semantics. *Artif. Intell. 138,* 1-2, 181–234.

Ullman, J. D. 1988. *Principles of Database and Knowledge-Base Systems, Volume I*. Principles of computer science series, vol. 14. Computer Science Press.

Weinzierl, A. 2017. Blending Lazy-Grounding and CDNL Search for Answer-Set Solving. In *LPNMR*. LNCS, vol. 10377. 191–204.