Proceedings of the 17th International Conference on Principles of Knowledge Representation and Reasoning (KR 2020)
Main Track

Answer Set Programming with Composed Predicate Names

Mario Alviano
DEMACS, University of Calabria, Italy
alviano@mat.unical.it

Abstract

Mainstream systems for Answer Set Programming implement intelligent grounding to eliminate object variables from the input program, often obtaining a propositional program of reasonable size. However, non-stratified negation may inhibit the simplification of some rule bodies due to the lack of knowledge on the truth of recursive atoms. Frustration is greatest when the program is clearly locally stratified, such as in case of numerical arguments in rule heads obtained by increasing some body arguments; common examples are minimal distances in graphs and time arguments in planning scenarios. This paper suggests to move some arguments in predicate names, so that the declarative semantics of Answer Set Programming is preserved, but non-stratified negation is possibly avoided thanks to symbolic rule instantiation. A proof of concept is given in terms of Jinja templates for arguments with a clear range.

1 Introduction

Answer Set Programming (ASP) is a popular language for nonmonotonic reasoning based on stable model semantics (Gelfond and Lifschitz 1991). Mainstream ASP systems eliminate object variables from the input program by means of *intelligent grounding* techniques (Kaufmann et al. 2016), possibly enhanced by magic sets in case of queries (Alviano and Faber 2011; Alviano et al. 2019b), so that stable model search is later performed on a propositional program by efficient solvers like CLASP (Gebser, Kaufmann, and Schaub 2012) and WASP (Dodaro et al. 2011; Alviano et al. 2019a). The grounding process starts by determining an order of the predicates based on head-to-body dependencies: the grounding of a rule with predicate p in the head needs to know the instances of any predicate q in its body, that is, rules defining q cannot be grounded after those defining p. When predicates are totally ordered, intelligent grounding is very efficient because each rule is processed only once. For otherwise, efficiency is achieved by semi-naive evaluation (Ullman 1988), that is, rules that must be processed multiple times are evaluated only for new substitutions of object variables.

Intelligent grounding simplifies the produced rules based on the knowledge of true and potentially true atoms (Faber, Leone, and Perri 2012). In particular, producing a rule with atom $p(\bar{t})$ in the head witnesses the potential truth of $p(\bar{t})$ in some stable models; if the body of the rule is empty, $p(\bar{t})$ must be true in all stable models. If $p(\bar{t})$ is known to be true, all rules in which it occurs negatively can be removed, as well as all its occurrences in positive bodies. Moreover, falsity of some atoms is determined thanks to the order in which predicates are processed: while processing predicate p, every atom $q(\bar{t})$ not known to be potentially true, and such that q precedes p, must be false in all stable models; accordingly, any `not` $q(\bar{t})$ in rule bodies can be removed.

Non-stratified negation may inhibit these simplifications. For example, for any $n \geq 0$, the rule

```
odd(X+1) :- X = 0..n-1, not odd(X).
```

defines odd numbers in the interval $[0..n]$. However, when X is replaced by 0, there is no knowledge that `odd(0)` must be false in all stable models, and therefore intelligent grounding produces `odd(1):- not odd(0)`, failing to understand that `odd(1)` is true in all stable models. Consequently, intelligent grounding produces `odd(2):- not odd(1)`, and so on. We observe that this ground program has no recursion, in particular involving negation. In fact, it is true that predicate `odd` negatively depends on itself, but argument `X` is increased in rule head, and therefore every rule produced by intelligent grounding only depends on previously produced rules. Something similar happens for planning scenarios, where many atoms have an argument representing a time step (Dimopoulos, Nebel, and Koehler 1997; Lifschitz 2002; Dimopoulos et al. 2019).

For these cases, we propose to move some arguments in predicate names, hence introducing the notion of *composed predicate name*. The example above would become

```
odd_{X+1} :- X = 0..n-1, not odd_X.
```

Semantically, there is no difference, but pragmatically we can now expand the symbolic program before starting intelligent grounding. We actually provide a proof of concept of the proposed idea by means of Jinja templates (https://palletsprojects.com/p/jinja/). Our example becomes

```
{% for X in range(0, n) %}
    odd_{{ X+1 }} :- not odd_{{ X }}.
{% endfor %}
```

Experiments on computing distances in graphs and on grounding planning instances show that this simple solution works in practice when the range of the arguments in com-

posed predicate names is clear and of reasonable size. On the other hand, there is no performance gain on using the proposed approach when the grounding bottleneck is already eliminated via some other techniques.

2 Background

We introduce only the minimal background to present our results, and refer the literature for details (Gebser et al. 2012). Let V, C, and P be fixed, nonempty, countable sets of (object) variables, (object) constants, and predicate symbols. A term is either a variable or a constant, and a predicate name is a predicate symbol. An atom has the form $p(t_1, \ldots, t_n)$, where p is a predicate name, $n \geq 0$, and each t_i is a term ($i \in [1..n]$). A rule r is an expression of the form

$$\alpha_0 \text{ :- } \alpha_1, \ldots, \alpha_m, \textbf{not } \alpha_{m+1}, \ldots, \textbf{not } \alpha_n.$$

where $n \geq m \geq 0$, and each α_i is an atom ($i \in [0..n]$). Let $H(r) := \alpha_0$, $B^+(r) := \{\alpha_1, \ldots, \alpha_m\}$, and $B^-(r) := \{\alpha_{m+1}, \ldots, \alpha_n\}$; r is a *fact* if $B^+(r) = B^-(r) = \emptyset$. A program Π is a set of rules. An expression (atom, rule, or program) is *ground* if it contains no variables.

For a program Π, let $gr(\Pi)$ be the set $\bigcup_{r \in \Pi} gr(r)$, where $gr(r)$ denotes the set of rules obtainable from rule r by replacing variables with constants. An *interpretation* I is a set of ground atoms, intuitively those interpreted as true. I satisfies a ground rule r if $H(r) \in I$ (true head), or $B^+(r) \not\subseteq I$ (false positive body), or $B^-(r) \cap I \neq \emptyset$ (false negative body). I is a *model* of a program Π, denoted $I \models \Pi$, if I satisfies all rules in $gr(\Pi)$. The *reduct* of Π wrt. I, denoted Π^I, is obtained from $gr(\Pi)$ by removing all rules whose body is false. I is a *stable model* of a program Π if $I \models \Pi$ and there is no $J \subset I$ such that $J \models \Pi^I$. Let $SM(\Pi)$ denote the set of stable models of Π.

Example 1. *Given a graph represented by facts* `vertex(x)` *and* `edge(x,y)`*, minimal distances between its vertices can be computed by means of the following program* Π_{dist}*:*

```
dist(X,X,0) :- vertex(X).
dist(X,Y,D+1) :- dist(X,Z,D), edge(Z,Y),
    not less(X,Y,D+1), D < vertices.
less(X,Y,D+1) :- dist(X,Y,D).
less(X,Y,D+1) :- less(X,Y,D), D < vertices.
```

where `vertices` *is the number of vertices in the graph, and arithmetic is naturally interpreted. For example, given* Π_1

```
vertex(a). vertex(b). vertex(c).
edge(a,b). edge(b,c). edge(b,a). edge(c,b).
```

program $gr(\Pi_1 \cup \Pi_{dist})$ *contains, among others, the rules*

```
dist(a,a,0) :- vertex(a).

dist(a,b,1) :- dist(a,a,0), edge(a,b),
    not less(a,b,1), 0 < 3.
less(a,a,1) :- dist(a,a,0).
less(a,b,1) :- dist(a,b,0).

dist(a,a,2) :- dist(a,b,1), edge(b,a),
    not less(a,a,2), 1 < 3.
dist(a,c,2) :- dist(a,b,1), edge(b,c),
    not less(a,c,2), 1 < 3.
less(a,a,2) :- less(a,a,1), 1 < 3.
```

Algorithm 1: Grounding(Π: program)

```
1 L := ∅;  U := ∅;  Π' := ∅;
2 Let C_1, ..., C_n be a top. order. for the SCCs of G_Π;
3 for i ∈ [1..n] do
4     foreach r ∈ gr(def(Π, C_i)) s.t. B+(r) ⊆ U and
        H(r) ∉ L and B-(r) ∩ L = ∅ do
5         B+(r) := B+(r) \ L;
6         B-(r) := {p(t̄) ∈ B-(r) | p(t̄) ∈ U or p ∈ C_i};
7         Π' := Π' ∪ {r};  U := U ∪ {H(r)};
8         if r is a fact then L := L ∪ {H(r)};
9 return Π';
```

and the only stable model in $SM(\Pi_1 \cup \Pi_{dist})$ *comprises the following instances of* `dist(X,Y,D)`*:* `(a,a,0)`*,* `(b,b,0)`*,* `(c,c,0)`*,* `(a,b,1)`*,* `(b,a,1)`*,* `(b,c,1)`*,* `(c,b,1)`*,* `(a,c,2)`*, and* `(c,a,2)`*.* ■

Given a program Π, *intelligent grounding* aims at obtaining a subset Π' of $gr(\Pi)$ such that $SM(\Pi') = SM(\Pi)$. Intuitively, intelligent grounding simplifies rules according to deterministic knowledge on the processed program. It assumes *safety* of rules, that is, every variable occurring in a rule r must also occur in $B^+(r)$, and instantiates rules following a topological ordering of the strongly connected components (SCCs) of the *dependency graph* G_Π, that is, a directed graph whose nodes are predicates, and having two kinds of links: positive links from p to p' if p and p' occur respectively in $H(r)$ and $B^+(r)$, for some rule r of Π; negative links from p to p' if p and p' occur respectively in $H(r)$ and $B^-(r)$, for some rule r of Π. Π is *stratified* (wrt. negation) if there is no cycle in G_Π involving negative links. Π is *locally stratified* (Przymusinski 1988) if $gr(\Pi)$ is stratified.

To ease the presentation, a naive grounding procedure is reported as Algorithm 1, where $def(\Pi, C_i)$ denotes the set of rules of Π such that the predicate of $H(r)$ belongs to C_i. Lower and upper bounds on the set of true atoms are stored in L and U, respectively, and processes every rule r in $def(\Pi, C_i)$ trying to unify $B^+(r)$ with U, the atoms having chances to be true; rules whose head is already known to be true or whose negative body is known to be false are skipped (line 4). Rules instantiated in this way are simplified by removing atoms in $B^+(r)$ that are known to be true (line 5), and atoms in $B^-(r)$ that are known to be false (line 6), that is, those not occurring in U and whose predicates belong to already processed components. The new rule witnesses the possibility for its head to be true (line 7), and unequivocally determines the truth of its head if the body is empty (line 8).

Example 2 (Continuing Example 1)**.** *Let* $\{vertex\}$, $\{edge\}$, $\{dist, less\}$ *be the topological ordering used by Algorithm 1 for instantiating* $\Pi_1 \cup \Pi_{dist}$*. After processing the first two components,* L*,* U *and* Π' *contain all facts in* Π_1*. Processing the last component requires several iterations on the rules of* Π_{dist}*; focusing on the ground rules from Example 1, their simplification is the following:*

```
dist(a,a,0).
```

```
dist(a,b,1) :- not less(a,b,1).
less(a,a,1).

dist(a,a,2) :- dist(a,b,1), not less(a,a,2).
dist(a,c,2) :- dist(a,b,1), not less(a,c,2).
less(a,a,2).
```

Note that `less(a,b,1):- dist(a,b,0)` *is not produced because no rule with* `dist(a,b,0)` *in the head is ever generated. Note also that* `dist(a,a,2):- dist(a,b,1), not less(a,a,2)` *is produced before adding* `less(a,a,2)` *to L.* ■

Even if the actual grounding procedures implemented by ASP systems are much more sophisticated than Algorithm 1, their behavior on programs like Π_{dist} is essentially the one described in Example 2. This fact can be verified by instantiating $\Pi_1 \cup \Pi_{dist}$ with CLINGO (https://potassco.org/clingo/) using the command-line option `--text`.

3 Composed Predicate Names

ASP is extended by allowing the use of *composed predicate names*, that is, predicate symbols possibly carrying terms. Formally, a composed predicate name has the form $p_{t_1,\ldots,t_n}$, where p is a predicate symbol, $n \geq 0$, and each t_i is a term ($i \in [1..n]$). Definitions from Section 2 naturally extend to the new notion of predicate name.

Example 3. *The problem from Example 1 can be represented by the following program* Π_{cdist}*:*

```
dist_0(X,X) :- vertex(X).
dist_{D+1}(X,Y) :- dist_D(X,Z), edge(Z,Y),
    not less_{D+1}(X,Y), D < vertices.
less_{D+1}(X,Y) :- dist_D(X,Y).
less_{D+1}(X,Y) :- less_D(X,Y), D < vertices.
```

Program $gr(\Pi_1 \cup \Pi_{cdist})$ *is essentially one-to-one to* $gr(\Pi_1 \cup \Pi_{dist})$*. However, grounding composed predicate names only, results into a stratified program, which can be later efficiently processed by intelligent grounding.* ■

Let us first define a one-to-one mapping to eliminate composed predicate names. To ease the presentation, for every $n \geq m \geq 0$ and atom $p_{t_{m+1},\ldots,t_n}(t_1,\ldots,t_m)$ occurring in a program Π, predicate p is assumed to be associated with a unique *arity*, that is, the pair $(m, n-m)$. Define $tr(p_{t_{m+1},\ldots,t_n}(t_1,\ldots,t_m))$ to be $p(t_1,\ldots,t_n)$, and extend tr to any expression in the natural way. In particular, $tr(\Pi)$ and $tr(SM(\Pi))$ are obtained from program Π and $SM(\Pi)$ by replacing every atom α with $tr(\alpha)$. Note that the (unique) arity of p is $(m, n-m)$ in Π, and $(0, n)$ in $tr(\Pi)$. Moreover, note that $tr(\Pi_{cdist})$ is Π_{dist}, and $tr(SM(\Pi_{cdist}))$ is $SM(\Pi_{dist})$; such links are formalized by the next theorem.

Theorem 1. *Let* Π *be a program with composed predicate names. It holds that* $SM(tr(\Pi)) = tr(SM(\Pi))$.

The claim above clarifies that composed predicate names are essentially syntactic sugar from a semantic point of view. However, we already suggested that their instantiation can lead to programs with good properties, essentially more fine-grained dependency graphs.

Example 4 (Continuing Example 3)**.** *Let* Π *be the program obtained from* Π_{cdist} *by replacing variable* D *with all natural numbers. A topological ordering of the SCCs of* G_Π *is the following:* $\{vertex\}$, $\{edge\}$, $\{dist_0\}$, $\{less_1\}$, $\{dist_1\}$, $\{less_2\}$*, and so on. Algorithm 1 on* $\Pi_1 \cup \Pi$ *would produce the following rules:*

```
dist_0(a,a). dist_0(b,b). dist_0(c,c).
less_1(a,a). less_1(b,b). less_1(c,c).
dist_1(a,b). dist_1(b,a). dist_1(b,c). dist_1(c,b).
less_2(a,a). less_2(b,b). less_2(c,c).  ...
dist_2(a,c). dist_2(c,a).
less_3(a,a). less_3(b,b). less_3(c,c).  ...
```

All rules are actually facts because Π *is stratified. Note that no rules are produced after processing the SCC* $\{less_3\}$*, which however does not guarantee the termination of Algorithm 1 because* Π *contains infinitely many rules.* ■

The above example suggests the need for a finite instantiation of composed predicate names in order to make their use of practical interest. The simplest solution is to limit the range of all variables occurring in composed predicate names. Let $range$ be a partial function from variables to finite sets of constants. A rule r is (finitely) *restricted* by $range$ if $B^+(r)$ contains the built-in relation $X \in range(X)$ whenever X is a variable occurring in a composed predicate name of r. A program Π is *restricted* by $range$ if all its rules are.

Example 5. *Let* $range(D)$ *be the interval* $[0..vertices-1]$*. Below is a restricted version of* Π_{cdist} *from Example 3:*

```
dist_0(X,X) :- vertex(X).
dist_{D+1}(X,Y) :- dist_D(X,Z), edge(Z,Y),
    not less_{D+1}(X,Y), D = 0..vertices-1.
less_{D+1}(X,Y) :- dist_D(X,Y), D=0..vertices-1.
less_{D+1}(X,Y) :- less_D(X,Y), D=0..vertices-1.
```

Note that the range is expressed by `D = 0..vertices-1`*, a common ASP construct for integer intervals.* ■

Composed predicate names of restricted programs can be easily instantiated. Formally, let Π be a program restricted by $range$. For a rule r of Π, let $exp(r, range)$ be the set of rules constructible from r by replacing X with constants in $range(X)$, for all variables X for which $range$ is defined, in all possible ways. For example, expanding

```
less_{D+1}(X,Y) :- dist_D(X,Y), D=0..vertices-1.
```

from Example 5, with `vertices` $= 3$, leads to

```
less_1(X,Y) :- dist_0(X,Y), 0=0..3-1.
less_2(X,Y) :- dist_1(X,Y), 1=0..3-1.
less_3(X,Y) :- dist_2(X,Y), 2=0..3-1.
```

Let $exp(\Pi, range)$ be the union of $exp(r, range)$, for all rules $r \in \Pi$. Observe that program $exp(\Pi, range)$ is finite and without composed predicate names, hence it is suitable for being processed by intelligent grounding algorithms.

Theorem 2. *Algorithm 1 on* $exp(\Pi, range)$ *produces a ground program* Π' *such that* $SM(\Pi) = SM(\Pi')$.

4 Jinja Templates and Experiments

The idea of composed predicate names can be implemented, among other possibilities, by means of Jinja templates,

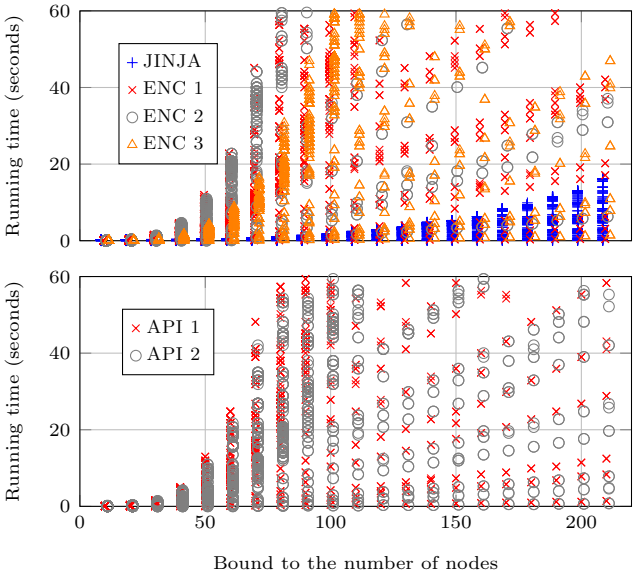

Figure 1: Computation of distances in graphs by CLINGO.

a modern and designer-friendly templating language for Python. Jinja templates are essentially text files with *statements*, denoted by `{% ... %}`, and *expressions*, denoted by `{{ ... }}`. Statements are used for conditionals and iterations, while expressions are used to print literal and non-literal content. For example, distances of nodes in a graph can be computed by means of the following template:

```
dist_{{ 0 }}(X,X) :- vertex(X).
{% for D in range(0, vertices) %}
  dist{{ D+1 }}(X,Y) :- dist_{{ D }}(X,Z),
    edge(Z,Y), not less_{{ D+1 }}(X,Y).
  less_{{ D+1 }}(X,Y) :- dist_{{ D }}(X,Y).
  less_{{ D+1 }}(X,Y) :- less_{{ D }}(X,Y).
{% endfor %}
```

We conducted some experiments to compare the performance of CLINGO 5.3.0 (Gebser et al. 2019) on different encodings (available on https://www.mat.unical.it/~alviano/experiment-kr2020.zip). Test cases were ran on an Intel Xeon 2.4 GHz with 16 GB of memory, with time and memory limited to 60 seconds and 15 GB, respectively. The aim of our experiments is to measure differences in grounding time, and in particular to highlight cases in which grounding is already a bottleneck for ASP systems; the subsequent stable model search, if applicable, was not ran because uninformative for this paper (intuitively, reordering a ground program may lead to dramatic and unpredictable execution times of stable model search). Hence, CLINGO was ran with the command-line option `--mode=gringo`.

We first computed distances in graphs from the ASP Competition (Alviano et al. 2013), in particular from the *Graceful Graph* problem. For each graph, we obtained different testcases by limiting the computation to the first n nodes (for a total of 3570 testcases). Results are shown in Figure 1, and clearly highlight that the Jinja template above scales much better than traditional ASP representations of this problem

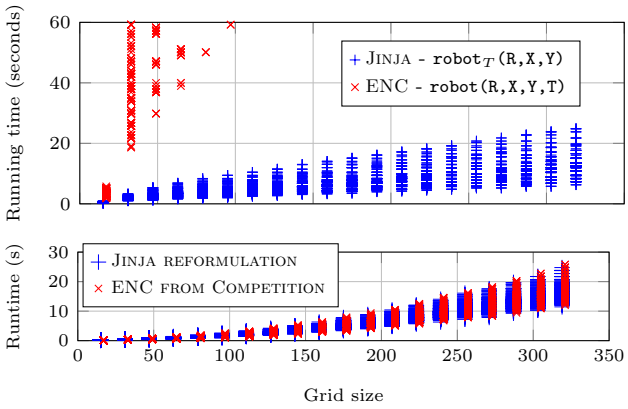

Figure 2: Grounding of Ricochet robots instances with CLINGO.

(ENC 1 is essentially Π_{dist}), even if negation is stratified as in the following program (ENC 2):

```
d(X,X,0) :- vertex(X).
d(X,Y,D+1) :- d(X,Z,D),edge(Z,Y),D<vertices.
dist(X,Y,D):-d(X,Y,D);not d(X,Y,D'):D'=0..D-1.
```

or in the following program (ENC 3):

```
d(X,X,0) :- vertex(X).
d(X,Y,D+1) :- d(X,Z,D),edge(Z,Y),D<vertices.
out(X,Y,D) :- d(X,Y,D), d(X,Y,D'), D > D'.
dist(X,Y,D) :- d(X,Y,D), not out(X,Y,D).
```

In this case, Jinja templates are convenient also wrt. using multi-shot computation to instantiate Π_{dist} for increasing values of D (API 1), even if predicates `less`$_D$ are compactly replaced by a single predicate `done` (API 2). We also observed no timeout using Jinja, 1709, 2201 and 1656 timeouts using traditional ASP encodings, and 1888 and 1731 timeouts using multi-shot computation.

A second set of testcases is obtained from Ricochet robots (Gebser et al. 2015), a problem asking to coordinate robots in a grid. For each instance, we obtained different testcases by increasing the size of the grid (for a total of 4780 testcases). Results are shown in Figure 2. The first plot evidences that Jinja templates are very convenient if positions of robots are represented by atoms of the form `robot_T(R,X,Y)`. The second plot is relative to the advanced encoding from ASP competitions, which represents positions of robots on the two axes with different predicates to avoid the grounding bottleneck. This encoding is not further improved by the use of Jinja templates because almost all produced ground atoms are relevant for stable model search. We observed 4217 timeouts with the encoding using `robot_T(R,X,Y)`, and no timeouts with other encodings.

5 Conclusion

Composed predicate names are simple and semantically aligned to common constructs of ASP. Arguments in composed predicate names that have a clear range can be easily processed by means of Jinja templates, and in some cases this is sufficient to eliminate the grounding bottleneck. We suggest to use Jinja templates in those cases, and

only if ranges have reasonable size. More sophisticated expansion techniques are left as future lines of research, as well as a comparison with Datalog extended with monotonic aggregates (Zaniolo 2015; Shkapsky, Yang, and Zaniolo 2015), stream reasoning (Beck, Dao-Tran, and Eiter 2018; Eiter, Ogris, and Schekotihin 2019), lazy grounding (Bomanson, Janhunen, and Weinzierl 2019), and theory-based extensions of ASP (Janhunen, Liu, and Niemelä 2011).

Acknowledgments

This work was partially supported by MISE under projects S2BDW (F/050389/01-03/X32) and ALCMEONE (F/050502/03/X32), by Regione Calabria under project DLV LargeScale (CUP J28C17000220006), and by Gruppo Nazionale per il Calcolo Scientifico (GNCS-INdAM).

References

Alviano, M., and Faber, W. 2011. Dynamic magic sets and super-coherent answer set programs. *AI Commun.* 24(2):125–145.

Alviano, M.; Calimeri, F.; Charwat, G.; Dao-Tran, M.; Dodaro, C.; Ianni, G.; Krennwallner, T.; Kronegger, M.; Oetsch, J.; Pfandler, A.; Pührer, J.; Redl, C.; Ricca, F.; Schneider, P.; Schwengerer, M.; Spendier, L. K.; Wallner, J. P.; and Xiao, G. 2013. The fourth answer set programming competition: Preliminary report. In Cabalar, P., and Son, T. C., eds., *LPNMR 2013, Corunna, Spain, September 15-19*, volume 8148 of *LNCS*, 42–53. Springer.

Alviano, M.; Amendola, G.; Dodaro, C.; Leone, N.; Maratea, M.; and Ricca, F. 2019a. Evaluation of disjunctive programs in WASP. In Balduccini, M.; Lierler, Y.; and Woltran, S., eds., *LPNMR 2019, Philadelphia, PA, USA, June 3-7, 2019, Proceedings*, volume 11481 of *Lecture Notes in Computer Science*, 241–255. Springer.

Alviano, M.; Leone, N.; Veltri, P.; and Zangari, J. 2019b. Enhancing magic sets with an application to ontological reasoning. *Theory Pract. Log. Program.* 19(5-6):654–670.

Beck, H.; Dao-Tran, M.; and Eiter, T. 2018. LARS: A logic-based framework for analytic reasoning over streams. *Artif. Intell.* 261:16–70.

Bomanson, J.; Janhunen, T.; and Weinzierl, A. 2019. Enhancing lazy grounding with lazy normalization in answer-set programming. In *AAAI 2019, Honolulu, Hawaii, USA, January 27 - February 1, 2019*, 2694–2702. AAAI Press.

Dimopoulos, Y.; Gebser, M.; Lühne, P.; Romero, J.; and Schaub, T. 2019. plasp 3: Towards effective ASP planning. *Theory Pract. Log. Program.* 19(3):477–504.

Dimopoulos, Y.; Nebel, B.; and Koehler, J. 1997. Encoding planning problems in nonmonotonic logic programs. In Steel, S., and Alami, R., eds., *Recent Advances in AI Planning, ECP'97, Toulouse, France, September 24-26, 1997, Proceedings*, volume 1348 of *LNCS*, 169–181. Springer.

Dodaro, C.; Alviano, M.; Faber, W.; Leone, N.; Ricca, F.; and Sirianni, M. 2011. The birth of a WASP: preliminary report on a new ASP solver. In Fioravanti, F., ed., *CILC 2011, Pescara, Italy, August 31 - September 2, 2011*, volume 810 of *CEUR Workshop Proceedings*, 99–113.

Eiter, T.; Ogris, P.; and Schekotihin, K. 2019. A distributed approach to LARS stream reasoning (system paper). *Theory Pract. Log. Program.* 19(5-6):974–989.

Faber, W.; Leone, N.; and Perri, S. 2012. The intelligent grounder of DLV. In Erdem, E.; Lee, J.; Lierler, Y.; and Pearce, D., eds., *Correct Reasoning - Essays on Logic-Based AI in Honour of Vladimir Lifschitz*, volume 7265 of *LNCS*, 247–264. Springer.

Gebser, M.; Kaminski, R.; Kaufmann, B.; and Schaub, T. 2012. *Answer Set Solving in Practice*. Synthesis Lectures on Artificial Intelligence and Machine Learning. Morgan & Claypool Publishers.

Gebser, M.; Kaminski, R.; Obermeier, P.; and Schaub, T. 2015. Ricochet robots reloaded: A case-study in multi-shot ASP solving. In Eiter, T.; Strass, H.; Truszczynski, M.; and Woltran, S., eds., *Advances in Knowledge Representation, Logic Programming, and Abstract Argumentation - Essays Dedicated to Gerhard Brewka on the Occasion of His 60th Birthday*, volume 9060 of *LNCS*, 17–32. Springer.

Gebser, M.; Kaminski, R.; Kaufmann, B.; and Schaub, T. 2019. Multi-shot ASP solving with clingo. *Theory Pract. Log. Program.* 19(1):27–82.

Gebser, M.; Kaufmann, B.; and Schaub, T. 2012. Conflict-driven answer set solving: From theory to practice. *Artif. Intell.* 187:52–89.

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

Janhunen, T.; Liu, G.; and Niemelä, I. 2011. Tight integration of non-ground answer set programming and satisfiability modulo theories. In *Proc. of the 1st Workshop on Grounding and Transformations for Theories with Variables*.

Kaufmann, B.; Leone, N.; Perri, S.; and Schaub, T. 2016. Grounding and solving in answer set programming. *AI Magazine* 37(3):25–32.

Lifschitz, V. 2002. Answer set programming and plan generation. *Artificial Intelligence* 138:39–54.

Przymusinski, T. C. 1988. On the declarative semantics of deductive databases and logic programs. In Minker, J., ed., *Foundations of Deductive Databases and Logic Programming*. Morgan Kaufmann. 193–216.

Shkapsky, A.; Yang, M.; and Zaniolo, C. 2015. Optimizing recursive queries with monotonic aggregates in deals. In Gehrke, J.; Lehner, W.; Shim, K.; Cha, S. K.; and Lohman, G. M., eds., *31st IEEE International Conference on Data Engineering, ICDE 2015, Seoul, South Korea, April 13-17, 2015*, 867–878. IEEE Computer Society.

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

Zaniolo, C. 2015. Expressing and supporting efficiently greedy algorithms as locally stratified logic programs. In Vos, M. D.; Eiter, T.; Lierler, Y.; and Toni, F., eds., *Tech. Comm. of ICLP 2015, Cork, Ireland, August 31 - September 4, 2015*, volume 1433 of *CEUR Workshop Proceedings*.