An Infrastructure for Stream Reasoning with Incremental Grounding

Giovambattista Ianni[1], Francesco Pacenza[1], and Jessica Zangari[1]

Department of Mathematics and Computer Science, University of Calabria
lastname@mat.unical.it - https://www.mat.unical.it

Abstract. In the context of multiple, repeated, execution of reasoning tasks, typical of stream reasoning and other applicative settings, we propose an incremental reasoning infrastructure, based on the answer set semantics. We focus particularly on the possibility of caching and re-using ground programs, thus knocking down the time necessary for performing this demanding task when it has to be repeated on similar knowledge bases. We present the outline of our incremental caching technique and report about our preliminary experiments.

1 Introduction

The practice of attributing meaning to quantified logical sentences using a grounded propositional version thereof dates back to the historical work of Jacques Herbrand [18]. Later, at the end of the past century, ground programs have been used as the operational basis for computing the semantics of logic programs in the context of the answer set semantics [17] and of the well-founded semantics [23]. The traditional structure of an answer set solver includes indeed two separated steps: a grounding module, which pre-processes an input, non-ground knowledge base, and produces a propositional theory; and a model generator, which computes the actual semantics in form of *answer sets*. Structurally similar pre-processing steps are taken when low-level constraint sets, or propositional SAT theories are obtained from high-level, non-ground input languages [21].

The generation of a propositional ground theory can be both time and space consuming and, as such, the grounding phase cannot be overlooked as a light preprocessing stage. There are a number of both application and benchmark settings in which the grounding step is prominent in terms of used resources [16]. Note also that grounding can be of EXPTIME complexity if arbitrarily long rules are allowed in input. Indeed, when focusing to the answer set semantics, a number of optimization techniques aim to reduce space and time costs of the grounding step [1, 4, 5, 15], or to blend it within the answer set search phase [8, 19, 22, 24].

In the context of stream reasoning and multi-shot evaluation [2, 3, 14], a quite typical setting is when the grounding step is repeatedly executed on slightly different input data, while a short computation time window is allowed. The contributions of this paper are the following:

– in the spirit of early truth maintenance systems [9], we set our proposed incremental technique through a multi-shot reasoning engine, based on answer set semantics, whose usage workflow allows continuous updates, query and reason over a stored knowledge base;
– we focus on caching ground programs, or parts thereof, whose re-evaluation can thus be avoided when repeated, similar reasoning tasks are issued to our engine. Our proposal is comparable to early and recent work on incremental update of datalog materializations (see [20] for an overview). Such approaches focus however on query answering over stratified Datalog and materialize just query answers. Our focus is instead on the generalized setting in which disjunction and unstratified negation is allowed and propositional logic programs are materialized and maintained;
– our stored knowledge bases grow monotonically from one shot to another, becoming more and more general, yet larger than usual ground programs. We show, in preliminary experiments, that this approach, which we called "overgrounding", pays off in terms of performance. We expect this setting to be particularly favourable when non-ground input knowledge bases are constituted of small set of rules, typical of declaratively programmed videogame agents, or robots.

In the following, after some brief preliminaries, we show the basic structure of our caching strategy, and we briefly illustrate our framework. Then we report about some preliminary experiments.

2 Preliminaries

We assume to deal with knowledge bases under the answer set semantics (Answer Set Programming (ASP) in the following [10,12,17]). A knowledge base KB is a set of rules. A rule r is in the form: $\alpha_1 \vee \alpha_2 \vee \cdots \vee \alpha_k$:- $\beta_1, \ldots, \beta_n$, *not* $\beta_{n+1}, \ldots,$ *not* β_m where $m \geqslant 0$, $k \geqslant 0$; $\alpha_1, \ldots, \alpha_k$ and $\beta_1, \ldots, \beta_m$ are atoms. An atom is in the form $p(\mathbf{X})$, where p is a predicate name and $\mathbf{X}$ is a list of terms that are either constants or a variables. A knowledge base (resp. a rule, an atom, a term) is said to be *ground* if it contains no variables. The *head* of r is defined as $H(r) = \{\alpha_1, \ldots, \alpha_k\}$; if $H(r) = \emptyset$ then r is a *constraint*. The set of all head atoms in KB is denoted by $Heads(P) = \bigcup_{r \in P} H(r)$. The *positive body* of r is defined as $B^+(r) = \{\beta_1, \ldots, \beta_n\}$. The *negative body* of r is defined as $B^-(r) = \{not\ \beta_{n+1}, \ldots, not\ \beta_m\}$. The *body* of r is defined as $B(r) = B^+(r) \cup B^-(r)$; if $B(r) = \emptyset$, $\|H(r)\| = 1$ and r is ground, then r is referred to as a *fact.*

Given a knowledge base KB and a set of facts F, the *Herbrand universe* of KB and F, denoted by $U_{KB,F}$, consists of all (ground) terms that can be built combining constants appearing in KB or in F. The *Herbrand base* of $KB \cup F$, denoted by $B_{KB,F}$, is the set of all ground atoms obtainable from the atoms of KB by replacing variables with elements from $U_{KB,F}$.

A *substitution* for a rule $r \in KB$ is a mapping from the set of variables of r to the set $U_{KB,F}$ of ground terms. A *ground instance* of a rule r is obtained applying a substitution to r. Given a knowledge base KB and a set of facts F the *instantiation (grounding)* $grnd(KB \cup F)$ of $KB \cup F$ is defined as the set of all ground instances of its rules. The answer sets $AS(KB \cup F)$ of KB are set of facts, defined as the minimal models of the so-called FLP reduct of $grnd(KB \cup F)$ [11].

3 Overgrounding and caching

In order to compute $AS(KB \cup F)$ for given knowledge base KB and set of facts F, state-of-the-art grounders usually compute a refined propositional program, obtained from a subset gKB of $grnd(KB \cup F)$ (see e.g. [4]). gKB is equivalent in semantics to the original knowledge base, i.e. $AS(gKB) = AS(grnd(KB \cup F)) = AS(KB \cup F)$. In turn, gKB is usually obtained using a refined version of the common immediate consequence operator. The choice of the instantiation strategy impacts on both computing time and on the size of the obtained instantiated program. Grounders usually maintain a set PT of "possibly true" atoms, initialized as $PT = F$; then, PT is iteratively incremented and used for instantiating only "potentially useful" rules, up to a fixpoint. Strategies for decomposing programs and for rewriting, simplifying and eliminating redundant rules can be of great help in controlling the size of the final instantiation [5, 7].

Let S be a set of ground atoms or ground rules. Let $Inst(KB, S)$ be defined as

$$Inst(KB, S) = \{r \in grnd(KB) \ s.t. \ B^+(r) \subseteq S\}$$

whenever S is intended as a set of rules, with a slight abuse of notation, we define $Inst(KB, S)$ as $Inst(KB, Heads(S))$.

The above operator can be seen as a way for generating and selecting only ground rules that can be built by using a set of allowed ground atoms S. If S is initially set to a set of input facts F, one can obtain a bottom-up constructed ground program equivalent to $KB \cup F$ by iteratively applying $Inst$.

Theorem 1 (adapted from [4]). For a set of facts F, we define $Inst(KB, F)^k$ as the k-th element of the sequence $Inst(KB, F)^0 = Inst(KB, \emptyset \cup F), \ldots, Inst(KB, F)^k = Inst(KB, Inst(KB, F)^{k-1} \cup F)$. The sequence $Inst(KB, F)^k$ converges in a finite number of steps to a finite fixed point $Inst(KB, F)^\infty$ and

$$AS(Inst(KB, F)^\infty \ \cup \ F) = AS(grnd(KB \cup F))$$

Assume that for a fixed knowledge base KB, we wish to compute a series of multi-shot evaluations in which input facts are changing according to a given sequence $F_1, \ldots, F_n$, i.e. we aim at computing the sets $AS(KB \cup F_1), \ldots, AS(KB \cup F_n)$.

The following holds:

Theorem 2. Let $UF_k = \bigcup_{1 \leq i \leq k} F_i$. It holds that

$$AS(Inst(KB, UF_k)^\infty \cup F_k) = AS(KB \cup F_k)$$

Input: a stored ground program $G_k = Inst(KB, UF_k)^\infty$
Input: union of "accumulated" input facts $UF_k = \bigcup_{1 \leq i \leq k} F_k$
Input: current input facts F_{k+1}
Output: updated ground program G_{k+1}, updated accumulated facts UF_{k+1}

```
1: Procedure Differential-GROUND($G_k$,$UF_{k+1}$,$F_{k+1}$)
2:     $\Delta G = Inst(KB, UF_k \cup F_{k+1})^\infty \setminus G_k$
3:     $G_{k+1} = G_k \cup \Delta G$
4:     $UF_{k+1} = UF_k \cup F_{k+1}$
```

Fig. 1: Incremental Grounding Algorithm.

In particular, for each k,

$$Inst(KB, UF_k)^\infty \subseteq Inst(KB, UF_{k+1})^\infty \tag{1}$$

Our caching strategy, shown in figure 1, can be outlined as follows: let a grounder be subject to a consecutive number of runs, in which KB is kept constant, while different sets of input facts are given. We keep $G_k = Inst(KB, UF_k)^\infty$ in memory as the result of grounding and caching previous processing steps. Whenever a new ground program is needed for processing new input facts F_{k+1}, we compute

$$\Delta G = Inst(KB, UF_{k+1})^\infty \setminus G_k \tag{2}$$

Then we obtain $G_{k+1} = G_k \cup \Delta G$. The newly obtained ground program G_{k+1} can be used for performing querying and reasoning tasks such as, e.g., computing $AS(KB \cup F_{k+1}) = AS(G_{k+1} \cup F_{k+1})$.

In other words, we ground KB with respect to a, monotonically increasing, set of accumulated facts UF_i; on the other hand, the sequence of input facts F_i can be arbitrary (i.e. a F_{i+1} can in principle be non-overlapping with previous input fact sets). Nonetheless, a given ground program G_k can be used for computing answer sets of KB with respect to all F_i for all $i \leq k$.

Clearly ΔG is not computed by evaluating (2) but in an efficient and incremental way. In our case we developed a variant of the typical iteration which is at the core of the known semi-naive algorithm. Such logic has been implemented in the $\mathcal{I}$-*DLV* grounder [5, 7].

4 System Architecture

An high-level infrastructure for incremental grounding is depicted in Figure 2. The system provides a server-like behaviour and allows to keep the main process alive, waiting for incoming requests. Once a client U establishes a connection with $\mathcal{I}$-*DLV*-incr, a private working session SC is opened. Within SC, U can specify, using XML commands, tasks to be carried out. In particular, after loading a KB along with an initial set of facts F_1, the system can be asked to

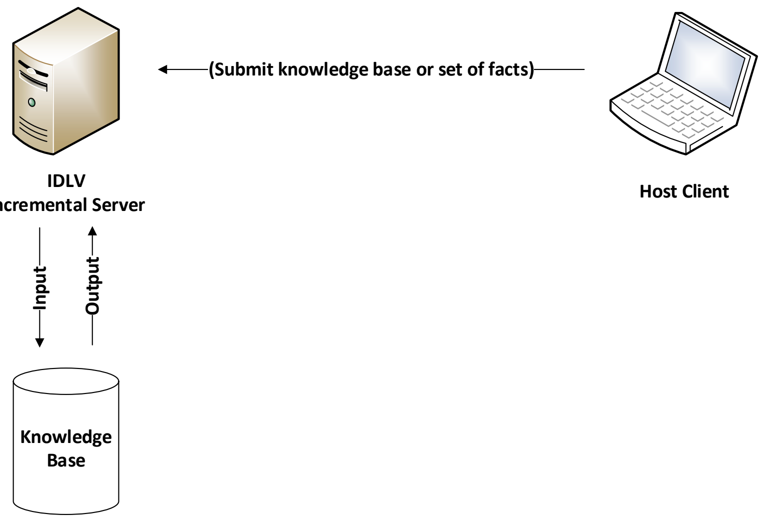

Fig. 2: An infrastructure for incremental grounding.

perform the grounding of KB over F_1; $Inst(KB, F_1)^\infty$ is then stored on server side. Then, further loading and grounding requests may be specified. U can provide additional sets of facts F_i for $1 < i \leq n$ so that $Inst(KB, UF_i)^\infty$ with $UF_i = \bigcup_{1 \leq i \leq n} F_i$ is computed. At each step i, the system is in charge of internally managing incremental grounding steps and automatically optimizing the computation by avoiding the re-instantiation of ground rules generated in a step $j < i$.

5 Benchmarks

Hereafter we report the results of a preliminary experimental activity carried out to assess the effectiveness of our incremental reasoning infrastructure. Experiments have been performed on a NUMA machine equipped with two 2.8 GHz AMD Opteron 6320 processors and 128GB of RAM. Unlimited time and memory were granted to running processes.

As benchmark, we considered the Sudoku domain. The classic Sudoku puzzle, or simply "Sudoku", consists of a tableau featuring 81 cells, or positions, arranged in a 9 by 9 grid. The grid is divided into nine sub-tableaux (regions, or blocks) containing nine positions each. Initially, in the game setup a number of positions are filled with a number between 1 and 9. The problem consists in checking whether the empty positions can be filled with numbers in a way such that each row, each column and each block shows all digits from 1 to 9 exactly once. When solving a Sudoku, players typically adopt deterministic inference strategies allowing, possibly, to obtain a solution. Several deterministic strategies are known [6]; herein, we take into account two simple strategies, namely, "naked single" and "hidden single". The former one permits to entail that a number n has to be associated to a cell C when all other numbers are excluded to be in C; for instance, in a Sudoku of 9 rows and 9 columns, assuming that we inferred that all numbers between 1 and 8 cannot be in the cell $(1, 1)$, then, it must contain 9. The hidden single strategy, instead, allows to derive that only a cell of a row/column/block can be associated with a particular number; for instance, in a Sudoku of 9 rows and 9 columns, the only cell that can contain 3 is $(4, 5)$ if, according to Sudoku rules, all other cells in the same block of $(4, 5)$, row 4 and column 5 cannot hold the number 3.

The iterated application of inference rules to given Sudoku tables is a good test for appreciating the impact of the incremental evaluation, since updated Sudoku tables contain all logical assertions derived in previous iterations.

In the experiments, we considered Sudoku tables of size 16x16 and 25x25 and experimented with knowledge bases, under answer set semantics, encoding deterministic inference rules. We compared two different evaluation strategies: (*i*) $\mathcal{I}$-*DLV*-incr implementing the incremental approach, and (*ii*) $\mathcal{I}$-*DLV*-no-incr which is endowed with the server-like behaviour but does not apply any incremental evaluation policy. Both systems have been executed in a server-like fashion. For a given Sudoku table the two inference rules above are modelled via ASP logic programs (as reported in [6]). The resulting answer set encodes a new tableau, possibly deriving new numbers to be associated to initially empty cells, and reflecting the application of inference rules; the new tableau is given as input to the system and again, by means of the same inferences, possibly, new cell values are entailed. The process is iterated until no further association is found. In general, given a Sudoku, it cannot be assumed that the deterministic approach leads to a complete solution; thus, for each considered Sudoku size, we selected only instances which are completely solvable with the two inference rules described above.

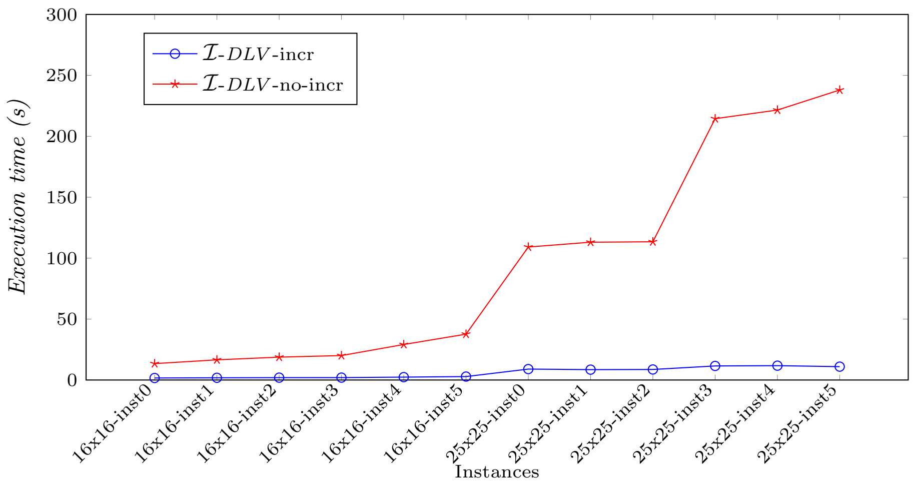

Fig. 3: Experiments on Sudoku benchmarks.

Results are depicted in Figure 3: instances are ordered by increasing time spent during the grounding stage by $\mathcal{I}$-*DLV*-no-incr. For each instance, it is reported the total grounding time (in seconds) computed over all iterations. $\mathcal{I}$-*DLV*-incr required at most 12 seconds to iteratively solve each instance and performed clearly better than $\mathcal{I}$-*DLV*-no-incr that instead required up to 237 seconds with an improvement of 95%. Figure 4 shows a closer look on the performance obtained in the instance 4 of size 25x25 which is the one requiring the highest amount of time to be solved and the highest number of iterations: for each iteration, the grounding time (in seconds) is reported. In the first iteration, both configurations spent almost the same time; for each further iteration, $\mathcal{I}$-*DLV*-incr required an average time of 0.13 seconds with a time reduction of 98% w.r.t. $\mathcal{I}$-*DLV*-no-incr showing an average time about 5.95 seconds. Overall, this behaviour confirms the potential of our incremental grounding approach in scenarios involving updates in the underlying knowledge base.

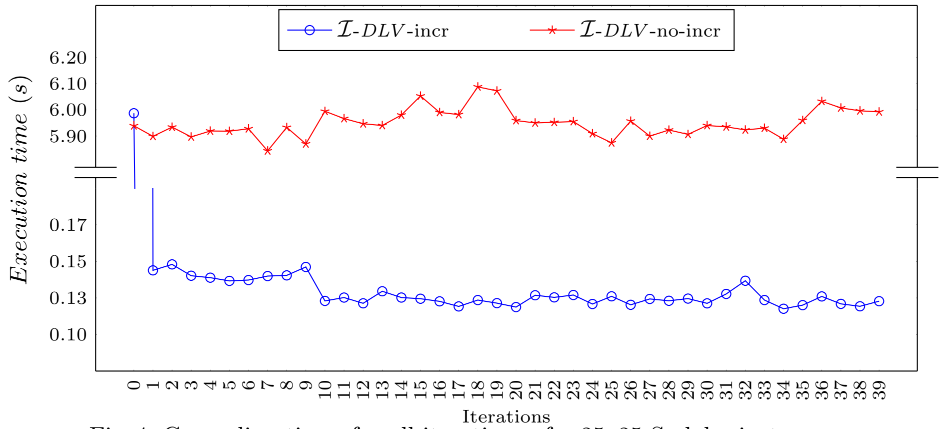

Fig. 4: Grounding times for all iterations of a 25x25 Sudoku instance.

6 Conclusions

In this paper we reported about our ongoing work towards the development of an incremental solver with caching of ground programs. Our technique is similar in spirit to the iClingo system [13]; this latter is also built in a multi-shot context, and allows to manually define which parts of a knowledge base are to be considered volatile, and which parts can be preserved in subsequent reasoning "shots". In our framework caching is totally transparent to knowledge-base designers, thus preserving declarativity; the same caching technique can be easily generalized to other semantics for rule-based knowledge bases such as the well-found semantics.

Our early experiments show the potential of the approach: it must be noted that our caching strategy is remarkably simple, in that cached ground programs grow monotonically from an iteration to another, thus becoming progressively larger but more generally applicable to a wider class of set input facts. We thus expect an exponential decrease in grounding times and an exponential decrease in the number of newly added rules in later iterations, as it is confirmed by our first experiments. The impact of larger ground instances on model generators is yet to be assessed, although we expect an acceptable performance loss.

Our work is currently being extended towards better defining the theoretical foundations showing the classes of programs and the conditions over which "over-grounding" is possible; also, we are interested in "interruptibility" of reasoning tasks, a context in which it is desirable to not discard parts of computed ground programs. As future work, we plan to experiment with further benchmark domains and with scenarios in which input information can be retracted. We plan also to investigate: the possibility of discarding rules when a memory limit is required; the impact of updates (i.e., additions/deletions of rules) in selected parts of the logic program; the introduction of ground programs which keep the properties of embeddings, yet allowing some form of simplification policies.

References

1. Alviano, M., Faber, W., Greco, G., Leone, N.: Magic sets for disjunctive datalog programs. Artificial Intelligence **187**, 156–192 (2012)
2. Beck, H., Eiter, T., Folie, C.: Ticker: A system for incremental asp-based stream reasoning. Theory and Practice of Logic Programming **17**(5-6), 744–763 (2017)
3. Brewka, G., Ellmauthaler, S., Gonçalves, R., Knorr, M., Leite, J., Pührer, J.: Reactive multi-context systems: Heterogeneous reasoning in dynamic environments. Artificial Intelligence **256**, 68–104 (2018)
4. Calimeri, F., Cozza, S., Ianni, G., Leone, N.: Computable functions in ASP: theory and implementation. In: International Conference on Logic Programming. Lecture Notes in Computer Science, vol. 5366, pp. 407–424. Springer (2008)
5. Calimeri, F., Fuscà, D., Perri, S., Zangari, J.: I-DLV: the new intelligent grounder of DLV. Intelligenza Artificiale **11**(1), 5–20 (2017)
6. Calimeri, F., Ianni, G., Perri, S., Zangari, J.: The eternal battle between determinism and nondeterminism: preliminary studies in the sudoku domain. Proocedings of Workshop on Knowledge Representation and Automated Reasoning (2013)
7. Calimeri, F., Perri, S., Zangari, J.: Optimizing answer set computation via heuristic-based decomposition. Theory and Practice of Logic Programming p. 1–26 (2019)
8. Dao-Tran, M., Eiter, T., Fink, M., Weidinger, G., Weinzierl, A.: Omiga : An open minded grounding on-the-fly answer set solver. In: European Conference on Logics in Artificial Intelligence. Lecture Notes in Computer Science, vol. 7519, pp. 480–483. Springer (2012)
9. Doyle, J.: A truth maintenance system. Artificial Intelligence **12**(3), 231–272 (1979)
10. Eiter, T., Ianni, G., Krennwallner, T.: Answer Set Programming: A Primer, pp. 40–110. Springer Berlin Heidelberg, Berlin, Heidelberg (2009)
11. Faber, W., Leone, N., Pfeifer, G.: Recursive aggregates in disjunctive logic programs: Semantics and complexity. In: European Conference on Logics in Artificial Intelligence. Lecture Notes in Computer Science, vol. 3229, pp. 200–212. Springer (2004)
12. Faber, W., Leone, N., Ricca, F.: Answer set programming. In: Wah, B.W. (ed.) Wiley Encyclopedia of Computer Science and Engineering. John Wiley & Sons, Inc. (2008)
13. Gebser, M., Kaminski, R., Kaufmann, B., Ostrowski, M., Schaub, T., Thiele, S.: Engineering an incremental ASP solver. In: International Conference on Logic Programming. Lecture Notes in Computer Science, vol. 5366, pp. 190–205. Springer (2008)
14. Gebser, M., Kaminski, R., Kaufmann, B., Schaub, T.: Multi-shot ASP solving with clingo. Theory and Practice of Logic Programming **19**(1), 27–82 (2019)
15. Gebser, M., Kaminski, R., König, A., Schaub, T.: Advances in *gringo* series 3. In: International Conference on Logic Programming and Nonmonotonic Reasoning. Lecture Notes in Computer Science, vol. 6645, pp. 345–351. Springer (2011)
16. Gebser, M., Maratea, M., Ricca, F.: The sixth answer set programming competition. Journal of Artificial Intelligence Research **60**, 41–95 (2017)
17. Gelfond, M., Lifschitz, V.: Classical negation in logic programs and disjunctive databases. New Generation Comput. **9**(3/4), 365–386 (1991)
18. Herbrand, J.: Recherches sur la théorie de la démonstration (1930)
19. Lefèvre, C., Béatrix, C., Stéphan, I., Garcia, L.: Asperix, a first-order forward chaining approach for answer set computing. Theory and Practice of Logic Programming **17**(3), 266–310 (2017)

20. Motik, B., Nenov, Y., Piro, R., Horrocks, I.: Maintenance of datalog materialisations revisited. Artificial Intelligence **269**, 76–136 (2019)
21. Nethercote, N., Stuckey, P.J., Becket, R., Brand, S., Duck, G.J., Tack, G.: Minizinc: Towards a standard CP modelling language. In: Principles and Practice of Constraint Programming, Proceedings. pp. 529–543 (2007)
22. Palù, A.D., Dovier, A., Pontelli, E., Rossi, G.: GASP: answer set programming with lazy grounding. Fundamenta Informaticae **96**(3), 297–322 (2009)
23. Van Gelder, A., Ross, K.A., Schlipf, J.S.: The Well-Founded Semantics for General Logic Programs. Journal of the ACM **38**(3), 620–650 (1991)
24. Weinzierl, A.: Blending lazy-grounding and CDNL search for answer-set solving. In: International Conference on Logic Programming and Nonmonotonic Reasoning. Lecture Notes in Computer Science, vol. 10377, pp. 191–204. Springer (2017)