TPLP **19** (5–6): 957–973, 2019. © Cambridge University Press 2019
doi:10.1017/S1471068419000292

Incremental Answer Set Programming with Overgrounding

FRANCESCO CALIMERI, GIOVAMBATTISTA IANNI, FRANCESCO PACENZA, SIMONA PERRI and JESSICA ZANGARI
Department of Mathematics and Computer Science, University of Calabria, Italy
(*e-mail:* `lastname@mat.unical.it`)-*https://www.mat.unical.it*

submitted 30 July 2019; accepted 31 July 2019

Abstract

Repeated executions of reasoning tasks for varying inputs are necessary in many applicative settings, such as stream reasoning. In this context, we propose an incremental grounding approach for the answer set semantics. We focus on the possibility of generating incrementally larger ground logic programs equivalent to a given non-ground one; so called *overgrounded programs* can be reused in combination with deliberately many different sets of inputs. Updating overgrounded programs requires a small effort, thus making the instantiation of logic programs considerably faster when grounding is repeated on a series of inputs similar to each other. Notably, the proposed approach works "under the hood", relieving designers of logic programs from controlling technical aspects of grounding engines and answer set systems. In this work we present the theoretical basis of the proposed incremental grounding technique, we illustrate the consequent repeated evaluation strategy and report about our experiments.

KEYWORDS: Knowledge Representation; Answer Set Programming; Stream Reasoning; Grounding; Instantiation of Logic Programs; Overgrounding

1 Introduction

The practice of attributing meaning to quantified logical sentences relying on equivalent propositional versions thereof dates back to the historical work of Jacques Herbrand (Herbrand 1930). Later, at the end of the past century, in the trail of such practice, *ground programs* have been used as the operational basis for computing the semantics of logic programs in the context of the well-founded semantics (Van Gelder et al. 1991) and of the answer set semantics (Gelfond and Lifschitz 1991; Eiter et al. 2009). Indeed, the typical structure of an Answer Set Programming (ASP) system implements its semantics by relying on a grounding module that takes in input a non-ground logic program P and produces an equivalent propositional theory $gr(P)$; the grounder is coupled with a subsequent solver module that applies proper resolution techniques on $gr(P)$ for computing the actual semantics in form of *answer sets* (Kaufmann et al. 2016). Steps similar to grounding are also taken when high-level input specifications are transformed in low-level constraint sets, or propositional SAT theories, such as when elaborating MiniZinc scripts (Nethercote et al. 2007).

The cost of generating a propositional theory can be significant both in terms of time and space; that is why the grounding phase cannot be overlooked just as a preprocessing

https://doi.org/10.1017/S1471068419000292 Published online by Cambridge University Press

stage. There are a large number of both benchmark and practical application settings in which the grounding step is prominent in terms of used resources (Gebser et al. 2017). Note also that, as soon as variable programs are given in input, the produced ground program is potentially of exponential size with respect to the input program[1]. As the popularity of ASP and its use increased, the scientific community worked on both theoretical and technical sides and produced a number of optimization techniques for reducing space and time costs of the grounding step (Calimeri et al. 2008; Gebser et al. 2011; Alviano et al. 2012; Calimeri et al. 2017), or to blend it within the answer set search phase to some extent (Dal Palù et al. 2009; Dao-Tran et al. 2012; De Cat et al. 2012; Weinzierl 2017; Lefèvre et al. 2017).

It must be observed that some contexts require to reason over data streams or via multiple, subsequent "shots" (Beck et al. 2017; Gebser et al. 2019); in such cases, where input data changes with time, instantiation can be quite a big time bottleneck, as not only it must be repeatedly computed on slightly different and dynamically changing input data, but also only short time windows are allowed between a "shot" and another. For instance, when dealing with real-time videogames, artificial players are allowed a very limited time per each decision. In competitions like the known GVGAI (General Video Game AI) (Pérez-Liébana et al. 2016), this limit is just 40 milliseconds, which is quite challenging even for state-of-the-art answer set systems. In such cases, the solving step is usually easy from the computational point of view; this makes the grounding step prominent in terms of optimization requirements.

A remarkable contribution has been introduced with the *iclingo* and *oclingo* systems (Gebser et al. 2011; Gebser et al. 2019). In *oclingo*, later generalized and integrated in latest version of the monolithic *clingo* system, a designer can procedurally control which parts of a logic program must be kept constant between two consecutive shots, thus allowing the caching of ground subprograms and of partial answer sets. Also, it is possible to control which parts of a program are subject to incremental evaluation with respect to an iteratively increasing integer parameter.Nevertheless, procedural controllability is not desirable in many development scenarios, especially because it requires a non-negligible knowledge of solver-specific internal algorithms. Let us cite, again, the videogame industry; while developing, designers look for easy and off-the-shelf solutions, and often do not have knowledge of declarative logic programming at all. Among other notable works, the Ticker incremental evaluation system (Beck et al. 2017) implements LARS, a stream reasoning formal framework with ASP-like semantics. LARS allows omni-comprehensive, yet demanding, temporal data management features which makes implementation more difficult and complex.

In this work, we focus on the usage of the pure ASP semantics in a repeated evaluation setting. In particular, we aim at closing the abovementioned gaps in the current state of the art by investigating the possibility of generating incrementally larger ground programs for a fixed non-ground logic program. These ground programs can be reused in combination with deliberately many different sets of inputs; also the knowledge designer is relieved from the burden of manually controlling the computational procedure.

[1] Strictly speaking, if variable arity and variable rule/program lengths are allowed, the program complexity of the decisional problem associated to the grounding of Datalog programs is complete for EXPTIME (Th. 4.5 of (Dantsin et al. 2001)).

https://doi.org/10.1017/S1471068419000292 Published online by Cambridge University Press

The contributions of this paper are summarized as follows. (i): We characterize a class of ground logic programs equivalent to the theoretical instantiation, called *embeddings* or *embedding programs*; they allow us to introduce a model-theoretic-like notion of ground programs with some desirable properties, and make dealing with formal properties of ground programs cleaner. *Overgrounded programs* are series of embedding programs growing monotonically that have both theoretical and practical impact: for instance, overgrounded programs can be easily generalized to other semantics for logic programming, such as the well-founded semantics; moreover, their incremental growth allows for easily implementing caching policies in many practical scenarios. (ii): We propose an incremental grounding strategy, allowing to reuse previously grounded programs in consecutive evaluation shots. In particular, while dealing with a specific reasoning task, we maintain a stored ground logic program which grows monotonically from one shot to another; such an *overgrounded program* becomes more and more general while moving from a shot to the next, increasingly adding potentially useful rules. Importantly, intermediate subsequent updates to the ground program are considerably less time-consuming: our technique allows, in a sense, to trade memory for time. (iii): We report about the experimental activities we conducted, aimed at assessing the practical impact of our approach; results show that it pays off in terms of performance, by knocking down grounding times and keeping solving times within more than reasonable bounds. We expect this setting to be particularly favourable when non-ground input programs may contain grounding-intensive rules, as for instance, in the case of declaratively programmed robots or videogame agents. (iv): We relieve designers of logic programs from two specific burdens. First, there is no need for procedural control over the incremental evaluation process, as features of the herein proposed incremental framework are almost transparent to designers. Second, there is no need to worry about which parts of logic programs might be more grounding-intensive; also highly general code, even non-optimized, can benefit from overgrounding. This allows to focus on representing knowledge in a single-encoding/multiple-inputs setting, thus preserving some desirable properties of knowledge representation and reasoning via ASP, namely declarativity and easiness of modelling.

In the following, after overviewing our approach and briefly presenting some preliminaries, we illustrate the theoretical basis our grounding strategy relies on. We then illustrate our framework and report about our experiments; we eventually discuss related work before drawing some final considerations. For space reasons, proofs are reported in the supplementary material attached to this paper at the TPLP archives.

2 Overgrounding: an overview

A canonical ASP system works by first *instantiating* a non-ground logic program P over input facts F, obtaining a propositional logic program $gr(P,F)$, and then computing the corresponding intended models, i.e., the answer sets $AS(gr(P,F))$. Importantly, systems build $gr(P,F)$ as a significantly smaller and refined version of the theoretical instantiation, defined via the Herbrand base (see, e.g., (Calimeri et al. 2008)), but preserve semantics, i.e., $AS(gr(P,F)) = AS(P \cup F)$. The choice of the instantiation function gr impacts on both computing time and on the size of the obtained instantiated program. gr usually maintains a set PT of "possibly true" atoms, initialized as $PT = F$; then,

https://doi.org/10.1017/S1471068419000292 Published online by Cambridge University Press

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[2].

Example 2.1
We show the overgrounding approach with a simple example. Let us consider the program P_0:

$$r(X,Y) \ :-\ e(X,Y),\ not\ ab(X). \qquad r(X,Z) \mid s(X,Z) \ :-\ e(X,Y),\ r(Y,Z).$$

When taking the set of facts $F_1 = \{e(c,a),\ e(a,b)\}$ into account, there are several ways for building a tailored instantiation of $P_0 \cup F_1$. For instance, one can simply assume F_1 as the initial set of "possibly true" facts, then generate new rules and new possibly true facts by iterating through positive head-body dependencies, obtaining the ground program G_1:

$$r_1 : r(a,b) \ :-\ e(a,b),\ not\ ab(a). \qquad r_2 : r(c,b) \mid s(c,b) \ :-\ e(c,a),\ r(a,b).$$
$$r_3 : r(c,a) \ :-\ e(c,a),\ not\ ab(c).$$

A more "aggressive" grounding strategy could also cut or simplify rules. For instance, one can use a simplification strategy which eliminates negative literals that are identified as definitely true. In the case above, it is easy to see that $ab(a)$ and $ab(c)$ have no chance of being true; hence, removing *not* $ab(a)$ and *not* $ab(c)$ from the rule bodies leads to the generation of G'_1:

$$r(a,b) \ :-\ e(a,b). \qquad r(c,b) \mid s(c,b) \ :-\ e(c,a),\ r(a,b).$$
$$r(c,a) \ :-\ e(c,a).$$

Nevertheless, G'_1 can be seen as less general, less re-usable than G_1, as it cannot be easily extended to a program which is equivalent to P_0 with respect to larger classes of input facts. Let us assume that, at some point, a subsequent run requires P_0 to be evaluated over a different set of input facts $F_2 = \{e(c,a),\ e(a,d),\ ab(c)\}$. Note that, with respect to F_1, F_2 features the additions $F^+ = \{e(a,d), ab(c)\}$ and the deletions $F^- = \{e(a,b)\}$. Nonetheless, differently from G'_1, G_1 can be easily incrementally updated by adding F^+ to the set of possibly true facts, yet preserving semantics. More precisely, one can ground P_0 with respect to the new set of possibly true facts $F_1 \cup F_2$; then, this new information can be propagated and only some additional rules $\Delta G_1 = \{r_4, r_5\}$, must be added, thus obtaining G_2:

$$r_1 : r(a,b) \ :-\ e(a,b),\ not\ ab(a). \qquad r_2 : r(c,b) \mid s(c,b) \ :-\ e(c,a),\ r(a,b).$$
$$r_3 : r(c,a) \ :-\ e(c,a), not\ ab(c). \qquad \mathbf{r_4 : r(c,d) \mid s(c,d) \ :-\ e(c,a),\ r(a,d).}$$
$$\mathbf{r_5 : r(a,d) \ :-\ e(a,d),\ not\ ab(a).}$$

We have now that G_2 is equivalent to P_0, both when evaluated over F_1 and when evaluated over F_2 as input facts, although only different portions of the whole set of rules in G_2 can be considered as relevant when considering F_1 or F_2 as inputs, respectively. On the other hand, G'_1 cannot be easily incremented with new rules so to be "compatible" with F_2, because the rule $r(c,a) \ :-\ e(c,a)$ would cause wrong inferences. Even more

[2] For an overview of grounding optimization techniques the reader can refer to (Gebser et al. 2011; Calimeri et al. 2017; Calimeri et al. 2019).

https://doi.org/10.1017/S1471068419000292 Published online by Cambridge University Press

interestingly, if a third *shot* of reasoning is requested over input facts $F_3 = \{e(a,d), e(c,a), e(a,b)\}$, we notice that G_2 does not require any further incremental update, as possibly true facts remain unchanged (i.e., $F_1 \cup F_2 = F_1 \cup F_2 \cup F_3$), i.e. the set ΔG_2 containing new incremental additions to G_2 would be empty.

It turns out that an instantiation strategy like the one producing G_1 and then G_2 should comply with specific properties which allow to define an incremental grounding strategy. Such properties are illustrated and used in the remainder of the paper.

3 Preliminaries

Throughout the paper we will assume to deal with programs under the answer set semantics. A program P is a set of rules. A rule r is of the form: $\alpha_1 \mid \alpha_2 \mid \ldots \mid \alpha_k$:- $\beta_1, \ldots, \beta_n, not\ \beta_{n+1}, \ldots, not\ \beta_m$, where $n, m, k \geqslant 0$. $\alpha_1, \ldots, \alpha_k$ and $\beta_1, \ldots, \beta_m$ are called *atoms*. An atom is of the form $p(\mathbf{X})$, for p a predicate name and $\mathbf{X}$ a list of variable names and constants. The *head* of r is defined as $H(r) = \{\alpha_1, \ldots, \alpha_k\}$; if $H(r) = \emptyset$ then r is said to be a *constraint*. The *positive body* of r is defined as $B^+(r) = \{\beta_1, \ldots, \beta_n\}$, while the *negative body* 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 $B(r) = \emptyset$ and $\|H(r)\| = 1$ then r is referred to as a *fact*. The set of all head atoms in P is denoted by $Heads(P) = \bigcup_{r \in P} H(r)$. As usual, we deal with "safe" logic programs, i.e., for any non-ground rule $r \in P$, and for any variable X appearing in R, there is at least an atom $p \in B^+(r)$ mentioning X.

A program (resp. a rule, a literal, a term) is said to be *ground* if it contains no variables. In the following, we will assume to deal with a fixed *Herbrand Universe* consisting of a finite set of constants U, and to deal with programs that do not contain facts, as these are separately given in form of sets of ground atoms: given a program P and a set of facts F, both P and F will only feature constant symbols appearing in U. The *Herbrand base* of a program P, denoted by B_P, is the set of all ground atoms obtainable from the atoms of P by replacing variables with elements from U. A *substitution* for a rule $r \in P$ is a mapping from the set of variables of r to the set U of ground terms. A *ground instance* of a rule r is obtained by applying a substitution to r.

Given a logic program P, the *theoretical instantiation (grounding)* of P, denoted as $grnd(P)$, is defined as the set of all ground instances of rules in P. We assume the reader is familiar with the notions of interpretation and model as subsets of B_P, and with the usual notation in the literature; in particular, when an interpretation I models an element e (resp. an atom, a body, a head, a rule) this is denoted by $I \models e$. Given P, and also a set of facts F, an answer set A of $P \cup F$ is a subset of B_P, defined as the minimal model of the so-called FLP reduct $(grnd(P) \cup F)^A$ of $grnd(P) \cup F$ (Faber et al. 2004). We denote the set of all answer sets of $P \cup F$ as $AS(P \cup F)$.

It is possible to compute a ground program equivalent to $grnd(P) \cup F$ by means of the operator defined next. In this section we will follow (Calimeri et al. 2008): note, however, that the instantiation operator defined here is slightly differently formalized, in that we purposely keep facts explicitly separated from rules.

Definition 3.1
[cf. (Calimeri et al. 2008)]. Given a program P and a set of ground atoms S, we define the operator $Inst(P, S)$ as $Inst(P, S) = \{r \in grnd(P)\ s.t.\ B^+(r) \subseteq S\}$. With a slight

https://doi.org/10.1017/S1471068419000292 Published online by Cambridge University Press

abuse of notation, for a set R of ground rules we define $Inst(P, R)$ as $Inst(P, Heads(R))$. $Inst(P, F)^k$ is defined as the k-th element of the sequence $Inst(P, F)^0 = Inst(P, \emptyset \cup F)$, $\ldots$, $Inst(P, F)^k = Inst(P, Inst(P, F)^{k-1} \cup F)$.

Intuitively, the notion above formalizes the idea of selecting, among all ground instances of rules in P, those *supported* by a given set S or by the heads of a given set of ground rules R. Given that the sequence above is defined by a monotonic operator it converges to its least fixpoint.

Proposition 3.1
[cf. (Calimeri et al. 2008)]. Given a program P and a set of facts F, the sequence $Inst(P, F)^k$, $k \geq 0$ converges to its least fixpoint $Inst(P, F)^\infty$.

Interestingly, given a program P and a set of facts F, the fixpoint above can be useful for computing a ground program that is equivalent to $grnd(P) \cup F$.

Theorem 3.1
[cf. (Calimeri et al. 2008)] For a program P and a set of facts F, $AS(P \cup F) = AS(Inst(P, F)^\infty \cup F)$.

The following theorem extends Theorem 3.1 of (Fages 1994) to the case of logic programs allowing disjunction in the heads; the proof descends from the notion of computation and Theorem 6.22 as given in (Leone et al. 1997); for space reasons, we refrain from reporting the full definition here and refer the reader to the cited work.

Theorem 3.2
For a given program P, a set of facts F and an answer set A for P, we can assign to each atom $a \in A$ an integer value $stage(a) = i$ so that $stage$ encodes a strict well-founded partial order over all atoms in A, in such a way that there exists a rule $r \in grnd(P) \cup F$ with (i): $a \in H(r)$, (ii): $A \models B(r)$ and (iii): for any atom $b \in B^+(r)$, $stage(b) < stage(a)$.

4 Embedding Programs

We introduce a declarative characterization of a class of equivalent ground programs, called *embeddings*. This notion is useful for proving, given a program P and a set of facts F, whether a ground program G having certain features is equivalent to $P \cup F$, i.e., G is a "correct grounding" that "embeds" $P \cup F$. In a sense, embeddings relate to partial instantiations generated by the $Inst$ operator, like Herbrand models relate to supported interpretations generated by the immediate consequence operator. An illustrative comparison between models and embedding programs is reported in Table 1.

Definition 4.1
[*Embeddings.*] For a program P, a set of facts F, a set of ground rules $R \subseteq (grnd(P) \cup F)$, and a rule $r \in (grnd(P) \cup F)$, we say that:

- R embeds the body of r, denoted $R \vdash_b r$, if $\forall a \in B^+(r)\ \exists r' \in R$ s.t. $a \in H(r')$;
- R embeds the head of r, denoted $R \vdash_h r$, if $r \in R$.
- R embeds r, denoted $R \vdash r$, if either: (i) $R \nvdash_b r$, or (ii) $R \vdash_h r$.

Given a logic program P and a set of input facts F, a set of ground rules $\mathscr{E} \subseteq grnd(P) \cup F$ is an *embedding program* for $P \cup F$, if $\forall r \in grnd(P) \cup F$, $\mathscr{E} \vdash r$.

https://doi.org/10.1017/S1471068419000292 Published online by Cambridge University Press

Table 1: Comparison of the classic model-theoretic semantics for logic programs and the embedding program semantics.

Let P be a program, F a set of facts, and $r \in grnd(P) \cup F$	
Model-Theoretic Semantics	***Embedding Programs Semantics***
I : Set of *atoms*	S : Set of *rules*
$I \models B(r)$, if $B^+(r) \subseteq I$ and $B^-(r) \cap \{\text{not } a \mid a \in I\} = \emptyset$	$S \vdash_b r$, if $\forall a \in B^+(r)\ \exists r' \in S$ s.t. $a \in H(r')$
$I \models H(r)$, if $H(r) \in I$	$S \vdash_h r$, if $r \in S$
$I \models r$, if either: (i) $I \not\models B(r)$, or (ii) $I \models H(r)$	$S \vdash r$, if either: (i) $S \not\vdash_b r$, or (ii) $S \vdash_h r$

Example 4.1
Let us consider the program P_0 of Example 2.1 along with the set of facts $F = \{e(a,b), e(b,b)\}$. The ground program below represents $F \cup grnd(P_0)$.

$r_1 : r(a,a)\ :-\ e(a,a),\ not\ ab(a).$
$r_2 : r(a,b)\ :-\ e(a,b),\ not\ ab(a).$
$r_3 : r(b,a)\ :-\ e(b,a),\ not\ ab(b).$
$r_4 : r(b,b)\ :-\ e(b,b),\ not\ ab(b).$
$r_5 : r(a,a) \mid s(a,a)\ :-\ e(a,a),\ r(a,a).$
$r_6 : r(a,a) \mid s(a,a)\ :-\ e(a,b),\ r(b,a).$
$r_7 : r(a,b) \mid s(a,b)\ :-\ e(a,a),\ r(a,b).$
$r_8 : r(a,b) \mid s(a,b)\ :-\ e(a,b),\ r(b,b).$
$r_9 : r(b,a) \mid s(b,a)\ :-\ e(b,a),\ r(a,a).$
$r_{10} : r(b,a) \mid s(b,a)\ :-\ e(b,b),\ r(b,a).$
$r_{11} : r(b,b) \mid s(b,b)\ :-\ e(b,a),\ r(a,b).$
$r_{12} : r(b,b) \mid s(b,b)\ :-\ e(b,b),\ r(b,b).$
$r_{13} : e(a,b).$
$r_{14} : e(b,b).$

By definition, $grnd(P_0) \cup F$ is clearly an embedding of $P_0 \cup F$. The set $E_1 = \{r_2, r_4, r_8, r_{11}, r_{12}, r_{13}, r_{14}\}$ is also an embedding of $grnd(P_0) \cup F$. Indeed, for every rule $r \in E_1$, $E_1 \vdash_h r$, and for every rule $r \in (grnd(P_0) \cup F) \setminus E_1$ it holds that $E_1 \not\vdash_b r$. The set $E_2 = \{r_4, r_8, r_{12}, r_{13}, r_{14}\}$ is not an embedding of $grnd(P_0) \cup F$; indeed, $E_2 \not\vdash r_2$ since $E_2 \vdash_b r_2$ and $E_2 \not\vdash_h r_2$.

Embedding programs enjoy a number of interesting properties, some of which are reported next. First, an embedding program is equivalent to $P \cup F$ (Theorem 4.1); also, an intersection of embedding programs is an embedding program, similarly to the intersection of models (Proposition 4.1).

Theorem 4.1
(Embedding equivalence). For a program P, a set of facts F and an embedding program $\mathscr{E}$ for $P \cup F$, $AS(grnd(P) \cup F) = AS(\mathscr{E})$.

Proposition 4.1
(Intersection of embedding programs). Given a logic program P, a set of facts F, $\mathscr{E}_1$ and $\mathscr{E}_2$ embedding programs for $P \cup F$, $\mathscr{E} = \mathscr{E}_1 \cap \mathscr{E}_2$ is an embedding program for $P \cup F$.

Example 4.2
Consider again the program P_0, the set of facts $F = \{e(a,b), e(b,b)\}$ and the embedding E_1 of the example above. It is easy to see that the set of rules $E_3 = \{r_2, r_4, r_7, r_8, r_{12}, r_{13}, r_{14}\}$ is also an embedding of $F \cup grnd(P_0)$, and that $E_4 = E_1 \cap E_3 = \{r_2, r_4, r_8, r_{12}, r_{13}, r_{14}\}$ is an embedding for $F \cup grnd(P_0)$ as well.

https://doi.org/10.1017/S1471068419000292 Published online by Cambridge University Press

Finally, the next theoretical results show that $P \cup F$ has a minimal embedding program, corresponding to the intersection of all embedding programs. Also, the minimal embedding program can be computed as the fixpoint of $Inst$, thus establishing a correspondence between embedding programs and the operational semantics of grounders.

Theorem 4.2
Given a program P and a set of facts F, $\mathscr{E} \subseteq grnd(P) \cup F$ is an embedding program for $P \cup F$ iff $\mathscr{E} \supseteq Inst(P, \mathscr{E}) \cup F$.

Theorem 4.3
Let $\mathscr{ES}$ be the set of embeddings of $P \cup F$; then,

$$Inst(P, F)^{\infty} \cup F = \bigcap_{\mathscr{E} \in \mathscr{ES}} \mathscr{E}.$$

Example 4.3
Note that $Inst(P_0, F)^{\infty} \cup F$ for the program P_0 of example 4.2 coincides with E_4. It can be verified that E_4 corresponds to the intersection of all embedding programs for $F \cup grnd(P_0)$. In order to grasp the intuition, one could note that rules r_{13} and r_{14} belong to all embeddings as they are facts; moreover, rule r_2, r_4, r_8 and r_{12} must belong to all embeddings as well, as their bodies are embedded by any set of rules containing F.

Theorem 4.4
Given a logic program P and a set of facts F, let $\mathscr{ES}$ be the set of embeddings of $P \cup F$. Then

$$AS(P \cup F) = AS\Big(\bigcap_{\mathscr{E} \in \mathscr{ES}} \mathscr{E}\Big).$$

Note that Theorem 4.3 combined with Theorem 4.1 constitutes an alternative, cleaner proof of Theorem 3.1.

5 Overgrounding

In the following we assume we are given a program P and a sequence of sets of facts F_1, ..., F_n; then, let us assume that we need to perform a series of distinct evaluations over a different F_i at each shot. In other words we aim at computing all the sets $AS(P \cup F_1)$, ..., $AS(P \cup F_n)$.

Definition 5.1
For an integer k, s.t. $1 \leq k \leq n$, we define $UF_k = \bigcup_{1 \leq i \leq k} F_i$ as the sets *accumulated facts* at shot k. Moreover, we define $G_k = Inst(P, UF_k)^{\infty}$ as the *overgrounded program* at shot k.

Each overgrounded program G_k is equivalent to $P \cup F_i$ for $1 \leq i \leq k$.

Theorem 5.1
The following two statements hold:
(1): $Inst(P, UF_{k-1})^{\infty} \subseteq Inst(P, UF_k)^{\infty}$; (2): $AS(Inst(P, UF_k)^{\infty} \cup F_i) = AS(P \cup F_i)$.

https://doi.org/10.1017/S1471068419000292 Published online by Cambridge University Press

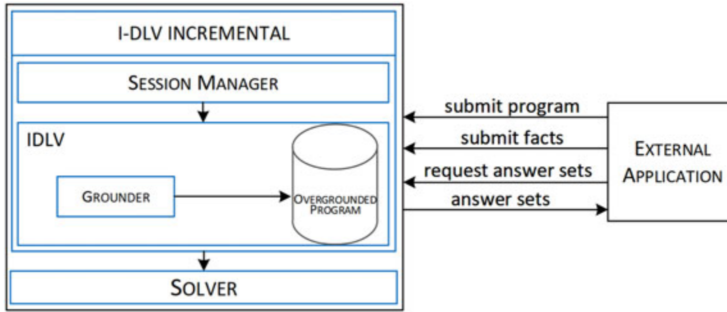

Figure 1: An infrastructure for incremental grounding.

We can now devise an incremental grounding strategy relying on the theoretical results illustrated so far. For the sake of simplicity, we illustrate the core of the idea and omit all technical details and optimizations about how to instantiate a program, referring the reader to the vast literature. At each shot k, we keep the set of accumulated facts UF_k and the overgrounded program G_k by incrementally updating UF_{k-1} and G_{k-1}. In this setting, $AS(P \cup F_k)$ can be obtained by computing $AS(G_k \cup F_k)$. More in detail, overgrounded programs are managed as follows:

- At shot 1, we set $UF_1 = F_1$, and $G_1 = Inst(P, UF_1)^\infty$
- At generic shot k:
    1. we set $UF_k = UF_{k-1} \cup F_k$,
    2. we compute a set of additional ground rules ΔG_k, and
    3. we set $G_k = G_{k-i} \cup \Delta G_k$.

The computation of ΔG_k can be efficiently performed by using an optimized incremental algorithm that takes in input the newly added facts $F_k \setminus UF_{k-1}$ and produces the new rules appearing in $G_k \setminus G_{k-1}$. In our case, we developed a variant of the typical incremental iteration of the semi-naive algorithm (Ullman 1988).

6 Prototype structure and experimental evaluation

The overgrounding strategy described above has been implemented by adapting the $\mathscr{I}$-*DLV* grounder (Calimeri et al. 2017; Calimeri et al. 2019); the architecture of the resulting prototype, called $\mathscr{I}$-*DLV*-incr, is depicted in Figure 1. The system behaves as a process staying alive and providing a service-oriented behaviour, waiting for requests. An external application EA can open a working session and specify tasks to be carried out; working sessions are handled by a SESSION MANAGER component. After submitting a load request for a logic program P along with an initial set of facts F_1, EA can ask to compute the answer sets of P over F_1: the GROUNDER component is in charge of producing and storing the overgrounded program $Inst(P, F_1)^\infty$; an external SOLVER system is adopted to compute the answer sets of $Inst(P, F_1)^\infty \cup F_1$. This process can be repeated/iterated: EA can provide additional sets of facts F_k for $2 \leq k \leq n$ so that $Inst(P, UF_k)^\infty$ with $UF_k = \bigcup_{1 \leq i \leq k} F_i$ is incrementally produced and stored. At each step k, 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 $i < k$. Again, the solver can be invoked to compute the answer sets of $Inst(P, UF_k)^\infty \cup F_k$.

https://doi.org/10.1017/S1471068419000292 Published online by Cambridge University Press

The overgrounding approach peculiarly trades off memory for computation time spent during the grounding stage. However, one might wonder about several questions: how big is the expected memory overhead in real scenarios, so to show in which contexts the increased memory demand can be considered acceptable; whether and how much grounding times decrease when overgrounding become increasingly larger, and thus rich of re-usable rules; whether and how much model generation times increase due to increasingly larger inputs. Indeed, it is reasonable to expect performance worsening in the model generation phase, because of the larger number of non-simplified rules to process.

In order to assess the above issues, we conducted an experimental evaluation, considering two specific benchmarks taken from two real world settings (Calimeri et al. 2013; Calimeri et al. 2018) with different specific features: Pac-Man and Sudoku. The two benchmarks constitute good and generalizable real cases of incremental scenarios: the Sudoku domain allows to perform a stress-test of the approach on grounding-intensive tasks; the Pacman game allows to assess effectiveness of overgrounding for real-time reasoning without the need for manual and involved customizations. Experiments on Sudoku have been performed on a NUMA machine equipped with two 2.8GHz AMD Opteron 6320 processors and 128GB of RAM, while experiments on Pac-Man have been performed on a machine equipped with a 2.2GHz Intel Xeon Processor E5-2650 v4 and 64GB of RAM.

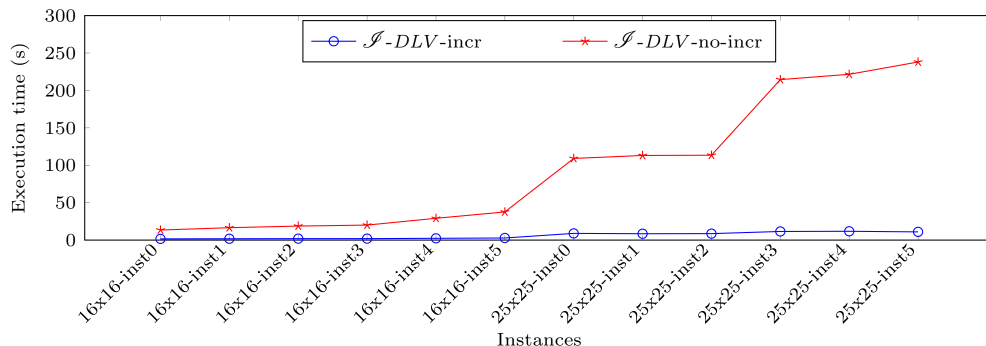

Figure 2: Experiments on Sudoku benchmarks.

6.1 Multi-shot Sudoku

The classic Sudoku puzzle, or simply “Sudoku”, consists of a tableau featuring 81 cells, or positions, arranged in a 9 by 9 grid. When solving a Sudoku, players typically adopt deterministic inference strategies allowing, possibly, to obtain a solution. Several deterministic strategies are known (Calimeri et al. 2013) and can be encoded in ASP; herein, we took into account two simple strategies, namely, “naked single” and “hidden single”.

The encoding of Sudoku deterministic inference rules is generally grounding-intensive, like in the following code fragment, encoding the naked single strategy:

```
candidatesAreMoreThan2(X,Y):-candidate(X,Y,N),candidate(X,Y,N1),N!=N1.
newValue(X,Y,N):-candidate(X,Y,N),not candidatesAreMoreThan2(X,Y),nogiven(X,Y).
```

where an atom *candidate(X,Y,N)* specifies that number N can be possibly assigned to the cell (X, Y), and *nogiven(X,Y)* tells that the value of the cell (X, Y) has not been inferred yet.

https://doi.org/10.1017/S1471068419000292 Published online by Cambridge University Press

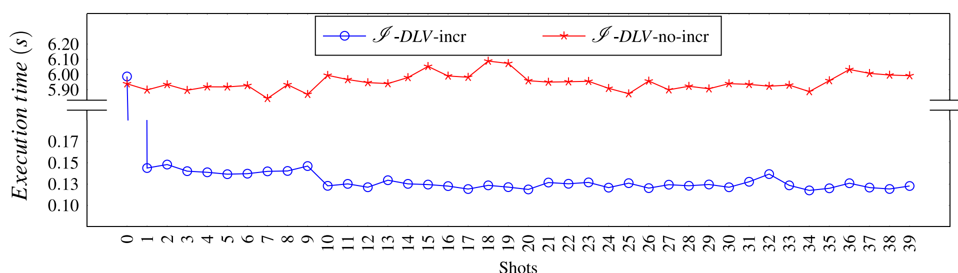

Figure 3: Grounding times for all iterations of a 25x25 Sudoku instance.

In the experiments we considered generalized Sudoku tables of size 16x16 and 25x25, and tested logic programs under answer set semantics encoding deterministic inference rules. We compared two different evaluation strategies: (*i*) $\mathscr{I}$-*DLV*-incr implementing the incremental approach, and (*ii*) $\mathscr{I}$-*DLV*-no-incr in which no incremental evaluation policy is applied. Both strategies have been executed in an online setting, in which consecutive series of input facts are submitted. For a given Sudoku table, the two inference rules above were modelled via ASP logic programs (see (Calimeri et al. 2013)). 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 then given as input to the system and, again, by means of the same inferences, new cell values are possibly 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; however, for each considered Sudoku size, we selected only instances which are completely solvable with the two inference rules described above.

Grounding times of $\mathscr{I}$-*DLV*-incr and $\mathscr{I}$-*DLV*-no-incr are plotted in Figure 2: for each instance, the total grounding time (in seconds) computed over all iterations is reported. $\mathscr{I}$-*DLV*-incr required at most 12 seconds to iteratively solve each instance, and performed clearly better than $\mathscr{I}$-*DLV*-no-incr that required up to 237 seconds, instead; this results in an improvement of 95%. Figure 3 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. At the first iteration, both configurations spent almost the same time; for each further iteration, though, $\mathscr{I}$-*DLV*-incr required an average time of 0.13 seconds: a time reduction of 98% w.r.t. $\mathscr{I}$-*DLV*-no-incr, that showed an average time of about 5.95 seconds. On the overall, this confirms the potential of our incremental grounding approach in scenarios involving updates in the underlying input facts.

6.2 Multi-shot Pac-Man

In our work (Calimeri et al. 2018) we show how to integrate an ASP-based reasoning module in the Unity game development framework, and showcase an artificial player for the classic real-time game Pac-Man. The Pac-Man moves in a board containing a number of pellets and four ghosts chasing him: the goal is to eat all pellets while avoiding the four ghosts. The Pac-Man must continuously decide the way to take, depending on "dynamic" (i.e., changing at each reasoning shot) facts representing the current status of the board (empty/pellet tiles, positions of the four ghosts). The new direction of

https://doi.org/10.1017/S1471068419000292 Published online by Cambridge University Press

the Pac-Man depends on which logical assertion among *next(up)*, *next(down)*, *next(left)*, *next(right)* belongs in the "best" guessed answer set. The intelligence of the Pac-Man is encoded via a logic program P_{pac}, that must be repeatedly executed; this requires multiple grounding+solving jobs over slightly different and unforeseen inputs.

For space reasons, we focus next only on parts of P_{pac} requiring a significant effort at the grounding stage. Instances of the *tile* predicate encode the game board divided into tiles; the fact *pacman(x,y)* represents the current position of the Pac-Man, while *ghost(x,y,g)* represents the position of ghost g; atoms of the form *nextTile(X, Y)* encode the possible next positions of the Pac-Man. The strategy adopted by P_{pac} is quite simple: priority is to get away from ghosts. To this end, distances between the next position candidates and all ghosts are computed: the next position chosen is among the ones increasing the distance to the nearest ghost. This behaviour is achieved via an ASP code fragment similar to the following:

```
nextTile(X,Y) :- pacman(Px,Y), next(right), X=Px+1, tile(X,Y).
nextTile(X,Y) :- pacman(Px,Y), next(left), X=Px-1, tile(X,Y).
nextTile(X,Y) :- pacman(X,Py), next(up), Y=Py+1, tile(X,Y).
nextTile(X,Y) :- pacman(X,Py), next(down), Y=Py-1, tile(X,Y).
adjacent(X1,Y1,X2,Y2) :- tile(X1,Y1), tile(X2,Y2), step(DX,DY), X2=X1+DX, Y2=Y1+DY.
adjacent(X1,Y1,X2,Y2) :- tile(X1,Y1), tile(X2,Y2), step(DX,DY), X2=X1-DX, Y2=Y1-DY.
distance(X1,Y1,X2,Y2,1) :- tile(X1,Y1), adjacent(X1,Y1,X2,Y2).
distance(X1,Y1,X3,Y3,D) :- number(D), distance(X1,Y1,X2,Y2,D-1), adjacent(X2,Y2,X3,Y3).
distPacmanGhost(D,G) :- nextTile(Xp,Yp), ghost(Xg,Yg,G), minDistance(Xp,Yp,Xg,Yg,D).
noMinDistPacmanGhost(X) :- distPacmanGhost(X,_), distPacmanGhost(Y,_), distance(X), X>Y.
minDistancePacmanNextGhost(MD) :- not noMinDistPacmanGhost(MD), distPacmanGhost(MD,_).
```

The above code contains parts which will likely have the same instantiation regardless of input facts (e.g., the *adjacent* and *distance* predicates), and parts whose instantiation will slightly differ depending on input facts (i.e., *nextTile* and *distPacmanGhost*, which depend on current positions of Pac-Man and ghosts). Such two categories of code fragments would roughly correspond to parts respectively marked with the former `#base` and `#cumulative` directives of iclingo (Gebser et al. 2019; Gebser et al. 2011), and later generalized with the `#program` keyword of clingo; however, it is not necessary to introduce specific grounding directives within ASP code while using $\mathscr{I}$-*DLV*-incr. In (Calimeri et al. 2018), in the absence of an overgrounding engine, we manually tabled the instantiation of the logic program above, we resorted to procedural code for many aspects of the reasoning process, and we limited the maximum visibility horizon of the Pac-Man to 10 tiles. By adopting the overgrounding approach, we were able to avoid manual optimizations and achieved a fully automatic incremental approach: the grounded program is internally stored right after the first computation, thus bypassing re-computations of heavy grounding tasks.

It is worth noting that the *adjacent* and *distance* predicates are defined in a general but inefficient way; this can likely be the case if such a predicate was taken from a predicate module library. Alternative definitions, improving grounding times, would be in principle possible: for instance, the distance between tiles is encoded with the predicate $distance(X_1, Y_1, X_2, Y_2, D)$, where D is computed for all couple of points $(X_1, Y_1) \times (X_2, Y_2)$; this is what one can expect from a modeller not knowing, and probably not wishing to know how to optimize this code. Pushing *nextTile* atoms within rule bodies would allow to reduce the grounding size by limiting grounding only to actually necessary distance values. Nonetheless, besides making code less readable and less declarative, this

https://doi.org/10.1017/S1471068419000292 Published online by Cambridge University Press

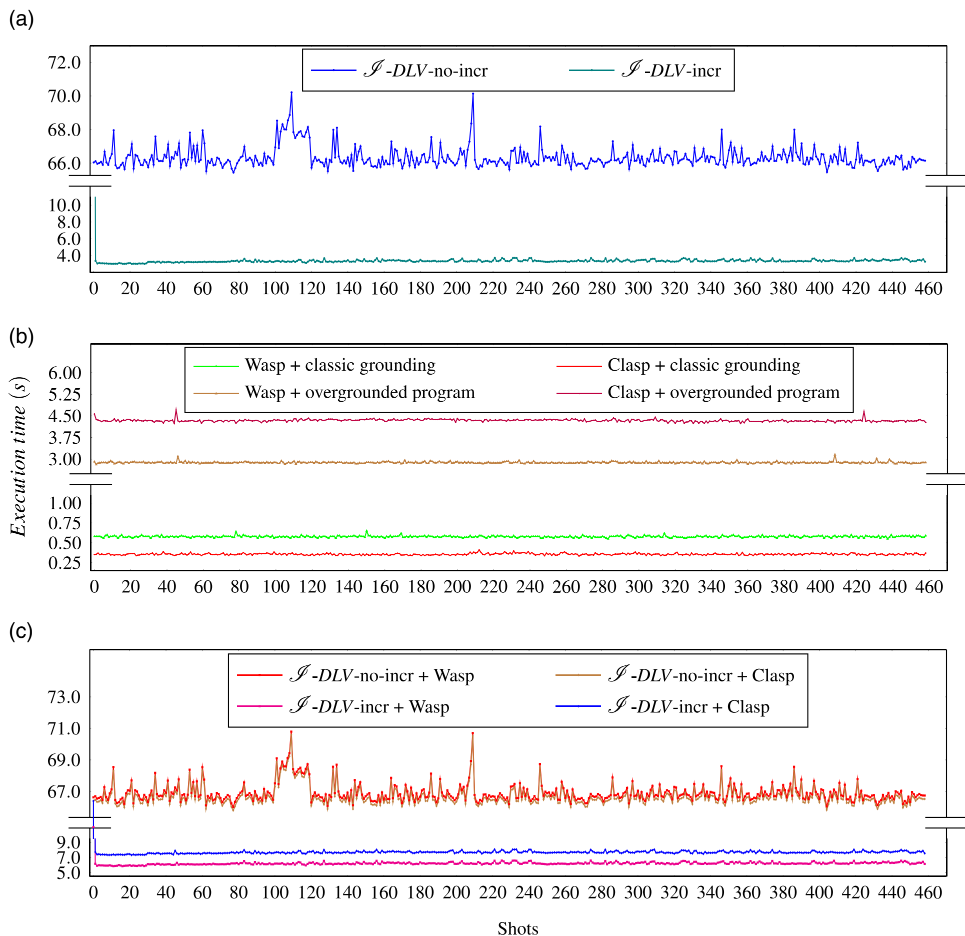

Figure 4: Pac-Man benchmark: **(a)** Grounding time. **(b)** Solving Time. **(c)** Total time.

modification would disrupt modularity, and it requires some expertise on operational details of grounders.

Experiments in the Pac-Man domain were conducted by logging a series of 459 consecutive steps taken during an actual game; each step encodes a different status of the game board in terms of logical assertions. Such inputs were, in turn, run along with P_{pac} in a controlled environment outside the game engine, averaging times over five separate runs. Grounding was performed by our new overgrounding engine $\mathscr{I}$-*DLV*-incr and by the non-incremental version $\mathscr{I}$-*DLV*-no-incr. Even though $\mathscr{I}$-*DLV*-no-incr re-computes ground programs per each shot, instantiations are generally smaller and better tailored to the current shot; instead, $\mathscr{I}$-*DLV*-incr re-uses overgrounded programs, which are generally larger.

The solving task was instead performed using WASP (Alviano et al. 2015) and clasp (Gebser et al. 2015). In order to assess performance in the worst case scenario, we allowed the Pac-Man to have a visibility horizon of 30 tiles in each direction.

Results are depicted in Figure 4 and Figure 5; both show results on the X axis in temporal execution order. Figure 4 *(a)* compares instantiation times of both grounders,

https://doi.org/10.1017/S1471068419000292 Published online by Cambridge University Press

while figure 4 *(b)* reports solver execution times when either an overgrounded program or a simplified instantiation is fed in input. Both figures show that the overgrounding engine has similar performance to its non-incremental counterpart on the first iteration, while grounding times are considerably smaller for all subsequent iterations. On the other hand, an acceptable worsening in solving times can be observed when overgrounded programs are fed in input. Figure 4 *(c)* reports total execution times for all the four possible combinations grounder+solver. All in all, we can conclude that the worse performance in solving times is widely absorbed by the time savings due to the use of incremental grounding.

As shown in Figure 5 *(a)*, the overgrounding engine stores progressively more rules. The impact of larger inputs for solvers is almost constant, while incremental grounding times have a slight increase on later iterations. We can notice that the number of added rules per each iteration follows a steep initial curve and almost stabilizes in later iterations. Nonetheless, almost all useful rules are added in the first iteration. Figure 5 *(b)* reports the recorded peak memory for the overgrounding engine when run with an horizon of $10, 20$ and 30 tiles respectively.

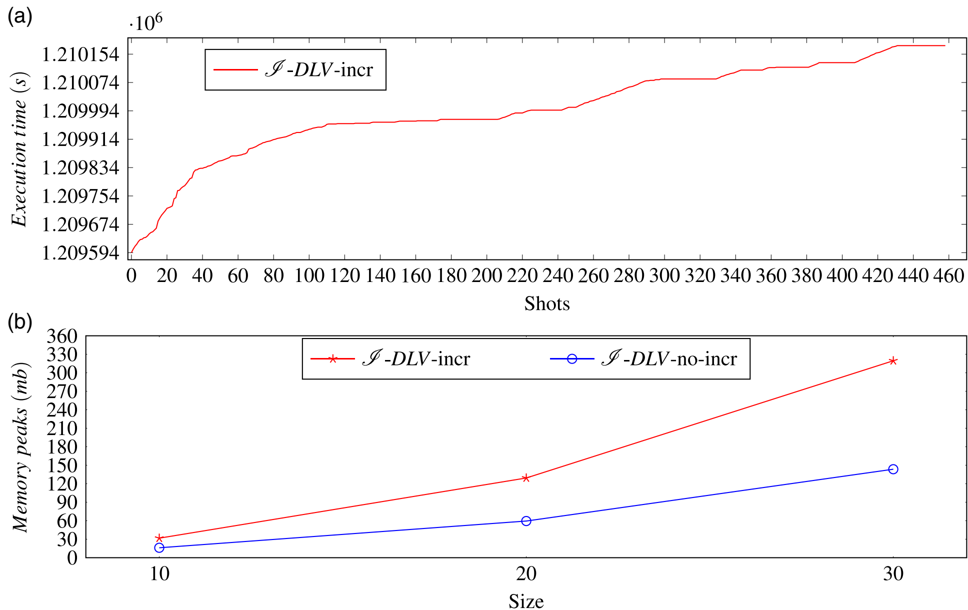

Figure 5: Pac-Man benchmark: **(a)** total number of rules after each iteration. **(b)** Memory peaks.

7 Related Work

Although we focus more on time requirements rather than on dealing with huge amounts of streamed data, our proposal is comparable to early and recent work on incremental update of Datalog materializations (Motik et al. 2019). In such work, *incremental maintainance* refers to a setting in which a pool of views (i.e. a logic program/knowledge base) is run repeatedly over a changing database. The views are updated from a run to another, and both insertion or deletion of tuples in input data trigger update activities on views.

https://doi.org/10.1017/S1471068419000292 Published online by Cambridge University Press

In our paper we adopt the same terminology, and we refer to a logic program P and to facts $F_1, \ldots, F_n$. $AS(P \cup F_1)$ can be, in principle, totally different from $AS(P \cup F_2)$. However, $AS(P \cup F_2)$ can be computed on $Inst(P, F_1 \cup F_2)^\infty$ which includes all the rules of $Inst(P, F_1)^\infty$, were this latter should be already pre-computed, thus reducing grounding times. However, the aforementioned approaches focus on query answering over stratified Datalog programs, and can just materialize query answers; our focus is on a generalized setting, where disjunction and unstratified negation are allowed and propositional logic programs are materialized and maintained. Furthermore, we purposely decided to not introduce deletion techniques when logic assertions are retracted between two consecutive runs: our overgrounded programs, while keeping semantic soundness, grow monotonically; yet, the absence of negative update activities allows to keep incremental grounding times significantly low.

Our work has also connections with the iclingo/oclingo approach (Gebser et al. 2011; Gebser et al. 2019) where incrementality is meant in a slightly different way: one has a base program P, which is coupled with additional module layers $M_1, \ldots, M_n$. An answer set A of $P \cup M_1$ can be "incremented" with new atoms so to build an answer set A' of $P \cup M_1 \cup M_2$. In other words, according to the iclingo/oclingo philosophy, one has to model her/his problem by thinking in terms of "layers" of modules.

To date, incremental solving is (still) an almost unexplored topic. By full incremental solving, we mean the widest general settings, in which both P and F are subject to unexpected changes. This research line would require to go beyond both our overgrounding-based and iclingo/oclingo approaches, by removing annotations and constraints in modelling, and introducing seamless updates of answer sets without restarting solving and/or grounding. The Ticker system (Beck et al. 2017) can be seen as a significant effort in this direction as it implements the LARS stream reasoning formal framework by using truth maintenance techniques, under ASP semantics.

In our work, we left full incremental reasoning out of the table, as the focus of the paper is on grounding-intensive repeated tasks. We note that such tasks are not necessarily polynomial, as unstratification and disjunction is possible.

Furthermore, it is worth noting that overgrounding is essentially orthogonal to recent advances in lazy grounding (Dal Palù et al. 2009; Lefèvre et al. 2017; Bogaerts and Weinzierl 2018); indeed, these essentially aim at blending grounding tasks within the solving step for reducing memory consumption; rather, our focus is on making grounding times negligible on repeated evaluations by explicitly allowing the usage of more memory, yet still keeping the two evaluation steps separated.

8 Conclusions

In this work we reported about theoretical properties of embedded and overgrounded programs, and the consequent development of an incremental grounder with permanent in-memory storage of ground programs. Experiments show the potential of the approach; it must be noted that our ground program caching strategy is remarkably simple: cached ground programs grow monotonically between consecutive shots, thus becoming progressively larger but more generally applicable to a wider class of sets of input facts. We measured a considerable decrease in grounding times and in the number of newly added rules in later shots; interestingly, the impact of larger ground instances on model gener-

https://doi.org/10.1017/S1471068419000292 Published online by Cambridge University Press

ators is fair. All in all, the results of benchmarks regarding memory footprints and the time performance clearly qualify the overgrounding approach for applications in which speed requirements are very tight and can be reached affording more memory.

The simplicity of the overgrounding approach paves the way to several extensions. We are currently investigating: 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. Further experiments, benchmark encodings and the binary of $\mathscr{I}$-*DLV*-incr are available at https://www.mat.unical.it/calimeri/material/iclp2019.

Acknowledgements

This work has been partially supported by the Italian MIUR Ministry and the Presidency of the Council of Ministers under project "Declarative Reasoning over Streams" under the "PRIN" 2017 call (CUP $H24I17000080001$, project 2017M9C25L_001), by the Italian MISE Ministry under project "S2BDW" (F/050389/01-03/X32) - "Horizon2020" PON I&C2014-20 and by Regione Calabria under project "DLV LargeScale" (CUP $J28C17000220006$) - POR 2014-20.

Supplementary material

To view supplementary material for this article, please visit https://doi.org/10.1017/S1471068419000292.

References

Alviano, M., Dodaro, C., Leone, N., and Ricca, F. 2015. Advances in WASP. In F. Calimeri, G. Ianni, and M. Truszczynski (Eds.), *LPNMR 2015*, Volume 9345 of *LNCS*, pp. 40–54. Springer.

Alviano, M., Faber, W., Greco, G., and Leone, N. 2012. Magic sets for disjunctive datalog programs. *Artificial Intelligence 187*, 156–192.

Beck, H., Eiter, T., and Folie, C. 2017. Ticker: A system for incremental asp-based stream reasoning. *TPLP 17,* 5-6, 744–763.

Bogaerts, B. and Weinzierl, A. 2018. Exploiting justifications for lazy grounding of answer set programs. In *IJCAI 2018, July 13-19, 2018, Stockholm, Sweden.*, pp. 1737–1745. ijcai.org.

Calimeri, F., Cozza, S., Ianni, G., and Leone, N. 2008. Computable functions in ASP: theory and implementation. In *ICLP*, Volume 5366 of *Lecture Notes in Computer Science*, pp. 407–424. Springer.

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

Calimeri, F., Germano, S., Ianni, G., Pacenza, F., Perri, S., and Zangari, J. 2018. Integrating rule-based AI tools into mainstream game development. In *RuleML+RR 2018*, pp. 310–317.

Calimeri, F., Ianni, G., Perri, S., and Zangari, J. 2013. The eternal battle between determinism and nondeterminism: preliminary studies in the sudoku domain. *20th RCRA International Workshop. 2013*.

Calimeri, F., Perri, S., and Zangari, J. 2019. Optimizing answer set computation via heuristic-based decomposition. *Theory and Practice of Logic Programming*, 1–26.

https://doi.org/10.1017/S1471068419000292 Published online by Cambridge University Press

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

Dantsin, E., Eiter, T., Gottlob, G., and Voronkov, A. 2001. Complexity and expressive power of logic programming. *ACM Comput. Surv. 33,* 3, 374–425.

Dao-Tran, M., Eiter, T., Fink, M., Weidinger, G., and Weinzierl, A. 2012. Omiga : An open minded grounding on-the-fly answer set solver. In *JELIA*, Volume 7519 of *LNCS*, pp. 480–483. Springer.

De Cat, B., Denecker, M., and Stuckey, P. 2012. Lazy model expansion by incremental grounding. In *Technical Communications of ICLP 2012*, pp. 201–211.

Eiter, T., Ianni, G., and Krennwallner, T. 2009. Answer set programming: A primer. In *Reasoning Web School 2009*, Volume 5689 of *LNCS*, pp. 40–110. Springer.

Faber, W., Leone, N., and Pfeifer, G. 2004. Recursive aggregates in disjunctive logic programs: Semantics and complexity. In *JELIA*, Volume 3229 of *LNCS*, pp. 200–212. Springer.

Fages, F. 1994. Consistency of clark's completion and existence of stable models. *Meth. of Logic in CS 1,* 1, 51–60.

Gebser, M., Kaminski, R., Kaufmann, B., Romero, J., and Schaub, T. 2015. Progress in clasp series 3. In *LPNMR 2015*, Volume 9345 of *LNCS*, pp. 368–383. Springer.

Gebser, M., Kaminski, R., Kaufmann, B., and Schaub, T. 2019. Multi-shot ASP solving with clingo. *TPLP 19,* 1, 27–82.

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

Gebser, M., Maratea, M., and Ricca, F. 2017. The sixth answer set programming competition. *J. Artif. Intell. Res. 60*, 41–95.

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

Herbrand, J. 1930. Recherches sur la théorie de la démonstration.

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

Lefèvre, C., Béatrix, C., Stéphan, I., and Garcia, L. 2017. Asperix, a first-order forward chaining approach for answer set computing. *TPLP 17,* 3, 266–310.

Leone, N., Rullo, P., and Scarcello, F. 1997. Disjunctive stable models: Unfounded sets, fixpoint semantics, and computation. *Information and Computation 135,* 2, 69–112.

Motik, B., Nenov, Y., Piro, R., and Horrocks, I. 2019. Maintenance of datalog materialisations revisited. *Artificial Intelligence 269*, 76–136.

Nethercote, N., Stuckey, P., Becket, R., Brand, S., Duck, G., and Tack, G. 2007. Minizinc: Towards a standard CP modelling language. In *CP 2007*, pp. 529–543.

Pérez-Liébana, D., Samothrakis, S., Togelius, J., Schaul, T., and Lucas, S. 2016. General video game AI: competition, challenges and opportunities. In *AAAI 2016*, pp. 4335–4337.

Tarski, A. 1955. A lattice-theoretical fixpoint theorem and its applications. *Pacific J. Math. 5,* 2, 285–309.

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

Van Gelder, A., Ross, K., and Schlipf, J. 1991. The Well-Founded Semantics for General Logic Programs. *Journal of the ACM 38,* 3, 620–650.

Weinzierl, A. 2017. Blending lazy-grounding and CDNL search for answer-set solving. In *LPNMR*, Volume 10377 of *LNCS*, pp. 191–204. Springer.

https://doi.org/10.1017/S1471068419000292 Published online by Cambridge University Press