TPLP **20** (5): 719–734, 2020. © The Author(s), 2020. Published by Cambridge University Press
doi:10.1017/S147106842000040X

Incremental maintenance of overgrounded logic programs with tailored simplifications∗

Giovambattista Ianni, Francesco Pacenza, Jessica Zangari

Department of Mathematics and Computer Science, University of Calabria, Rende, Italy
(*e-mail:* `lastname@mat.unical.it`) - *https://www.mat.unical.it*

submitted 10 August 2020; accepted 11 August 2020

Abstract

The repeated execution of reasoning tasks is desirable in many applicative scenarios, such as stream reasoning and event processing. When using answer set programming in such contexts, one can avoid the iterative generation of ground programs thus achieving a significant payoff in terms of computing time. However, this may require some additional amount of memory and/or the manual addition of operational directives in the declarative knowledge base at hand. We introduce a new strategy for generating series of monotonically growing propositional programs. The proposed *overgrounded programs with tailoring* (OPTs) can be updated and reused in combination with consecutive inputs. With respect to earlier approaches, our *tailored simplification* technique reduces the size of instantiated programs. A maintained OPT slowly grows in size from an iteration to another while the update cost decreases, especially in later iterations. In this paper we formally introduce tailored embeddings, a family of equivalence-preserving ground programs which are at the theoretical basis of OPTs and we describe their properties. We then illustrate an OPT update algorithm and report about our implementation and its performance.

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

1 Introduction

A wide range of applicative contexts require to perform continuous reasoning over event streams. In turn, this requires the repeated execution of a reasoning task over the same fixed logic program, but with changing inputs. Among these applications, many can be categorized within the stream reasoning field (Dell'Aglio et al. 2017) and include, e.g., real-time motion tracking (Suchan et al. 2018), and decision making for agents, robots, and artificial players in videogames (Saribatur et al. 2019; Calimeri et al. 2018).

In the above applications, performance requirements are often very demanding: for instance, an artificial player, deployed in a real-time videogame, is subject to a very fast flow of input events, yet it is allowed a very limited time for each decision. In the GVGAI competition (Pérez-Liébana et al. 2016) this limit is just 40 milliseconds. However, engines based on the answer

∗ We thank the reviewers of this paper, whose constructive comments helped to improve our work. This work has been partially supported by MIUR under project "Declarative Reason- ing over Streams" (CUP H24I17000080001) – PRIN 2017, by MISE under project "S2BDW" (F/050389/01-03/X32) – "Horizon2020" PON I&C2014-20, by Regione Calabria under project "DLV Large Scale" (CUP J28C17000220006) – POR Calabria 2014-20.

https://doi.org/10.1017/S147106842000040X Published online by Cambridge University Press

set semantics are good candidates for reasoning in such domains, as they encompass advanced reasoning features, declarativity and a potentially good performance.

Recall that the typical workflow of Answer Set Programming (ASP) systems consists in an *instantiation* (or *grounding*) phase and a subsequent *solving* (or *answer sets search*) phase. In the first step, a *grounder* module produces an equivalent propositional program $gr(P \cup F)$ from an input non-ground logic program P and a set of facts F; in the latter step, a *solver* module applies dedicated search techniques on $gr(P \cup F)$ for computing the actual semantics of $P \cup F$ in the form of *answer sets* (Kaufmann et al. 2016). Repeated executions, called *shots* or *iterations*, can be conceptually abstracted to the task of finding the set of answer sets $AS(P \cup F_i)$ for a sequence of input fact sets $F_1, \ldots, F_n$.

Both the grounding and solving performance is critical when highly paced repeated executions are required. This stimulated a research effort towards the development of incremental reasoning techniques in the answer set programming community. The *clingo* system and its earlier prototypes (Gebser et al. 2019) allow a designer to procedurally control how and which parts of the logic program at hand must be incremented, updated and evaluated among consecutive shots. This approach introduces ample flexibility but requires a non-negligible knowledge of solver-specific internal algorithms. Nevertheless, declarativity and fast-prototyping capabilities are a priority in many development scenarios, such as the previously mentioned videogame industry. In this typical setting, designers look for easy and off-the-shelf scripting solutions, and do not have knowledge of declarative logic programming at all.

Another approach to incremental reasoning under the answer set semantics consists in using overgrounding techniques (Calimeri et al. 2019). In this work the grounding step is incrementally performed by maintaining an *overgrounded program* G_P, which is made "compatible" with new input facts by monotonically enlarging it from one shot to another. Overgrounding is attractive since no operational statements are required to incrementally drive the computation. The time performance of this technique is promising: an overgrounded program, after some update iterations, *converges* to a propositional theory general enough to be reused together with possible future inputs, with no further update required. This virtually eliminates grounding activities in later iterations, however the performance of solvers could decrease because of larger input programs. One can think at overcoming the limitations of overgrounding approaches by introducing techniques limiting the number of generated rules and reducing their size by applying known simplification methods for ground logic programs (Gebser et al. 2008; Faber et al. 2012).

However, nonobvious technical obstacles prevent a straightforward extension of overgrounding techniques in the above direction: in general indeed, simplification criteria are applied based on specific inputs. Consider, e.g., if one simplifies a ground program by properly removing atoms which are known to be true in all answer sets at a fixed shot. This, and more sophisticated simplification techniques can however be invalidated in later shots, as, for instance, if a logical assertion is no longer supported by the current input. Thus, diverse general questions arise. One could wonder which properties a ground program should have in order to be "reusable" with a family $\mathscr{F}$ of different inputs; also, it remains open whether a ground program can be modified in a way such that $\mathscr{F}$ can be enlarged with small computational cost, and how. The contributions of this paper are:

(i): We characterize a class of ground programs equivalent to the theoretical instantiation, called *tailored embeddings*. Tailored embeddings make it cleaner to deal with equivalence properties of simplified programs. *Overgrounded programs with tailoring* (OPTs in the following) are series of tailored embeddings that keep a monotonic growth approach, yet permitting simplifica-

https://doi.org/10.1017/S147106842000040X Published online by Cambridge University Press

tion techniques. (*ii*): We propose a new incremental grounding strategy, allowing to seamlessly adapt and reuse a ground program in consecutive evaluation shots. In particular, OPTs are generated by alternating *desimplification* steps, taking care of restoring previously deleted and reduced rules and *incremental* grounding steps, which add and simplify new rules. The maintained program becomes more and more general (i.e., the family of "compatible" input facts becomes increasingly larger) while moving from a shot to the next, and the update activity becomes progressively lighter. (*iii*): We implemented the above strategy in the *I-DLV* grounder. We report about the experimental activities we conducted, comparing with our previous overgrounding strategy and with other state-of-the-art systems. Results confirm that grounding times blend over iterations in the incremental setting, and that the performance of solvers takes advantage from the reduced size of OPTs with respect to plain overgrounded programs.

The tailored overgrounding approach has several advantages both of theoretical and practical relevance: tailored embeddings overcome many limits of the previous notion of embedding (Calimeri et al. 2019), and can be easily generalized to other semantics for logic programming, such as the well-founded semantics; their monotonic growth allows for easily implementing caching policies; if a grounding task must be interrupted, restarts on a new shot are almost straightforward to be implemented, since almost no rollback is required; the proposed framework is transparent to knowledge designers; highly general, non-optimized code can benefit from tailored overgrounding as there is no need to worry about which parts of logic programs might be more grounding-intensive. The rest of the paper is structured as follows: after overviewing our approach and briefly presenting preliminary notions, we introduce the notion of tailored embedding and its properties. We then present OPTs and a maintenance algorithm thereof, and we report about our prototype and its experimental evaluation; we eventually discuss related work before drawing final considerations. Most proofs are reported in the supplementary material corresponding to this paper at the TPLP archives.

2 Overgrounding with tailored simplifications: an overview

As mentioned, canonical ASP systems work by first *instantiating* a non-ground logic program P over input facts F, obtaining a propositional logic program $gr(P \cup F)$, and then computing the corresponding models, i.e., the set of answer sets $AS(gr(P \cup F))$. Notably, systems build $gr(P \cup F)$ as a significantly smaller and refined version of the theoretical instantiation but preserve semantics, i.e., $AS(gr(P \cup F)) = AS(P \cup F)$. The choice of the instantiation function gr impacts on both computing time and on the size of the instantiation. The grounding procedure gr usually maintains a set PT of "possibly true" atoms, initialized as $PT = F$; then, PT is iteratively incremented and used for generating only "potentially useful" propositional rules, up to a fixpoint. Grounding techniques evolved considerably since the early systems based on the explicit usage of domain predicates, and nowadays strategies for decomposing programs and for rewriting, simplifying and eliminating rules are of great help in controlling the size of the final instantiation. The reader can refer to related literature for an overview on grounding optimization techniques (Gebser et al. 2008; Calimeri et al. 2017; Calimeri et al. 2019). We show our approach with a simple example. Let us consider the program P_0 consisting of rules:

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

and the set of input facts $F_1 = \{e(c,a),\ e(a,b),\ ab(c)\}$. If only constants a, b and c are available, the theoretical grounding of $P_0 \cup F_1$ consists of all possible substitutions of variables with

https://doi.org/10.1017/S147106842000040X Published online by Cambridge University Press

constants, obtaining 9 and 27 instances for the two rules, respectively. With a smarter grounding function gr, one can 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:

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

A more "aggressive" grounding strategy could also cut or simplify rules: literals identified as definitely true can be eliminated and rules that cannot fire can be deleted. We can remove facts $e(a,b)$, and $e(c,a)$ from bodies of r_1 and r_2, respectively, and rule r_3 entirely, obtaining TG_1, composed of rules r'_1 and r'_2:

$$r'_1 : r(a,b) \leftarrow \sout{e(a,b)},\ not\ ab(a). \qquad r'_2 : r(c,b) \mid s(c,b) \leftarrow \sout{e(c,a)},\ r(a,b).$$
$$r_3 : \sout{r(c,a) \leftarrow e(c,a),\ not\ ab(c).}$$

Nevertheless, TG_1 can be seen as less re-usable than G_1, as it cannot be easily extended to a program which is equivalent to P_0 with respect to different input facts. Indeed, let us assume that, at some point, a subsequent run requires P_0 to be grounded over facts $F_2 = \{e(c,a),\ e(a,d)\}$. Note that, with respect to F_1, F_2 features the addition of facts $F^+ = \{e(a,d)\}$ and the deletion of facts $F^- = \{e(a,b),\ ab(c)\}$. The fact $e(c,a)$ belongs to both F_1 and F_2, and can be seen as a "persistent" fact. On the one hand, G_1 can be easily made valid for input F_2 by just adding new rules, which take into account F^+ as new possibly true facts. On the other hand, TG_1 can be of interest in that simplifications make it smaller than G_1. However, r'_1 in TG_1 and the absence of r_3 would cause wrong inferences for input facts F_2, since TG_1 is constructed on the assumption that $e(a,b)$ and $ab(c)$ are true.

Our proposed technique allows to adapt a simplified ground program TG_x to a new input F_{x+1} by iterating a *desimplification* step and an *incremental* step on TG_x. The desimplified version of TG_x is enriched with new simplified rules added in the incremental step. When F_2 is provided as input, the desimplification step restores r_3 and reverts r'_1 to r_1:

$$r_1 : r(a,b) \leftarrow \mathbf{e(a,b)},\ not\ ab(a). \qquad r'_2 : r(c,b) \mid s(c,b) \leftarrow \sout{e(c,a)},\ r(a,b).$$
$$r_3 : \mathbf{r(c,a)} \leftarrow \mathbf{e(c,a)},\ \mathbf{not\ ab(c)}.$$

Moreover, in the incremental step two new rules r_4 and r_5 are added depending on the new fact $e(a,d)$:

$$\mathbf{r_4} : \mathbf{r(c,d)} \mid \mathbf{s(c,d)} \leftarrow \mathbf{e(c,a)},\ \mathbf{r(a,d)}. \qquad \mathbf{r_5} : \mathbf{r(a,d)} \leftarrow \mathbf{e(a,d)},\ \mathbf{not\ ab(a)}.$$

Then, r_4 can be simplified by removing $e(c,a)$, whereas $e(a,d)$ can be deleted from r_5, obtaining:

$$\mathbf{r'_4} : \mathbf{r(c,d)} \mid \mathbf{s(c,d)} \leftarrow \mathbf{\sout{e(c,a)}},\ \mathbf{r(a,d)}. \qquad \mathbf{r'_5} : \mathbf{r(a,d)} \leftarrow \mathbf{\sout{e(a,d)}},\ \mathbf{not\ ab(a)}.$$

Thus, $TG_2 = \{r_1, r'_2, r_3, r'_4, r'_5\}$, whereas r'_4 and r'_5 were not formerly present in TG_1. We have now that TG_2 is equivalent to P_0 when evaluated over F_2 as input facts, whereas TG_2 with input facts F_1 would cause wrong inferences. Indeed, r'_5 is simplified according to new facts belonging to F_2 but not to F_1. Nevertheless, if $F_3 = F_1$ is submitted as input, the desimplification step would generate TG_3 from TG_2 by reverting the rule r'_5 to r_5.

One might notice that TG_3 is built assuming $F_1 \cup F_2 \cup F_3$ as possibly true facts, and assuming $F_1 \cap F_2 \cap F_3$ as certainly true facts. Intuitively, the last element of the series $\{TG_i\}$ is the one embracing the larger family of inputs for which "compatibility" is guaranteed, and requiring

https://doi.org/10.1017/S147106842000040X Published online by Cambridge University Press

lesser update work in later iterations: in case a fourth shot is requested over input facts $F_4 = \{e(a,d), e(c,a), e(a,b)\}$, the desimplification step will leave TG_3 unaltered and the incremental step will not generate new rules; this happens since possibly true facts and persistent facts are left unchanged. Thus, $TG_4 = TG_3$. We illustrate next how programs like TG_1, TG_2 and TG_3 are related to each other, and which formal requirements are necessary to develop a correct incremental grounding strategy.

3 Preliminaries

We assume to deal with finite programs under the answer set semantics. A program P is a set of rules. A rule r has form: $\alpha_1 \mid \alpha_2 \mid \ldots \mid \alpha_k \leftarrow \beta_1, \ldots, \beta_n, \mathit{not}\ \beta_{n+1}, \ldots, \mathit{not}\ \beta_m.$ where $k,n,m \geqslant 0$. $\alpha_1,\ldots,\alpha_k$ and $\beta_1,\ldots,\beta_m$ are called *atoms*. An atom has form $p(\mathbf{X})$, for p a predicate name and $\mathbf{X}$ a list of variable names and constants. A literal l has form a or $\mathit{not}\ a$, where a is an atom. The *head* of r is defined as $H(r) = \{\alpha_1, \ldots, \alpha_k\}$; the *positive body* of r is defined as $B^+(r) = \{\beta_1,\ldots,\beta_n\}$, whereas the *negative body* is $B^-(r) = \{\mathit{not}\ \beta_{n+1},\ldots,\mathit{not}\ \beta_m\}$. The *body* of r is $B(r) = B^+(r) \cup B^-(r)$; if $B(r) = \emptyset$ and $H(r) = \{a\}$ for a single atom a, then r is said a *fact*. 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 one atom $a \in B^+(r)$ mentioning X.

A program (resp. a rule, a literal, an atom) is said to be *ground* if it contains no variables. The set of all head atoms in a ground program G is denoted by $Heads(G) = \bigcup_{r\in G} H(r)$; its set of facts is $Facts(G)$. We assume to deal with a fixed *Herbrand Universe* consisting of a finite set of constants U and with programs that can be combined with separate input facts. Given a program P and a set of facts F, both P and F will only feature constants appearing in U. A *substitution* for a rule $r \in P$ is a mapping from the set of variables of r to the set U. A *ground instance* of a rule r is obtained by applying a substitution to r. Given a logic program P, the *theoretical instantiation (grounding)* $grnd(P)$ of P is defined as the finite set of all ground instances of rules in P.

We assume the reader is familiar with the notions of interpretations and models, and with the usual notation in the literature; in particular, when an interpretation I models a ground element e (i.e., an atom, a body, a head, a rule) this is denoted by $I \models e$. Given P and a set of facts F, a set of ground atoms A is an answer set of $P \cup F$ whenever A is a 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 $AS(P \cup F)$. In the following we recall some known results. We are given a logic program P and set of facts F.

Theorem 3.1
(Calimeri et al. 2008) Given a set of ground rules S, we define the operator $Inst(P,S) = \{r \in grnd(P)\ s.t.\ B^+(r) \subseteq Heads(S)\}$. We also define $Inst^1(P,F) = Inst(P,\emptyset \cup F)$, $\ldots$, $Inst^k(P,F) = Inst(P, Heads(Inst^{k-1}(P,F)) \cup F)$. The sequence $Inst^k(P,F)$, $k \geq 1$, converges to the least fixed point $Inst^f(P,F) = Inst^\infty(P,F)$ for some finite value f. Also, $AS(P \cup F) = AS(Inst^\infty(P,F) \cup F)$.

We herein recall the notion of embedding, i.e., an instantiation of P which contains a subset of rules of $grnd(P) \cup F$ sufficient to preserve the answer set semantics for some sets of input facts.

Definition 3.1
[*Embedding*] (Calimeri et al. 2019) For a set of ground rules $R \subseteq (grnd(P) \cup F)$, and a ground rule $r \in (grnd(P) \cup F)$, we say that:
- R embeds r *by body*, denoted $R \vdash_b r$, if $\forall a \in B^+(r)\ \exists r' \in R$ s.t. $a \in H(r')$;

https://doi.org/10.1017/S147106842000040X Published online by Cambridge University Press

- R embeds r *by head*, denoted $R \vdash_h r$, if $r \in R$;
- R embeds r, denoted $R \vdash r$, if either $R \nvdash_b r$ or $R \vdash_h r$.

A set of ground rules $E \subseteq grnd(P) \cup F$ is an *embedding program* for $P \cup F$, if $\forall r \in grnd(P) \cup F$, $E \vdash r$.

Note that embeddings mimic traditional model-theoretic notions for a logic program, but in the context of positive dependencies in ground rules. In particular, for an embedding E and a ground rule r, defining $E \vdash r \Leftrightarrow E \nvdash_b r \vee E \vdash_h r$ enforces a dependence from $B^+(r)$ to $H(r)$; similarly, for a model M, the statement $M \models r \Leftrightarrow M \not\models B(r) \vee M \models H(r)$ enforces an implicative dependence from $B(r)$ to $H(r)$.

Proposition 3.1
[Embedding equivalence] (Calimeri et al. 2019) Given an embedding E for $P \cup F$, $AS(E) = AS(grnd(P) \cup F) = AS(Inst^{\infty}(P,F) \cup F) = AS(\bigcap_{E' \in \mathbf{ES}} E')$, where $\mathbf{ES}$ is the set of embeddings of $P \cup F$.

Example 3.1
Let us consider program P_0 and the set of facts F_1 as mentioned in Section 2. Then $G_1 = \{r_1, r_2, r_3\} \cup F_1$ is an embedding for $P_0 \cup F_1$. According to Proposition 3.1, G_1 is also the minimal embedding for $P_0 \cup F_1$.

4 Tailored embeddings

We are given a logic program P and a set of input facts F. We will now work with possibly simplified versions of rules of $grnd(P)$. For a rule $r \in grnd(P) \cup F$, a *simplified rule* (or *simplified version*) s of r is a rule annotated with the set $B^*(s)$, where $B^*(s) = B(r) \setminus B(s)$. The rule r is denoted as $hom(s)$, i.e., r is the *homologous rule* of s belonging to the theoretical grounding whose body is obtainable as $B(hom(s)) = B(s) \cup B^*(s)$. For a set of simplified rules S we define $hom(S) = \{r \in grnd(P) \cup F \mid \exists s \in S \text{ and } hom(s) = r\}$. A rule $q \in grnd(P) \cup F$ is regarded as a simplified rule with $B^*(q) = \emptyset$ and $hom(q) = q$. Similarly, a set $Q \subseteq grnd(P) \cup F$ is regarded as a set of simplified rules with $hom(Q) = Q$.

Note that sets of simplified rules are not comparable under set inclusion, although one can consider, e.g., the set of rules $a \leftarrow b$ and $c \leftarrow d$ as a "somewhat smaller" subset of the set composed by rules $a \leftarrow b, c$ and $c \leftarrow d$. We thus appropriately generalize set inclusion and set intersection to sets of simplified rules. Given two sets of simplified rules S and R, we say that S is a *simplified subset* of R ($S \sqsubseteq R$) if for each $s \in S$ there is a rule $r \in R$ s.t. $B(s) \subseteq B(r)$ and $hom(s) = hom(r)$. The *simplified intersection* $R \sqcap Q$ of two set of simplified rules R and Q is:

$$R \sqcap Q = \{t \mid r \in R, q \in Q, B(t) = B(r) \cap B(q), B^*(t) = B^*(r) \cup B^*(q), hom(t) = hom(r) = hom(q)\}.$$

Example 4.1
Let us consider rules $r_1, \ldots, r_5$ and their primed versions as mentioned in Section 2. We assume that for each $i = 1 \ldots 5$, $hom(r'_i) = r_i$. Given $TG_3 = \{r_1, r'_2, r_3, r'_4, r_5\}$ and $TG_1 = \{r'_1, r'_2\}$, we have that $TG_1 \sqsubseteq TG_3$. For $T_0 = \{r_1, r'_2\}$, the intersection $T_0 \sqcap TG_1$ is instead the set $\{r'_1, r'_2\}$.

https://doi.org/10.1017/S147106842000040X Published online by Cambridge University Press

Definition 4.1
Given two sets of simplified rules R and Q, we define $Simpl(R,Q)$ as an operator working on each simplified rule $r \in R$ according to the following simplification types.

1. r is removed from R, if there is a literal $not\ a \in B^-(hom(r))$ s.t. $a \in Facts(Q)$;
2. r is removed from R if there is a atom $a \in B^+(hom(r))$ and $a \notin Heads(Q)$;
3. we move from $B(r)$ to $B^*(r)$ each atom $a \in B^+(hom(r))$ s.t. $a \in Facts(Q)$.

Intuitively, the types 1 and 3 depend on atoms which are assumed to be certainly true in any answer set; and the type 2 depends on atoms that are assumed to be certainly false in any answer set. With slight abuse of notation, we define $Simpl(R)$ as $Simpl(R,R)$. A number k of repeated applications of $Simpl$ to the same set R is denoted as $Simpl^k(R)$. Note that, for $k \geq 1$, $Simpl^{k+1}(R) \sqsubseteq Simpl^k(R)$: we denote the fixed point reached in finitely many steps by the sequence of values $Simpl^k(R)$ as $Simpl^\infty(R)$. We trivially extend the operators $\vdash$, $\vdash_h$ and $\vdash_b$ for a set of simplified rules on the left-hand side and for simplified rules on the right-hand side. As given next, a *tailored embedding* is a set of simplified rules which extends the notion of embedding by including the possibility of using simplification operations in order to obtain smaller, yet correct, ground programs.

Definition 4.2
[*Tailored embedding*] Given a set of simplified rules R and a rule $r \in grnd(P) \cup F$, we say that R *tailors* r ($R \Vdash r$) if at least one of the following holds:

1. $R \vdash r$;
2. there exists a simplified rule $s \in R$ such that $hom(s) = r$, $R \vdash_h s$ and $R \vdash_h a$ for each atom $a \in (B^+(r) \setminus B^+(s))$;
3. there is a literal $not\ a \in B^-(r)$ and $R \vdash a$.

A set of simplified rules E is a *tailored embedding* for $P \cup F$, if $\forall r \in grnd(P) \cup F$, $E \Vdash r$.

Intuitively, a ground rule r is tailored according to the new operator "$\Vdash$" either if it is embedded by R or, otherwise, there are in R the conditions for applying one of the possible simplification types to r. Note that $R \vdash b$ is meant as a shortcut for $R \vdash \{b \leftarrow \emptyset\}$.

Informally speaking, the notion of tailored embedding overcomes the one of embedding: although remarkably simple and useful, the latter notion lacks the fact that there are many other classes of optimized ground programs which are of interest, both theoretically and practically. In other words, embeddings do not properly formalize smaller, yet equivalence-preserving, ground programs produced by actual grounders. The new conditions describe equivalence-preserving ground programs in which a ground rule can be shortened or deleted at all, provided it is "tailored". This narrows the gap between the formalization (Calimeri et al. 2019) and real applications.

Example 4.2
Let us consider the ground program $TG_3 = \{r_1, r_2', r_3, r_4', r_5\}$ as shown in Section 2, and the set of facts F_1. $T = TG_3 \cup F_1$ is a tailored embedding since: *(a)* T tailors r_1, r_3, r_5 and all the facts in F_1, since T embeds all such rules; *(b)* $T \Vdash r_2$ since r_2' is a simplified version of r_2 for which $T \vdash_h r_2'$ and $T \vdash e(c,a)$; *(c)* similarly, $T \Vdash r_4$ since $T \vdash_h r_4'$ and $T \vdash e(c,a)$. Any other rule $r \in grnd(P_0) \cup F_1$ is trivially tailored since it holds that $T \nvdash_b r$ thus implying $T \vdash r$.

https://doi.org/10.1017/S147106842000040X Published online by Cambridge University Press

Tailored embeddings enjoy a number of interesting properties: an embedding is a tailored embedding (Proposition 4.1); a tailored embedding is equivalent to $P \cup F$ (Theorem 4.1); also, a simplified intersection of tailored embeddings is a tailored embedding (Proposition 4.2); and, importantly, the intersection of all tailored embeddings represents the least tailored embedding under simplified set inclusion and corresponds to an iterative, operational construction made using the *Simpl* and *Inst* operators (Theorem 4.2 and Corollary 4.1).

Proposition 4.1
An embedding E for $P \cup F$ is a tailored embedding for $P \cup F$.

Proof
We observe that given an embedding E for $P \cup F$, for each $r \in grnd(P) \cup F$, we have that $E \vdash r$. Then $E \Vdash r$ by the case 1 of Definition 4.2. □

Example 4.3
Let us consider again the example of Section 2 and in particular, the ground program $G_1 = \{r_1, r_2, r_3\} \cup F_1$. G_1 is an embedding for $P_0 \cup F_1$ as each rule $r \in grnd(P_0) \cup F_1$ is embedded by G_1: if $r \in G_1$, $G_1 \vdash_h r$, whereas if $r \in \{grnd(P_0) \cup F_1 \setminus G_1\}$, $G_1 \nvdash_b r$. Also, G_1 is a tailored embedding for $P_0 \cup F_1$ because for each rule $r \in grnd(P_0) \cup F_1$ we have that $G_1 \vdash r$, since we can apply the case 1 of Definition 4.2. Note that for $P_0 \cup F_1$, the ground program $TG_1 = \{r'_1, r'_2\} \cup F_1$ is a tailored embedding but it cannot be an embedding. Indeed, $TG_3 \nvdash r_3$ since $TG_3 \vdash_b r_3$ and $TG_3 \nvdash_h r_3$ and according to Definition 3.1, TG_3 had to embed all rules in $grnd(P_0) \cup F_1$ to be an embedding.

Theorem 4.1
[Equivalence]. Given a tailored embedding E for $P \cup F$, then $AS(grnd(P) \cup F) = AS(E)$.

Proposition 4.2
[Intersection]. Given two tailored embeddings E_1 and E_2 for $P \cup F$, $E_1 \sqcap E_2$ is a tailored embedding for $P \cup F$.

Theorem 4.2
Let **TE** be the set of tailored embeddings of $P \cup F$ and $\mathscr{E} = Inst^{\infty}(P, F) \cup F$. Then,

$$Simpl^{\infty}(\mathscr{E}) = \bigsqcap_{T \in \mathbf{TE}} T.$$

Corollary 4.1
By combining Th. 4.1, Pr. 4.2 and Th. 4.2, we have that:

$$AS(P \cup F) = AS\Big(\bigsqcap_{T \in \mathbf{TE}} T\Big) = AS(Simpl^{\infty}(Inst^{\infty}(P, F) \cup F)).$$

Example 4.4
For program P_0 and facts F_1 of Section 2, the least tailored embedding of $P_0 \cup F_1$ under simplified set inclusion is the set $\{r'_1, r'_2\} \cup F_1$.

https://doi.org/10.1017/S147106842000040X Published online by Cambridge University Press

5 Overgrounding with tailoring

We illustrate in this section our technique for maintaining appropriate series of tailored embeddings which we call *overgrounded programs with tailoring* (OPTs).

In the following, the logic program P will be coupled with a sequence of sets of input facts $F_1, \ldots, F_n$. We aim to incrementally compute the sets $AS(P \cup F_1), \ldots, AS(P \cup F_n)$ by reducing the burden of the grounding step at the bare minimum, especially in later iterations. We update and maintain one element of the series of OPTs $G_1, \ldots, G_n$ via the repeated execution of an incremental instantiation function called INCRINST, and taking as arguments the program P, a ground program G and a set of input facts F. At iteration 1, we initialize the global sets of ground atoms $D = AF = PF = \emptyset$, and we let $G_1 = \text{INCRINST}(P, \emptyset, F_1)$. For an iteration $i > 1$, we will set $G_i = \text{INCRINST}(P, G_{i-1}, F_i)$.

The series $G_1, \ldots, G_n$ has three useful properties: *(i)* for each i, $G_i \cup F_i$ is a tailored embedding for $P \cup F_i$ and thus $AS(P \cup F_i) = AS(G_i \cup F_i)$; *(ii)* for the shot $i+1$, the INCRINST function obtains G_{i+1} from G_i by means of an iterative process, which repeatedly undoes now invalid simplifications in G_i (the *desimplification* step) and then computes additional new rules ΔG_{i+1} (the *incremental grounding* step); *(iii)* G_{i+1} extends G_i, as all the rules of G_i appear in G_{i+1} possibly in their desimplified version, i.e., $G_i \sqsubseteq G_{i+1}$. The global set D collects the rules that were deleted at some iteration and could be restored later on, whereas AF and PF keep record of so called *accumulated facts* and *persistently true facts*, respectively. After computing G_i for a shot i, we will have that $AF = \bigcup_{1 \leq k \leq i} F_k$ and $PF = \bigcap_{1 \leq k \leq i} F_k$. Intuitively, $Heads(G_i)$ will represent *possibly true atoms* built by applying the *Inst* operator starting from AF as initial set of possible atoms. An atom outside $Heads(G_i) \cup AF$ is assumed to be *certainly false* and might trigger simplifications of type 2. Similarly, $Facts(G_i)$ will be taken as the set of *certainly true* atoms which allow to apply simplifications of types 1 and 3.

Outline of the INCRINST *function.* An abstract version of the INCRINST function is given in the next page. Let us assume to be at iteration $i+1$ for $i > 1$. The INCRINST function is composed of a DESIMPL step and a ΔINST step. On the one hand, in the DESIMPL step the rules in G_i are possibly *desimplified* whereas previously deleted rules are possibly restored. On the other hand, undeleted rules and new facts $NF = F_{i+1} \setminus AF$ can trigger the generation of new rules, which are incrementally processed in the ΔINST step. These new rules are simplified and added to G_{i+1}.

The set NR keeps track of rules restored from D and of new rules added in the ΔINST step. Atoms in $Heads(NR) \cup NF$ can invalidate simplifications of type 2 as they represent no longer certainly false atoms. The set OF is instead used to keep track of atoms that are no longer assumed to be certainly true at the current shot; the atoms in OF can invalidate former simplifications of types 1 and 3. The iterative process internal to INCRINST continues until no new rules are added and no new derived facts need to be retracted, i.e., when both NR and OF do not change anymore.

Desimplification step. The DESIMPL step makes an update on a copy DG of the current ground program G in which simplifications of types 1 through type 3 are undone. Note that the desimplification might trigger new additions to OF and NR, which in turn can cascade new desimplifications and/or new incremental additions. We purposely allow redundant desimplifications: an atom $f \in Facts(G)$ might be added to OF as soon as a rule r with $H(r) = \{f\}$ is desimplified

https://doi.org/10.1017/S147106842000040X Published online by Cambridge University Press

Input: Non-ground program P, ground program G, input facts F
Output: A desimplified and enlarged ground program
Updates: the set of deleted rules D, collection of sets AF and PF

```
1: function IncrInst(P,G,F)
2:     DG = G,
3:     NR = ∅, NF = F \ AF, OF = PF \ F
4:     AF = AF ∪ F, PF = PF ∩ F
5:     while NR∪NF or OF have new additions do
6:         // DeSimpl step
7:         for all r ∈ D do                                   //undo simpl. types 1 and 2
8:             L1 = {not a ∈ B−(r) s.t. a ∈ OF}
9:             L2 = {a ∈ B+(r) s.t. a ∈ Heads(NR) ∪ NF}
10:            if L1 ∪ L2 ≠ ∅ then
11:                D = D \ {r}
12:                NR = NR ∪ {r}
13:            end if
14:        end for
15:        for all r ∈ DG do
16:            L3 = {a ∈ B+(r) s.t. a ∈ OF}
17:            for all l ∈ L3 do                               //undo simpl. type 3
18:                if B(r) = ∅ ∧ ‖H(r)‖ = 1 then
19:                    OF = OF ∪ H(r)
20:                end if
21:                B(r) = B(r) ∪ {l}
22:            end for
23:        end for
24:        // ΔInst step
25:        do
26:            for all r ∈ P do
27:                for all g ∈ getInstances(r,DG,NR∪NF) do
28:                    if Simpl[1,3]({g},NR∪F) = ∅ then        //g is deleted
29:                        D = D ∪ {g}
30:                    else
31:                        NR = NR ∪ Simpl[1,3]({g},NR∪F)
32:                    end if
33:                end for
34:            end for
35:        while there are additions to NR
36:    end while
37:    S = Simpl∞(NR,DG∪NR∪F), D = D ∪ hom(NR) \ hom(S)
38: return DG∪S
39: end function
```

(line 19). However, although there can be some other rule r_f in G such that $H(r_f) = \{f\}$ and $B(r_f) = \emptyset$, the restore operation on r does not affect the correctness of DG.

Incremental grounding step. In this step we instantiate and simplify each rule $r \in P$ that can be constructed using the new ground atoms available in $Heads(NR) \cup NF$ up to a fixpoint. The *getInstances* function processes a non-ground rule r, the ground program DG and the set $NR \cup NF$. All possible new matches for the input rule r are differentially obtained and simplified. The *getInstances* function can be implemented by carefully adapting semi-naive evaluation

https://doi.org/10.1017/S147106842000040X Published online by Cambridge University Press

techniques. This can avoid the generation of duplicated rules, thus saving computation time and memory consumption.

Simplifications. Our algorithm applies simplifications over new rules NR only and in two separate moments: *(i)* as soon as a new rule is generated (line 31) and *(ii)* at the end of the main cycle (line 37). In the latter case, we apply all simplification types. In the former case, the $Simpl[1,3]$ operator is meant to apply only simplifications of types 1 and 3. These two simplification types can be applied earlier and can prevent the generation of rules that will be nonetheless deleted later. We observe that we simplify only newly added rules appearing in NR, but with respect to the current value of F. This will make G_{i+1} not "compatible" with inputs F_k, $1 \leq k \leq i$. Nevertheless, if some F_k appears again as input in a later iteration, the correctness of G_{i+1} can be achieved with a further desimplification step. It is worth noting that a more conservative strategy could consider only simplifications depending on PF.

Example 5.1
Let us recall again the example given in Section 2 and consider program P_0, the intermediate program $TG_1 = \{r'_1, r'_2\}$ and the set of input facts $F_1 = \{e(c,a),\ e(a,b),\ ab(c)\}$. $TG_1 \cup F_1$ is a tailored embedding for $P_0 \cup F_1$. Assume also that at this stage $D = \{r_3\}$. Given a new set of input facts $F_2 = \{e(c,a),\ e(a,d)\}$, we have that $\text{INCRINST}(P, TG_1, F_2)$ works as follows: NF is initially set to $\{e(a,d)\}$, $OF = \{e(a,b),\ ab(c)\}$, and DG is initially set to TG_1. The DESIMPL step will produce the updated set $DG = \{r_1, r'_2, r_3\}$ by modifying r'_1 in r_1, while the rule r_3 is undeleted and moved from D to NR. The ΔINST step generates the new rules r'_5 and r'_4 and adds them to NR. r'_5 is a simplified version of r_5 constructed using the new atom $e(a,d)$, while r'_4 is a reduced version of r_4 built using the new atom $r(a,d)$. No further desimplifications and changes to DG, NR and OF happen in the next DESIMPL and ΔINST steps nor in the final simplification. The set $\{r_1, r'_2, r_3, r'_4, r'_5\}$ is eventually returned.

Theorem 5.1
Let $G_1 = \text{INCRINST}(P, \emptyset, F_1)$. For each i s.t. $1 < i \leq n$, let $G_i = \text{INCRINST}(P, G_{i-1}, F_i)$. Then for each i s.t. $1 \leq i \leq n$, $AS(G_i \cup F_i) = AS(P \cup F_i)$.

The proof of the above theorem, shown in the supplementary material corresponding to this paper at the TPLP archives, goes along the lines of showing how to enlarge, under simplified set inclusion, a tailored embedding $G_{i-1} \cup F_{i-1}$ for $P \cup F_{i-1}$ to a tailored embedding $G_i \cup F_i$ for $P \cup F_i$.

6 Implementation and experimental evaluation

The tailored overgrounding strategy described above has been implemented by extending the *I-DLV* grounder (Calimeri et al. 2016; Calimeri et al. 2019) to a version called *I²-DLV*. The new system allows to preload a non-ground logic program P, to iteratively submit input facts F_i, and to obtain $AS(P \cup F_i)$. During the process, a simplified subset TG of $grnd(P)$ is kept in memory. Whenever new input F_{i+1} is submitted, TG is updated according to the tailored overgrounding strategy; a filtering stage then pipes relevant rules to the solver of choice. Homologous, simplified and deleted rules are kept track of by adding mark-up to a single copy of each rule. Our evaluation was conducted in order to assess *(a)* the size of inputs fed to solvers and *(b)* the evolution of the performance of the combination of grounder and solver, given also the

https://doi.org/10.1017/S147106842000040X Published online by Cambridge University Press

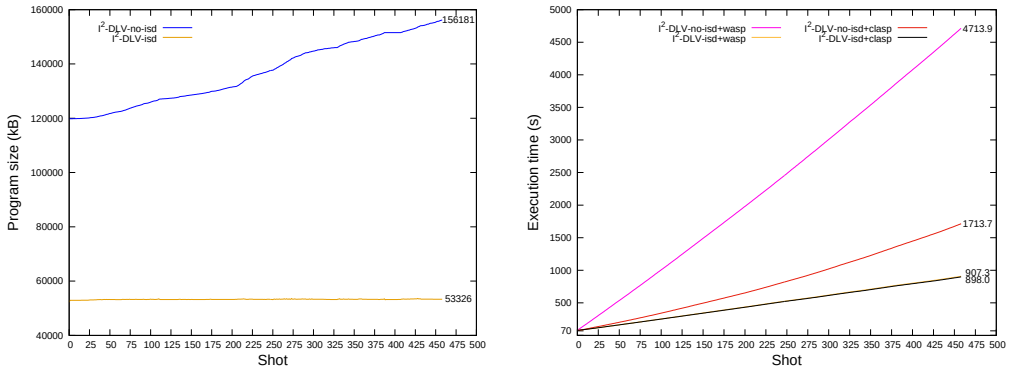


Figure 1: Results of Pac-Man benchmark. Size on the left side (a) and times on the right side (b).

changing instantiation size. Since sources of choice points are left substantially unchanged by simplification activities, we expected good improvements in performance due to faster solving times for deterministic parts of ground programs. We considered two benchmarks taken from two real world settings with different specific features: Pac-Man (Calimeri et al. 2018) and Content Caching (Beck et al. 2017; Eiter et al. 2019). The two benchmarks constitute good and generalizable real cases of incremental scenarios: the Pac-Man game allows to assess effectiveness of overgrounding for continuous reasoning in the context of videogames, while Content Caching is a typical example of decision making over fast-paced event streams. Experiments have been performed on a NUMA machine equipped with two 2.8GHz AMD Opteron 6320 CPUs, with 16 cores and 128GB of RAM. The measurements have been performed using WASP version 3.0.0, clasp integrated in clingo version 5.4.0 and Ticker version 1.0. We used two grounder versions: $\mathscr{I}^2$-*DLV*-isd stands for our new grounder featuring the new incremental simplification and desimplification techniques (*isd* in the following), while $\mathscr{I}^2$-*DLV*-no-isd is a new improved implementation of plain overgrounded programs (Calimeri et al. 2019), in which isd techniques are disabled.

Pac-Man. The first experiment has been conducted in the domain of the classic real-time game Pac-Man. We used the logic program P_{pac} that describes the decision-making process of an artificial player guiding the Pac-Man in a real implementation (Calimeri et al. 2018). P_{pac} is repeatedly executed together with different inputs describing the current status of the game map, like e.g., the current position of enemy ghosts, the position of pellets, etc. Several parts of P_{pac} can be considered "grounding-intensive", like the ones describing the predicate $distance(X_1,Y_1,X_2,Y_2,D)$, where D is computed for all pairs of points $(X_1,Y_1)\times(X_2,Y_2)$, taking care of the shortest path between (X_1,Y_1) and (X_2,Y_2), given the shape of the labyrinth in the game map. The evaluation has been conducted by logging a series of 459 consecutive sets of input data taken during an actual game; such inputs were run along with P_{pac} in a controlled environment outside the game engine. The solving task was performed using WASP (Alviano et al. 2015) and clasp (Gebser et al. 2012). Figure 1 *(a)* compares instantiation sizes for both grounders, while Figure 1 *(b)* reports about cumulative execution times for the four possible combinations of grounders and solvers. The X axis diagrams data in shot execution order. Results show that both solvers benefit of the smaller inputs produced by tailored overgroundings, with WASP showing a remarkable

https://doi.org/10.1017/S147106842000040X Published online by Cambridge University Press

improvement. For all the four combinations, the slope of the cumulative time curve reflects an almost constant execution time, with the exception of the first shot measuring around 70 seconds. In this shot, grounding times account for almost all the computation time. A slight progressive worsening in the execution time per shot can be seen especially for combinations involving the old grounder $\mathscr{I}^2$-*DLV*-no-isd. This is due to the larger program input fed to solvers. For both grounders, we noticed that instantiation times become immediately negligible in later iterations, with $\mathscr{I}^2$-*DLV*-isd being around 7% less performant than $\mathscr{I}^2$-*DLV*-no-isd because of simplification and desimplification activity.

Content Caching. In this benchmark, the caching policy of a given video content is controlled using a logic program P_{cc}. The caching policy of choice is encoded in the answer sets of $P_{cc} \cup E$ where E encodes a continuous stream of events describing the evolving popularity level of the content at hand. This application has been originally designed in the LARS framework using time window operators in order to quantify over past events (Beck et al. 2018). We adapted the conversion method specified in the work presenting Ticker (Beck et al. 2017) to obtain P_{cc} as a plain logic program under answer set semantics, while events were converted to corresponding sets of input facts. These kinds of stream reasoning applications can be fairly challenging, depending on the pace of events and the size of the time window at hand. Our experiments were run in a worst-case scenario in which the caching policy could be decided based on events happening in the last 100 seconds, were the event pace was assumed to be 0.1 seconds. In this setting, a stream reasoning system must be able to deal with a total of 100×10 different timestamp symbols, and with proportionally large ground programs. Again, we compared the four combinations of grounders and solvers, and the Ticker system in its two implementations: the Ticker ad-hoc truth maintenance based version (ticker-incr), and the clingo-based one (ticker-asp). Figure 2 *(a)* shows that both grounders add new rules to their respective overgrounded program up to around shot 1000, which corresponds to the number of time points allowed in the chosen 100 seconds window. After this threshold, instantiated programs stay constant, with $\mathscr{I}^2$-*DLV*-isd generally producing a smaller input. In Figure 2 *(b)*, the slope of cumulative times shows that ticker-incr has some initial computational cost due to its pre-grounding phase, then performs better in terms of later per-shot times. The four combinations using our grounders have less initial computational cost, while their per-shot times increase slightly in later iterations, with $\mathscr{I}^2$-*DLV*-no-isd paired with clasp having the best performance, which is quite close to ticker-incr. Ticker-asp does not feature incremental optimization strategies, thus it is not comparable with other solutions.

7 Related work

One of the differentiating ideas of our approach is that OPTs can be "patched" and adapted to new inputs only by adding new information. In other words, OPTs grow monotonically, although an effort has been done to maintain this growth slow. A second characteristic of our approach is that we do not include modelling directives for controlling incrementality and multi-shot programs. This choice comes with many advantages (like easier usage and modelling) at the price of the loss of control. In the above respects, our proposal has connections with several lines of research.

It is worth mentioning the early and recent work surveying and proposing incremental update of pool of views (Motik et al. 2019). The aforesaid approaches focus on query answering over stratified Datalog programs and aim to just materialize query answers; our focus is on a gener-

https://doi.org/10.1017/S147106842000040X Published online by Cambridge University Press

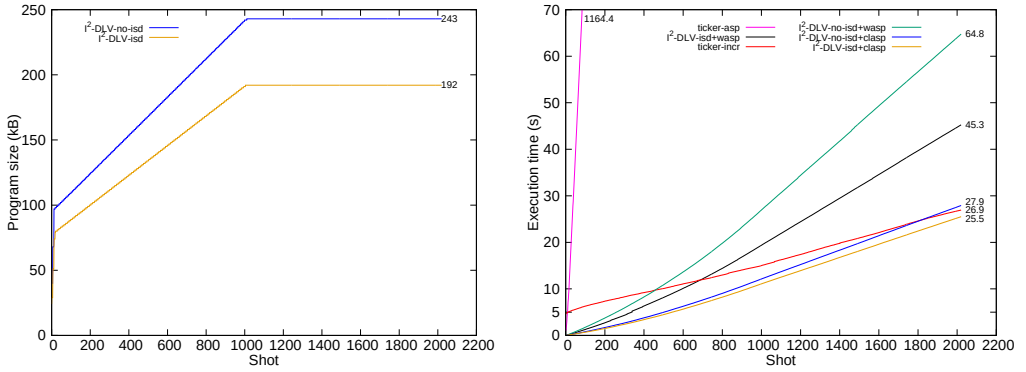


Figure 2: Results of Content Caching benchmark. Size on the left side (a) and times on the right side (b).

alized setting, where disjunction and unstratified negation are allowed and propositional logic programs are materialized and maintained, for the purpose of computing answer sets. In contrast with typical delete/rederive techniques, which require an additional effort to avoid overdeletions and ensure correctness, we purposely allow to perform more desimplifications than necessary. Also, the absence of rederivation activities allows us to keep incremental grounding times low.

In the clingo approach (Gebser et al. 2019) the notion of incrementality is conceived in a different way: problems are modelled by thinking in terms of "layers" of modules, for which users specify the grounding and solving sequence thereof. In the context of overgrounded programs, modelling can be focussed on a single declarative program. The incremental evaluation over the sequence of input facts is implicit and does not require the user's attention. Full incremental reasoning, the widest general setting in which a logic program is subject to arbitrary changes and one aims to implicitly maintain answer sets, is to date an almost unexplored topic. 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 back-end incremental truth maintenance techniques under ASP semantics, but is limited to a language fragment with no recursion. We believe a tighter level of integration between grounders and solvers is necessary in order to achieve full incremental reasoning. Among tightly integrated approaches, it is worth mentioning lazy grounding (Dal Palù et al. 2009; Lefèvre et al. 2017; Bomanson et al. 2019). Note that overgrounding is essentially orthogonal to lazy grounding techniques, since these latter 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, while still keeping the two evaluation steps separated. Finally, it is worth highlighting that overgrounded programs with tailoring can be seen as an application of relativized hyperequivalent logic programs (Truszczynski and Woltran 2009). A member G of a sequence of OPTs is a logic program which is equivalent to P relative to (part of) a finite set of inputs $F_1, \ldots, F_n$. Investigation on the hyperequivalence properties of OPTs, possibly under semantics other than the answer set one, deserves further research.

8 Conclusions

Herein we reported about theoretical properties of tailored embeddings and overgrounded programs with tailoring; we then presented an algorithm which is at the core of a new incremental

https://doi.org/10.1017/S147106842000040X Published online by Cambridge University Press

grounder. The experiments conducted on our implementation show that smaller instantiations are beneficial for the overall ground & solve pipeline, and that the grounding effort can be blended over multiple shots, with higher computational impact in earlier iterations. This paves the way to the design of an expressive reasoning system with very short response times, and capable to work over streams of inputs in highly dynamic environments. The tailored overgrounding strategy is suitable for several extensions. We are currently exploring the possibility of discarding rules when a memory cap is required; also we are investigating towards a cancel/restart technique useful when a reasoning task is aborted. In both the above scenarios, thanks to the monotonic growth properties of OPTs, partially computed instantiations can be reused on the next shot with almost no rollback burden. The contexts in which OPTs are beneficial are not restricted to the above, and will be matter of further research. Detailed experiment results, benchmark encodings, datasets and the binaries of the systems and repeatability information are available at https://github.com/DeMaCS-UNICAL/I-DLV/wiki/Incremental-IDLV.

References

ALVIANO, M., DODARO, C., LEONE, N., AND RICCA, F. Advances in WASP. In *LPNMR 2015*, Volume 9345 of LNCS, pp. 40–54. Springer.

BECK, H., BIERBAUMER, B., DAO-TRAN, M., EITER, T., HELLWAGNER, H., AND SCHEKOTIHIN, K. Stream reasoning-based control of caching strategies in CCN routers. In *ICC 2017*, pp. 1–6. IEEE.

BECK, H., DAO-TRAN, M., AND EITER, T. 2018. LARS: A logic-based framework for analytic reasoning over streams. *Artificial Intelligence 261*, 16–70.

BECK, H., EITER, T., AND FOLIE, C. 2017. Ticker: A system for incremental ASP-based stream reasoning. *Theory and Practice of Logic Programming 17,* 5-6, 744–763.

BOMANSON, J., JANHUNEN, T., AND WEINZIERL, A. Enhancing lazy grounding with lazy normalization in answer-set programming. In *AAAI 2019*, pp. 2694–2702. AAAI Press.

CALIMERI, F., COZZA, S., IANNI, G., AND LEONE, N. 2008. Computable functions in ASP: theory and implementation. In *ICLP* 2008, Volume 5366 of LNCS, pp. 407–424. Springer.

CALIMERI, F., FUSCÀ, D., PERRI, S., AND ZANGARI, J. 2016. I-DLV: The New Intelligent Grounder of DLV. In *AI*IA*, Volume 10037 of LNCS, pp. 192–207. 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. Integrating rule-based AI tools into mainstream game development. In *RuleML+RR 2018*, Volume 11092 of LNCS, pp. 310–317. Springer.

CALIMERI, F., IANNI, G., PACENZA, F., PERRI, S., AND ZANGARI, J. 2019. Incremental answer set programming with overgrounding. *Theory and Practice of Logic Programming 19,* 5-6, 957–973.

CALIMERI, F., PERRI, S., AND ZANGARI, J. 2019. Optimizing answer set computation via heuristic-based decomposition. *Theory and Practice of Logic Programming 19,* 4, 603–628.

DAL PALÙ, A., DOVIER, A., PONTELLI, E., AND ROSSI, G. 2009. GASP: answer set programming with lazy grounding. *Fundamenta Informaticae 96,* 3, 297–322.

DELL'AGLIO, D., VALLE, E. D., VAN HARMELEN, F., AND BERNSTEIN, A. 2017. Stream reasoning: A survey and outlook. *Data Science 1,* 1-2, 59–83.

EITER, T., OGRIS, P., AND SCHEKOTIHIN, K. 2019. A distributed approach to LARS stream reasoning (system paper). *Theory and Practice of Logic Programming 19,* 5-6, 974–989.

FABER, W., LEONE, N., AND PERRI, S. 2012. The intelligent grounder of DLV. In *Correct Reasoning - Essays on Logic-Based AI in Honour of Vladimir Lifschitz*, Volume 7265 of LNCS, pp. 247–264. Springer.

https://doi.org/10.1017/S147106842000040X Published online by Cambridge University Press

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

GEBSER, M., KAMINSKI, R., KAUFMANN, B., AND SCHAUB, T. Multi-shot ASP solving with clingo. *Theory and Practice of Logic Programming 19,* 1, 27–82.

GEBSER, M., KAUFMANN, B., NEUMANN, A., AND SCHAUB, T. Advanced preprocessing for answer set solving. In *ECAI 2008*, Volume 178 of FAIA, pp. 15–19. IOS Press.

GEBSER, M., KAUFMANN, B., AND SCHAUB, T. Conflict-driven answer set solving: From theory to practice. *Artificial Intelligence 187*, 52–89.

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. *Theory and Practice of Logic Programming 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.

PÉREZ-LIÉBANA, D., SAMOTHRAKIS, S., TOGELIUS, J., SCHAUL, T., AND LUCAS, S.. General video game AI: competition, challenges and opportunities. In *AAAI 2016*, pp. 4335–4337. AAAI Press.

SARIBATUR, Z. G., PATOGLU, V., AND ERDEM, E. 2019. Finding optimal feasible global plans for multiple teams of heterogeneous robots using hybrid reasoning: an application to cognitive factories. *Autonomous Robots 43,* 1, 213–238.

SUCHAN, J., BHATT, M., WALEGA, P. A., SCHULTZ, C. P. L.. Visual explanation by high-level abduction: On answer-set programming driven reasoning about moving objects. In *AAAI 2018*, pp. 1965–1972. AAAI Press.

TRUSZCZYNSKI, M. AND WOLTRAN, S. 2009. Relativized hyperequivalence of logic programs for modular programming. *Theory and Practice of Logic Programming 9,* 6, 781–819.

https://doi.org/10.1017/S147106842000040X Published online by Cambridge University Press