Proceedings of

2nd Workshop on Grounding and Transformations for Theories With Variables

September 15th 2013

David Pearce, Shahab Tasharrofi, Evgenia Ternovska and Concepción Vidal (Eds.)

associated to

12th International Conference on Logic Programming and Nonmonotonic Reasoning

Corunna, Spain, September 15-19 2013

2nd Workshop on Grounding and Transformations for Theories With Variables

Corunna, September 15th 2013

Organizing Committee

David Pearce,	Universidad Politécnica de Madrid, Spain.
Shahab Tasharrofi,	Simon Fraser University, Canada.
Evgenia Ternovska,	Simon Fraser University, Canada.
Concepción Vidal,	University of Corunna, Spain.

Program Committee

Shahab Tasharrofi,	Simon Fraser University, Canada
Stefan Woltran,	Vienna University of Technology, Austria
Simona Perri,	University of Calabria, Italy
Marcello Balduccini,	Kodak Research Laboratories, USA
Mirek Truszczynski,	University of Kentucky, USA
Torsten Schaub,	University of Potsdam, Germany
Joohyung Lee,	Arizona State University, USA
Marc Denecker,	K.U.Leuven, Belgium
Enrico Pontelli,	New Mexico State University, USA
Evgenia Ternovska,	Simon Fraser University, Canada
Agustin Valverde,	Universidad de Málaga, Spain
Stefania Costantini,	Universita di L'Aquila, Italy
Concepción Vidal,	University of Corunna, Spain

Table of Contents

Extending ASP with functions: implementation techniques and impact on grounding modules

Giovambattista Ianni

Department of Mathematics and Computer Science,
University of Calabria ianni@mat.unical.it

Function symbols are acknowledged as a useful modelling tool in logic programming. Its introduction in existing languages comes however at the price of solving several technical issues, both theoretical and implementation-related. The undecidability of reasoning on ASP with functions, implied that functions were subject to severe restrictions or disallowed at all, drastically limiting ASP applicability. In this talk we overview the research about the introduction of function symbols (functions) in Answer Set Programming (ASP) – conducted at University of Calabria in the Artificial Intelligence group of the Department of Mathematics and Computer Science. We show how most of the technical difficulties preventing this introduction were addressed with the introduction of a family of expressive and decidable classes of programs with functions (finitely-ground programs, fd-programs and DFRP programs). Concerning implementation, we illustrate the impact of the introduction of functions symbols (and, as a byproduct, of lists and sets constructs) in the DLV system, by showing two different realization methods.

References

1. Leone, N.; Pfeifer, G.; Faber, W.; Eiter, T.; Gottlob, G.; Perri, S.; and Scarcello, F. The DLV System for Knowledge Representation and Reasoning. *ACM TOCL* 7(3):499–562.
2. F. Calimeri, S. Cozza, and G. Ianni. External sources of knowledge and value invention in logic programming. *Annals of Mathematics and Artificial Intelligence*, 50(3–4):333–361, 2007.
3. Calimeri, F.; Cozza, S.; Ianni, G.; and Leone, N. Computable Functions in ASP: Theory and Implementation. In *ICLP 2008*, 407–424.
4. Calimeri, F.; Cozza, S.; Ianni, G.; and Leone, N. An ASP System with Functions, Lists, and Sets. In *LPNMR'09*, 483–489.
5. Calimeri, F.; Cozza, S.; Ianni, G.; and Leone, N. Magic Sets for the Bottom-Up Evaluation of Finitely Recursive Programs. In *LPNMR'09*, 71–86.
6. Calimeri, F.; Cozza, S.; Ianni, G.; and Leone, N. since 2008. DLV-Complex homepage. http://www.mat.unical.it/dlv-complex.
7. M. Alviano, W. Faber, and N. Leone. Disjunctive asp with functions: Decidable queries and effective computation. *Theory and Practice of Logic Programming, 26th Int'l. Conference on Logic Programming (ICLP'10) Special Issue*, 10(4–6):497–512, 2010.
8. Calimeri, F.; Cozza, S.; Ianni, G.; and Leone, N. Enhancing ASP by Functions: Decidable Classes and Implementation Techniques. AAAI 2010.
9. Calimeri, F.; Cozza, S.; Ianni, G.; and Leone, N. Finitely recursive programs: Decidability and bottom-up computation. *AI Commun. (AICOM)*, 24(4):311-334 (2011).

10. M. Alviano, W. Faber, N. Leone and M. Manna Disjunctive datalog with existential quantifiers: Semantics, decidability, and complexity issues. *Theory and Practice of Logic Programming*, 12(4–5):701–718, 2012.

Grounding HEX-Programs with Expanding Domains*

Thomas Eiter, Michael Fink, Thomas Krennwallner, and Christoph Redl

Institut für Informationssysteme, Technische Universität Wien
Favoritenstraße 9-11, A-1040 Vienna, Austria
{eiter,fink,tkren,redl}@kr.tuwien.ac.at

Abstract. Recently, liberally domain-expansion safe HEX-programs have been presented as a generalization of strongly safe HEX-programs that enlarges the scope of effective applicability. While such programs can be finitely grounded, a concrete grounding algorithm that is practically useful remained open. In this paper, we present such an algorithm and show how to integrate it into the model-building framework for HEX-programs, which is extended for this purpose. While traditional HEX-evaluation relies on program decomposition for grounding, our new algorithm can directly ground any liberally domain-expansion safe program without decomposition. However, as splitting is still sometimes useful for performance reasons, we develop a new decomposition heuristics that aims at maximizing efficiency. An experimental evaluation confirms the practicability of our approach.

1 Introduction

HEX-programs [6] are declarative logic programs that enrich Answer Set Programming (ASP) by so-called external atoms. They provide a means to couple any external computation or data source with a logic program: intuitively, information from the program, given by predicate extensions, is passed to an external source which returns output values of an (abstract) function that it computes. This extension has been motivated by emerging needs such as accessing distributed information, context awareness, complex or specific data structures, etc. Widening the application range of ASP with its systems like SMODELS, DLV and CLASP, HEX programs and the DLVHEX solver have enabled challenging applications such as querying data and ontologies on the Web, multi-context reasoning, e-government, and more (cf. [3]). In that, external atoms proved to be a valuable construct of high expressivity, which enables recursive data exchange between the program and external sources and, via a modular software plugin architecture, is convenient to realize customized data access, adding built-ins or to process specific datatypes.

A characteristic feature of HEX-programs is that new values (not occurring in the program) might arise by external source access. For example, an atom $\&concat[ab, c](Y)$ that intuitively appends c to ab, returns in Y the string abc. However, admitting such a behavior, so called *value invention*, poses severe challenges to grounding a respective HEX-program prior to solving, as common in ASP systems. It is intuitively clear that in

* This research has been supported by the Austrian Science Fund (FWF) project P20840, P20841, P24090, and by the Vienna Science and Technology Fund (WWTF) project ICT08-020.

general it is impossible to predetermine the "relevant" domain, like in the example above when $concat$ bears no meaning, and may be practically infeasible (even if it is finite). Imposing *strong safety* [7] amounts to disallowing value invention, which prevents the natural usage of even simple external atoms (like $concat$ above). On the other hand, to adopt standard safety conditions and to either perform a pre-evaluation of external atoms (as, e.g., for lua [9]), or to request the user to provide domain predicates, is likewise an unsatisfactory treatment of value invention.

To remedy the situation, strong safety has been recently relaxed [5] to yield *liberally domain-expansion safe* HEX-programs, which are still finitely groundable and more general than various other safety notions in the literature, e.g., VI-programs [1] and ω-restricted programs [14]. However, two important issues remained unresolved. First, to provide a concrete grounding algorithm for liberally domain-expansion safe HEX-programs that can efficiently produce an (equivalent) finite ground program. Second, to suitably integrate this algorithm into the existing HEX evaluation framework, in which a program is decomposed (exploiting a generalized splitting theorem [3]) into *evaluation units* that are grounded and solved separately; this needs to be respected by the grounder.

In this work we tackle these issues providing the following contributions:

- We introduce a grounding algorithm for the recently defined class of *(liberally) domain-expansion safe (de-safe)* HEX*-programs* [5]. Roughly speaking, the algorithm is based on (optimized) iterative grounding using a guess-and-check approach, which first computes a partial grounding and then checks its sufficiency for answer set computation.
- We integrate the new grounding algorithm into the existing evaluation framework for HEX-programs. To this end, we generalize the modular decomposition underlying the model-building process to arbitrary liberally de-safe HEX programs as evaluation units.
- A new evaluation heuristics for the framework aims at dealing with the opposing goals of, on the one hand, larger units to exploit learning during evaluation (cf. [4]), and on the other hand, splitting units for more efficient grounding. It greedily merges evaluation units unless efficient grounding requires a split, in preference of learning.
- We present an experimental evaluation of an implementation our algorithm on synthetic and application-driven benchmarks, which witnesses significant improvements.

An extended version of the paper, which includes proofs, is available at http://www.kr.tuwien.ac.at/staff/redl/grounding/groundingext.pdf.

2 Preliminaries

HEX-programs are built over mutually disjoint sets $\mathcal{P}$ of ordinary predicates, $\mathcal{X}$ of external predicates, $\mathcal{C}$ of constants, and $\mathcal{V}$ of variables. In accordance with [10, 4], a *(signed) ground literal* is a positive or a negative formula $\mathbf{T}a$ resp. $\mathbf{F}a$, where a is a ground atom of form $p(c_1, \ldots, c_\ell)$, with predicate $p \in \mathcal{P}$ and constants $c_1, \ldots, c_\ell \in \mathcal{C}$, abbreviated $p(\mathbf{c})$. For a ground literal $\sigma = \mathbf{T}a$ or $\sigma = \mathbf{F}a$, let $\overline{\sigma}$ denote its opposite, i.e., $\overline{\mathbf{T}a} = \mathbf{F}a$ and $\overline{\mathbf{F}a} = \mathbf{T}a$. An *assignment* $\mathbf{A}$ is a consistent set of literals $\mathbf{T}a$ or $\mathbf{F}a$, where $\mathbf{T}a$ expresses that a is true and $\mathbf{F}a$ that a is false. An *interpretation* is a complete (maximal) assignment $\mathbf{A}$, also identified by the set of true atoms $\mathbf{TA} = \{a \mid \mathbf{T}a \in \mathbf{A}\}$.

HEX-Program Syntax. HEX-programs are a generalization of (disjunctive) logic programs under the answer set semantics [11]; for details and background see [6].

An *external atom* is of the form $\&g[\mathbf{Y}](\mathbf{X})$, where $\mathbf{Y} = Y_1, \ldots, Y_\ell$ are input parameters with $Y_i \in \mathcal{P} \cup \mathcal{C} \cup \mathcal{V}$ for all $1 \leq i \leq \ell$, and $\mathbf{X} = X_1, \ldots, X_m$ are output terms with $X_i \in \mathcal{C} \cup \mathcal{V}$ for all $1 \leq i \leq m$. Moreover, we assume that the input parameters of every external predicate $\&g \in \mathcal{X}$ are typed such that $type(\&g, i) \in \{\mathbf{const}, \mathbf{pred}\}$ for every $1 \leq i \leq \ell$. We make also the restriction that $Y_i \in \mathcal{P}$ if $type(\&g, i) = \mathbf{pred}$ and $X_i \in \mathcal{C} \cup \mathcal{V}$ otherwise. For an ordinary predicate $p \in \mathcal{P}$, let $ar(p)$ denote the arity of p and for an external predicate $\&g \in \mathcal{X}$, let $iar(\&g)$ denote the input arity and $oar(\&g)$ the output arity of $\&g$.

A HEX-program consists of rules

$$a_1 \vee \cdots \vee a_k \leftarrow b_1, \ldots, b_m, \text{not } b_{m+1}, \ldots, \text{not } b_n \ , \tag{1}$$

where each a_i is an (ordinary) atom $p(X_1, \ldots, X_\ell)$ with $X_i \in \mathcal{C} \cup \mathcal{V}$ for all $1 \leq i \leq \ell$, each b_j is either an ordinary atom or an external atom, and $k + n > 0$.

The *head* of a rule r is $H(r) = \{a_1, \ldots, a_n\}$ and the *body* is $B(r) = \{b_1, \ldots, b_m, \text{not } b_{m+1}, \ldots, \text{not } b_n\}$. We call b or $\text{not } b$ in a rule body a *default literal*; $B^+(r) = \{b_1, \ldots, b_m\}$ is the *positive body*, $B^-(r) = \{b_{m+1}, \ldots, b_n\}$ is the *negative body*. For a program Π (a rule r), let $A(\Pi)$ ($A(r)$) be the set of all ordinary atoms and $EA(\Pi)$ ($EA(r)$) be the set of all external atoms occurring in Π (in r).

HEX-**Program Semantics**. The semantics of a ground external atom $\&g[\mathbf{p}](\mathbf{c})$ wrt. an interpretation, i.e., a complete assignment, $\mathbf{A}$ is given by the value of a $1+k+l$-ary Boolean-valued *oracle function*, denoted by $f_{\&g}$, that is defined for all possible values of $\mathbf{A}$, $\mathbf{p}$ and $\mathbf{c}$. We make the restriction that for given $\mathbf{A}$, $\mathbf{p}$, set $\{\mathbf{x} \mid f_{\&g}(\mathbf{A}, \mathbf{p}, \mathbf{x}) = 1\}$ is computable. An input predicate p of an external predicate with input list $\&g[\mathbf{p}]$ is *monotonic* (*antimonotonic*), iff $f_{\&g}(\mathbf{A}, \mathbf{p}, \mathbf{c}) = 1$ implies $f_{\&g}(\mathbf{A}', \mathbf{p}, \mathbf{c}) = 1$ ($f_{\&g}(\mathbf{A}, \mathbf{p}, \mathbf{c}) = 0$ implies $f_{\&g}(\mathbf{A}', \mathbf{p}, \mathbf{c}) = 0$) for all $\mathbf{A}'$ s.t. $ext(p, \mathbf{A}') \supseteq ext(p, \mathbf{A})$ and $ext(q, \mathbf{A}') = ext(q, \mathbf{A})$ for $q \in \mathbf{p}$ and $q \neq p$. It is *nonmonotonic* iff it is neither monotonic nor antimonotonic. We denote this by $\mathbf{Y}_m$, $\mathbf{Y}_a$, $\mathbf{Y}_n$ the predicates from $\mathbf{Y}$ which are monotonic, antimonotonic and nonmonotonic, respectively. Satisfaction of ground ordinary rules and ASP programs [11] is then extended to HEX-rules and programs in the obvious way.

Non-ground programs are handled by grounding as usual. The *grounding* $grnd_C(r)$ of a rule r wrt. $C \subseteq \mathcal{C}$ is the set of all rules $\{\sigma(r) \mid \sigma : \mathcal{V} \mapsto C\}$, where σ is a *grounding substitution* mapping each variable to a constant, and $\sigma(r)$ denotes the rule which results if each variable X in r is replaced by $\sigma(X)$. The *grounding of a program* Π wrt. C (respectively $\mathcal{C}$ if not mentioned explicitly) is defined as $grnd_C(\Pi) = \bigcup_{r \in \Pi} grnd_C(r)$. The set of constants appearing in a program Π is denoted C_Π.

An answer set of a program Π is a model $\mathbf{A}$ of the FLP-reduct [8] $fgrnd_{\mathcal{C}}(\Pi)^{\mathbf{A}} = \{r \in grnd_{\mathcal{C}}(\Pi) \mid \mathbf{A} \models B(r)\}$ such that $\mathbf{TA}$ is subset-minimal, i.e., no $\mathbf{A}'$ with $\mathbf{TA}' \subsetneq \mathbf{TA}$ is a model. We denote the set of all answer sets of a program Π by $\mathcal{AS}(\Pi)$, and write $\Pi \equiv \Pi'$ if $\{\mathbf{TA} \mid \mathbf{A} \in \mathcal{AS}(\Pi)\} = \{\mathbf{TA} \mid \mathbf{A} \in \mathcal{AS}(\Pi')\}$.

Example 1. Consider program $\Pi = \{p \leftarrow \&id[p]()\}$, where $\&id[p]()$ is true iff p is true. Π has the unique answer set $\mathbf{A}_1 = \emptyset$, which is a subset-minimal model of $fgrnd_{\mathcal{C}}(\Pi)^{\mathbf{A}_1} = \emptyset$.

Safety. In general, the set $\mathcal{C}$ contains constants that do not occur in the program Π and $\mathcal{C}$ can even be infinite (e.g., the set of all strings). Therefore, safety criteria are adopted which guarantee the existence of a finite portion $\Pi' \subseteq grnd_{\mathcal{C}}(\Pi)$ (also called *finite*

grounding of Π; usually by restricting to a finite $C \subseteq \mathcal{C}$) that has the same answer sets as Π. A program is *safe*, if all rules r are *safe*, i.e., every variable in r is *safe* in the sense that it occurs either in an ordinary atom in $B^+(r)$, or in the output list $\mathbf{X}$ of an external atom $\&g[\mathbf{Y}](\mathbf{X})$ in $B^+(r)$ where all variables in $\mathbf{Y}$ are safe. However, this notion is not sufficient, as the following example shows.

Example 2. Let $\Pi = \{s(a);\ t(Y) \leftarrow s(X), \&concat[X, a](Y); s(X) \leftarrow t(X), d(X)\}$, where $\&concat[X, a](Y)$ is true iff Y is the string concatenation of X and a. Then Π is safe but $\&concat[X, a](Y)$ can introduce infinitely many constants. □

Therefore, *strong safety* was introduced in [7], which ensures that the output of cyclic external atoms is limited. This notion was recently relaxed to *(liberal) domain-expansion safety (de-safety)* [5], on which we focus here. It is based on *term bounding functions*, which intuitively declare terms in rules as *bounded*, if there are only finitely many substitutions for this term in a *canonical grounding* $CG(\Pi)$ of Π.[1] This grounding is infinite in general and serves to define liberal de-safe HEX-programs; for such programs, however, $CG(\Pi)$ is finite. In this paper, we present an algorithm for efficient construction of a concrete finite ground program that is equivalent to $CG(\Pi)$ (and thus to Π).

Definition 1 (Term Bounding Function (TBF)). *A TBF $b(\Pi, r, S, B)$ maps a program Π, a rule $r \in \Pi$, a set S of already safe attributes, and a set B of already bounded terms in r to an enlarged set $b(\Pi, r, S, B) \supseteq B$ of bounded terms, s.t. every $t \in b(\Pi, r, S, B)$ has finitely many substitutions in $CG(\Pi)$ if (i) the attributes S have a finite range in $CG(\Pi)$ and (ii) each term in $terms(r) \cap B$ has finitely many substitutions in $CG(\Pi)$.*

Liberal domain-expansion safety of programs is then parameterized with a term bounding function, such that concrete syntactic and/or semantic properties can be plugged in; concrete term bounding functions are described in [5]. The concept is defined in terms of domain-expansion safe attributes $S_\infty(\Pi)$, which are stepwise identified as $S_n(\Pi)$ in mutual recursion with bounded terms $B_n(r, \Pi, b)$ of rules r in Π.

Definition 2 ((Liberal) Domain-expansion Safety). *Given a TBF b, the set of* bounded terms $B_n(r, \Pi, b)$ *in step $n \geq 1$ in a rule $r \in \Pi$ is $B_n(r, \Pi, b) = \bigcup_{j \geq 0} B_{n,j}(r, \Pi, b)$ where $B_{n,0}(r, \Pi, b) = \emptyset$ and for $j \geq 0$, $B_{n,j+1}(r, \Pi, b) = b(\Pi, r, S_{n-1}(\Pi), B_{n,j})$.*

The set of domain-expansion safe attributes $S_\infty(\Pi) = \bigcup_{n \geq 0} S_n(\Pi)$ *of a program Π is iteratively constructed with $S_0(\Pi) = \emptyset$ and for $n \geq 0$:*

- $p\restriction i \in S_{n+1}(\Pi)$ *if for each $r \in \Pi$ and atom $p(t_1, \ldots, t_{ar(p)}) \in H(r)$, it holds that $t_i \in B_{n+1}(r, \Pi, b)$, i.e., t_i is* bounded*;*
- $\&g[\mathbf{Y}]_r\restriction_{\text{I}} i \in S_{n+1}(\Pi)$ *if each $\mathbf{Y}_i$ is a* bounded *variable, or $\mathbf{Y}_i$ is a predicate input parameter p and $p\restriction 1, \ldots, p\restriction ar(p) \in S_n(\Pi)$;*
- $\&g[\mathbf{Y}]_r\restriction_{\text{O}} i \in S_{n+1}(\Pi)$ *if and only if r contains an external atom $\&g[\mathbf{Y}](\mathbf{Y})$ such that $\mathbf{Y}_i$ is bounded, or $\&g[\mathbf{Y}]_r\restriction_{\text{I}} 1, \ldots, \&g[\mathbf{Y}]_r\restriction_{\text{I}} ar_{\text{I}}(\&g) \in S_n(\Pi)$.*

A program Π is (liberally) de-safe*, if it is safe and all its attributes are de-safe.*

[1] $CG(\Pi)$ is least fixed point $G_\Pi^\infty(\emptyset)$ of a monotone operator $G_\Pi(\Pi') = \bigcup_{r \in \Pi} \{r' \mid r' \in grnd_{\mathcal{C}}(r), \exists \mathbf{A} \subseteq \mathcal{A}(\Pi'), \mathbf{A} \not\models \bot, \mathbf{A} \models B^+(r')\}$ on programs Π' [5], where $\mathcal{A}(\Pi')$ denotes the set of all atoms in Π'.

Example 3. The program Π from Example 2 is liberally de-safe as infinitely many constants are prevented by domain predicate $d(X)$ in the last rule. □

As shown in [5], every de-safe HEX-program has a finite grounding with the same answer sets as the original program. This result holds for *every* TBF, because the preconditions of a TBF force it to be sufficiently strong.

For further explanation and discussion of liberal de-safety, and for an analysis showing that the concept subsumes a number of other notions of safety we refer to [5].

3 Grounding Liberally Domain-expansion Safe HEX-Programs

In this section we present a grounding algorithm for *liberally domain-expansion safe* HEX*-programs* as introduced in [5]. It is based on the following idea. Iteratively ground the input program and then check if the grounding contains all relevant ground rules. The check works by evaluating external sources under relevant interpretations and testing if they introduce any new values which were not respected in the grounding. If this is the case, then the set of constants is expanded and the program is grounded again. If the check does not identify additional constants which must be respected in the grounding, then it is guaranteed that the unrespected constants from $\mathcal{C}$ are irrelevant in order to ensure that the grounding has the same answer sets as the original program. For liberally domain-expansion safe programs, this procedure will eventually reach a fixpoint, i.e., all relevant constants are respected in the grounding.

We start with some basic concepts which are all demonstrated in Example 4. We assume that rules are standardized apart (i.e., have no variables in common). Let R be a set of external atoms and let r be a rule. By $r|_R$ we denote the rule obtained by removing external atoms not in R, i.e., such that $H(r|_R) = H(r)$ and $B^s(r|_R) = ((B^s(r) \cap A(r)) \cup (B^s(r) \cap R))$ for $s \in \{+, -\}$. Similarly, $\Pi|_R = \bigcup_{r \in \Pi} r|_R$, for a program Π. Furthermore, let $var(r)$ be the set of variables from $\mathcal{V}$ appearing in a rule r.

Definition 3 (Liberal Domain-expansion Safety Relevance). *A set R of external atoms is* relevant for liberal de-safety *of a program Π, if $\Pi|_R$ is liberally de-safe and $var(r) = var(r|_R)$, for all $r \in \Pi$.*

Intuitively, if an external atom is not relevant, then it cannot introduce new constants. Note that for a program, the set of de-safe relevant external atoms is not necessarily unique, leaving room for heuristics. In the following definitions we choose a specific set.

We further need the concepts of input auxiliary and external atom guessing rules. We say that an external atom $\&g[\mathbf{Y}](\mathbf{X})$ *joins* an atom b, if some variable from $\mathbf{Y}$ occurs in b, where in case b is an external atom the occurrence is in the output list of b.

Definition 4 (Input Auxiliary Rule). *Let Π be a* HEX*-program, and let $\&g[\mathbf{Y}](\mathbf{X})$ be some external atom with input list $\mathbf{Y}$ occurring in a rule $r \in \Pi$. Then, for each such atom, a rule $r_{inp}^{\&g[\mathbf{Y}](\mathbf{X})}$ is composed as follows:*

- *The head is $H(r_{inp}^{\&g[\mathbf{Y}](\mathbf{X})}) = \{g_{inp}(\mathbf{Y})\}$, where g_{inp} is a fresh predicate; and*
- *The body $B(r_{inp}^{\&g[\mathbf{Y}](\mathbf{X})})$ contains each $b \in B^+(r) \backslash \{\&g[\mathbf{Y}](\mathbf{X})\}$ such that $\&g[\mathbf{Y}](\mathbf{X})$ joins b, and b is de-safety-relevant if it is an external atom.*

Intuitively, input auxiliary rules are used to derive all ground tuples $\mathbf{y}$ under which the external atom needs to be evaluated. Next, we need *external atom guessing rules*.

Definition 5 (External Atom Guessing Rule). *Let Π be a* HEX*-program, and let $\&g[\mathbf{Y}](\mathbf{X})$ be some external atom. Then a rule $r_{guess}^{\&g[\mathbf{Y}](\mathbf{X})}$ is composed as follows:*

- *The head is $H(r_{guess}^{\&g[\mathbf{Y}](\mathbf{X})}) = \{e_{r,\&g[\mathbf{Y}]}(\mathbf{X}), ne_{r,\&g[\mathbf{Y}]}(\mathbf{X})\}$*
- *The body $B(r_{guess}^{\&g[\mathbf{Y}](\mathbf{X})})$ contains*
  - *(i) each $b \in B^+(r) \setminus \{\&g[\mathbf{Y}](\mathbf{X})\}$ such that $\&g[\mathbf{Y}](\mathbf{X})$ joins b and b is de-safety-relevant if it is an external atom; and*
  - *(ii) $g_{inp}(\mathbf{Y})$.*

Intuitively, they guess the truth value of external atoms using a choice between the *external replacement atom* $e_{r,\&g[\mathbf{Y}]}(\mathbf{X})$, and fresh atom $ne_{r,\&g[\mathbf{Y}]}(\mathbf{X})$.

Our approach is based on a grounder for ordinary ASP programs. Compared to the naive grounding $grnd_{\mathcal{C}}(\Pi)$, which substitutes all constants for all variables in all possible ways, we allow the ASP grounder GroundASP to optimize rules such that, intuitively, rules may be eliminated if their body is always false, and ordinary body literals may be removed from the grounding if they are always true, as long as this does not change the answer sets.

Definition 6. *We call rule r' an* o-strengthening *of r, if $H(r') = H(r)$, $B(r') \subseteq B(r)$ and $B(r) \setminus B(r')$ contains only ordinary literals, i.e., no external atom replacements.*

Definition 7. *An algorithm GroundASP is a* faithful ASP grounder *for a safe ordinary program Π, if it outputs an equivalent ground program Π' such that*

- *Π' consists of o-strengthenings of rules in $grnd_{C_\Pi}(\Pi)$;*
- *if $r \in grnd_{C_\Pi}(\Pi)$ has no o-strengthening in Π', then every answer set of $grnd_{C_\Pi}(\Pi)$ falsifies some ordinary literal in $B(r)$; and*
- *if $r \in grnd_{C_\Pi}(\Pi)$ has some o-strengthening $r' \in \Pi'$, then every answer set of $grnd_{C_\Pi}(\Pi)$ satisfies $B(r) \setminus B(r')$.*

The formalization of the algorithm is shown in Algorithm GroundHEX. Our naming convention is as follows. Program Π is the non-ground input program. Program Π_p is the non-ground ordinary ASP *prototype program*, which is an iteratively updated extension of Π with additional rules. In each step, the *preliminary ground program* Π_{pg} is produced by grounding Π_p using a standard ASP grounding algorithm. Program Π_{pg} converges against a fixpoint from which the final *ground* HEX*-program* Π_g is extracted.

The algorithm first chooses a set of de-safety relevant external atoms, e.g., all external atoms as a naive and conservative approach or following a greedy approach as in our implementation, and then introduces input auxiliary rules $r_{inp}^{\&g[\mathbf{Y}](\mathbf{X})}$ for every external atom $\&g[\mathbf{Y}](\mathbf{X})$ in a rule r in Π in Part (a). For all non-relevant external atoms, we introduce external atom guessing rules which ensure that the ground instances of these external atoms are introduced in the grounding, even if we do not explicitly add them. Then, all external atoms $\&g[\mathbf{Y}](\mathbf{X})$ in all rules r in Π_p are replaced by ordinary *replacement atoms* $e_{r,\&g[\mathbf{Y}]}(\mathbf{X})$. This allows the algorithm to use a faithful ASP grounder GroundASP in the main loop at (b). After the grounding step, the algorithm checks

Algorithm GroundHEX

Input: A liberally de-safe HEX-program Π
Output: A ground HEX-program Π_g s.t. $\Pi_g \equiv \Pi$
(a) Choose a set R of *de-safety-relevant* external atoms in Π
$\Pi_p := \Pi \cup \{r_{inp}^{\&g[\mathbf{Y}](\mathbf{X})} \mid \&g[\mathbf{Y}](\mathbf{X}) \text{ in } r \in \Pi\} \cup \{r_{guess}^{\&g[\mathbf{Y}](\mathbf{X})} \mid \&g[\mathbf{Y}](\mathbf{X}) \notin R\}$
Replace all external atoms $\&g[\mathbf{Y}](\mathbf{X})$ in all rules r in Π_p by $e_{r,\&g[\mathbf{Y}]}(\mathbf{X})$
(b) **repeat**
  $\Pi_{pg} :=$ GroundASP(Π_p) /* partial grounding */
  /* evaluate all de-safety-relevant external atoms */
(c) **for** $\&g[\mathbf{Y}](\mathbf{X}) \in R$ *in a rule* $r \in \Pi$ **do**
    $\mathbf{A}_{ma} := \{\mathbf{T}p(\mathbf{c}) \mid a(\mathbf{c}) \in A(\Pi_{pg}), p \in \mathbf{Y}_m\} \cup \{\mathbf{F}p(\mathbf{c}) \mid a(\mathbf{c}) \in A(\Pi_{pg}), p \in \mathbf{Y}_a\}$
    /* do this under all relevant assignments */
(d) **for** $\mathbf{A}_{nm} \subseteq \{\mathbf{T}p(\mathbf{c}), \mathbf{F}p(\mathbf{c}) \mid p(\mathbf{c}) \in A(\Pi_{pg}), p \in \mathbf{Y}_n\}$ *s.t.* $\nexists a : \mathbf{T}a, \mathbf{F}a \in \mathbf{A}_{nm}$ **do**
      $\mathbf{A} := (\mathbf{A}_{ma} \cup \mathbf{A}_{nm} \cup \{\mathbf{T}a \mid a \leftarrow\in \Pi_{pg}\}) \setminus \{\mathbf{F}a \mid a \leftarrow\in \Pi_{pg}\}$
(e) **for** $\mathbf{y} \in \{\mathbf{c} \mid r_{inp}^{\&g[\mathbf{Y}](\mathbf{X})}(\mathbf{c}) \in A(\Pi_{pg})\}$ **do**
(f) Let $O = \{\mathbf{x} \mid f_{\&g}(\mathbf{A}, \mathbf{y}, \mathbf{x}) = 1\}$
      /* add the respective ground guessing rules */
      $\Pi_p := \Pi_p \cup \{e_{r,\&g[\mathbf{y}]}(\mathbf{x}) \vee ne_{r,\&g[\mathbf{y}]}(\mathbf{x}) \leftarrow\mid \mathbf{x} \in O\}$
until Π_{pg} *did not change*
(g) Remove input auxiliary rules and external atom guessing rules from Π_{pg}
Replace all $e_{\&g[\mathbf{y}]}(\mathbf{x})$ in Π by $\&g[\mathbf{y}](\mathbf{x})$
return Π_{pg}

if the grounding is large enough, i.e., if it contains all relevant constants. For this, it traverses all relevant external atoms at (c) and all relevant input tuples at (d) and at (e). Then, constants returned by external sources are added to Π_p at (f); if the constants were already respected, then this will have no effect. Thereafter the main loop starts over again. The algorithm will find a program which respects all relevant constants. It then removes auxiliary input rules and translates replacement atoms to external atoms at (g).

We illustrate our grounding algorithm with the following example.

Example 4. Let Π be the following program:

$$f_1: \ d(a). \quad f_2: \ d(b). \quad f_3: \ d(c). \qquad \begin{aligned} r_1: \ & s(Y) \leftarrow \&diff[d,n](Y), d(Y). \\ r_2: \ & n(Y) \leftarrow \&diff[d,s](Y), d(Y). \\ r_3: \ & c(Z) \leftarrow \&count[s](Z). \end{aligned}$$

Here, $\&diff[s_1, s_2](x)$ is true for all elements x, which are in the extension of s_1 but not in that of s_2, and $\&count[s](i)$ is true for the integer i corresponding to the number of elements in s. The program partitions the domain (extension of d) into two sets (extensions of s and n) and computes the size of s. The external atoms $\&diff[d,n](Y)$ and $\&diff[d,s](Y)$ are not relevant for de-safety. Π_p at the beginning of the first iteration is as follows (neglecting input auxiliary rules, which are facts). Let $e_1(Y)$, $e_2(Y)$ and $e_3(Z)$ be shorthands for $e_{r_1,\&diff[d,n]}(Y)$, $e_{r_2,\&diff[d,s]}(Y)$. and $e_{r_3,\&count[s]}(Z)$, respectively.

$$\begin{aligned} f_1: \ & d(a). \quad f_2: \ d(b). \quad f_3: \ d(c). & r_1: \ & s(Y) \leftarrow e_1(Y), d(Y). \\ g_1: \ & e_1(Y) \vee ne_1(Y) \leftarrow d(Y). & r_2: \ & n(Y) \leftarrow e_2(Y), d(Y). \\ g_2: \ & e_2(Y) \vee ne_2(Y) \leftarrow d(Y). & r_3: \ & c(Z) \leftarrow e_3(Z). \end{aligned}$$

The ground program Π_{pg} contains no instances of r_3 because the optimizer recognizes that $e_{r_3,\&count[s]}(Z)$ occurs in no rule head and no ground instance can be true in any answer set. Then the algorithm comes to the checking phase. It does not evaluate the external atoms in r_1 and r_2, because they are not relevant for de-

safety because of the domain predicate $d(Y)$. But it evaluates $\&count[s](Z)$ under all $\mathbf{A} \subseteq \{s(a), s(b), s(c)\}$ because the external atom is nonmonotonic in s. Then the algorithm adds rules $\{e_3(Z) \vee ne_3(Z) \leftarrow \mid Z \in \{0,1,2,3\}\}$ to Π_p. After the second iteration, the algorithm terminates. □

One can show that this algorithm is sound and complete:

Proposition 1. *If Π is a liberally de-safe* HEX*-program, then* GroundHEX*$(\Pi) \equiv \Pi$.*

4 Integrating the Algorithm into the Model-building Framework

The answer sets of a HEX-program Π are determined using a modular decomposition based on the concept of an *evaluation graph* $\mathcal{E}(V, E)$, whose nodes V are *evaluation units*, i.e. subsets of Π, that are acyclically connected by edges $E = \rightarrow_m \cup \rightarrow_n$ that are inherited from an underlying *dependency graph* $G = \langle \Pi, \rightarrow_m \cup \rightarrow_n \rangle$, where $\rightarrow_m$ captures monotonic and $\rightarrow_n$ nonmonotonic dependencies of the units resp. rules [3].

The evaluation proceeds then unit by unit along the structure of the evaluation graph bottom up. For a unit u, each union of answer sets of predecessor units of u, called an *input model of* u, is added as facts to the program at u. This extended program is grounded and solved; the resulting set of *output models of* u is sent to the successor units of u in the same way. The properties of evaluation graphs guarantee that the output models of a dedicated final unit correspond to the answer sets of the whole program.

In order to ground the units before evaluation using a grounding algorithm for ordinary ASP, each unit in the evaluation graph must be from the class of *extended pre-groundable* HEX*-programs*, which is a proper subset of all strongly safe HEX programs. It was shown in [13] that every strongly safe HEX-program possesses at least one evaluation graph, i.e., the program can be decomposed into extended pre-groundable HEX-programs.

The motivation for the evaluation framework in [3] was mainly performance enhancement. However, as not every strongly safe program is extended pre-groundable, program decomposition is in some cases *indispensable* for program evaluation. This is in contrast to the grounding algorithm introduced in this paper, which can directly ground any liberally de-safe, and thus strongly safe, program.

Example 5. Program Π from Example 4 cannot be grounded by the traditional HEX algorithms as it is not extended pre-groundable. Instead, it needs to be partitioned into two units $u_1 = \{f_1, f_2, f_3, r_1, r_2\}$ and $u_2 = \{r_3\}$ with $u_1 \rightarrow_n u_2$. Now u_1 and u_2 are extended pre-groundable HEX-programs. Then the answer sets of u_1 must be computed before u_2 can be grounded. Our algorithm can ground the whole program immediately.□

Therefore, in contrast to the previous algorithms one can keep the whole program as a single unit, but also still apply decomposition with liberally de-safe programs as units. To this end, we define a *generalized evaluation graph* like an evaluation graph in [3], but with de-safe instead of extended pre-groundable programs as nodes. We can then show that the algorithm BUILDANSWERSETS in [3] remains sound and complete for generalized evaluation graphs, if the grounding algorithm from above is applied:

Proposition 2. *For a generalized evaluation graph $\mathcal{E} = (V, E)$ of a de-safe* HEX*-program Π,* BUILDANSWERSETS *with* GroundHEX *for grounding returns $\mathcal{AS}(\Pi)$.*

Algorithm GreedyGEG

Input: A liberally de-safe HEX-program Π
Output: A generalized evaluation graph $\mathcal{E} = \langle V, E \rangle$ for Π
Let V be the set of (subset-maximal) strongly connected components of $G = \langle \Pi, \rightarrow_m \cup \rightarrow_n \rangle$
Update E
while V *was modified* **do**
  for $u_1, u_2 \in V$ *such that* $u_1 \neq u_2$ **do**
(a) **if** *there is no indirect path from* u_1 *to* u_2 *(via some* $u' \neq u_1, u_2$*) or vice versa* **then**
(b) **if** *no de-relevant* $\&g[\mathbf{y}](\mathbf{x})$ *in some* u_2 *has a nonmonotonic predicate input from* u_1 **then**
        $V := (V \setminus \{u_1, u_2\}) \cup \{u_1 \cup u_2\}$
        Update E
return $\mathcal{E} = \langle V, E \rangle$

While program decomposition led to performance increase for the solving algorithms from [3], it is counterproductive for new learning-based algorithms [4] because learned knowledge cannot be effectively reused. In guess-and-check ASP programs, existing heuristics for evaluation graph generation frequently even split the guessing from the checking part, which is derogatory to the learning. Thus, from this perspective is advantageous to have few units. However, for the grounding algorithm a worst case is that a unit contains an external atom that is relevant for de-safety and receives nonmonotonic input from the same unit. In this case it needs to consider exponentially many assignments.

Example 6. Reconsider program Π from Example 4. Then the algorithm evaluates $\&count[s](Z)$ under all $\mathbf{A} \subseteq \{s(a), s(b), s(c)\}$ because it is nonmonotonic and de-safety-relevant. Now assume that the program contains the additional constraint

$$c_1 : \quad \leftarrow s(X), s(Y), s(Z), X \neq Y, X \neq Z, Y \neq Z \ ,$$

i.e., no more than two elements can be in set s. Then the algorithm would still check all $\mathbf{A} \subseteq \{s(a), s(b), s(c)\}$, but it is clear that the subset with three elements, which introduces the constant 3, is irrelevant because this interpretation will never occur in an answer set. If the program is split into units $u_1 = \{f, r_1, r_2, c_1\}$ and $u_2 = \{r_3\}$ with $u_2 \rightarrow_n u_1$, then $\{s(a), s(b), s(c)\}$ does not occur as an answer set of u_1. Thus, u_2 never receives this interpretation as input and never is evaluated under this interpretation. □

Algorithm GroundHEX evaluates the external sources under all interpretations such that the set of observed constants is maximized. While monotonic and antimonotonic input atoms are not problematic (the algorithm can simply set all to true resp. false), nonmonotonic parameters require an exponential number of evaluations. Thus, in such cases program decomposition is still useful as it restricts grounding to those interpretations which are actually relevant in some answer set. Program decomposition can be seen as a hybrid between traditional and lazy grounding [12], as program parts are instantiated which are larger than single rules but smaller than the whole program.

We thus introduce a heuristics in Algorithm GreedyGEG for generating a good generalized evaluation graph, which iteratively merges units. Condition (a) maintains acyclicity, while the condition at (b) deals with two opposing goals: (1) minimizing the number of units, and (2) splitting the program whenever a de-relevant nonmonotonic external atom would receive input from the same unit. It greedily gives preference to (1).

We illustrate the heuristics with an example.

Example 7. Reconsider program Π from Examples 4 and 6. Algorithm GreedyGEG creates a generalized evaluation graph with the two units $u_1 = \{f_1, f_2, f_3, r_1, r_2, c_1\}$ and $u_2 = \{r_3\}$ with $u_2 \rightarrow_n u_1$, which is as desired. □

It is not difficult to show that the heuristics yields a sound result.

Proposition 3. *For a liberally de-safe program Π, Algorithm GreedyGEG returns a suitable generalized evaluation graph of Π.*

5 Implementation and Evaluation

For implementing our technique, we integrated GRINGO as grounder GroundASP and CLASP into our prototype system DLVHEX[2]. We evaluated the implementation on a Linux server with two 12-core AMD 6176 SE CPUs with 128GB RAM. For this we use five benchmarks and present the total wall clock runtime (wt), the grounding time (gt) and the solving time (st). We possibly have $wt \neq gt + st$ because wt includes also computations other than grounding and solving (e.g., passing models through the evaluation graph). For determining de-safety relevant external atoms, our implementation follows a greedy strategy and tries to identify as many external atoms as irrelevant as possible. Detailed benchmark results are available at http://www.kr.tuwien.ac.at/staff/redl/grounding/benchmarks.ods.

Reachability. We consider reachability, where the edge relation is provided as an external atom $\&out[X](Y)$ delivering all nodes Y that are directly reached from a node X. The traditional implementation imports all nodes into the program and then uses domain predicates. An alternative is to query outgoing edges of nodes on-the-fly, which needs no domain predicates. This benchmark is motivated by route planning applications, where importing the full map might be infeasible due to the amount of data.

The results are shown in Table 1a. We use random graphs with a node count from 5 to 70 and an edge probability of 0.25. For each count, we average over 10 instances. Here we can observe that the encoding without domain predicates is more efficient in all cases because only a small part of the map is active in the logic program, which does not only lead to a smaller grounding, but also to a smaller search space during solving.

Set Partitioning. In this benchmark we consider a program similar to Example 4, which implements for each domain element x a choice from $sel(x)$ and $nsel(x)$ by an external atom, i.e., a partitioning of the domain into two subsets.

The domain predicate $domain$ is not necessary with de-safety because $\&diff$ does not introduce new constants. The effect of removing it is presented in Table 1b. Since $\&diff$ is monotonic in the first parameter and antimonotonic in the second, the measured overhead is small in the grounding step. Although the ground programs of the strongly safe and the liberally safe variants of the program are identical, the solving step is slower in the latter case; we explain this with caching effects. Grounding liberally de-safe programs needs more memory than grounding strongly safe programs, which might have negative effects on the later solving step. However, the total slowdown is moderate.

[2] http://www.kr.tuwien.ac.at/research/systems/dlvhex

Table 1: Benchmark Results (in secs; "—" means timeout, set to 300 secs)

(a) Reachability

#	w. domain predicates			w/o domain predicates		
	wall clock	ground	solve	wall clock	ground	solve
15	0.59	0.28	0.08	0.49	0.23	0.06
25	5.78	4.67	0.33	2.94	1.90	0.35
35	36.99	33.99	1.00	14.02	11.30	0.95
45	161.91	155.40	2.18	53.09	47.19	2.22
55	—	—	n/a	171.46	158.58	5.74
65	—	—	n/a	—	—	n/a

(b) Set Partitioning

#	w. domain predicates			w/o domain predicates		
	wall clock	ground	solve	wall clock	ground	solve
10	0.49	0.01	0.39	0.52	0.02	0.41
20	3.90	0.05	3.62	4.67	0.10	4.23
30	16.12	0.18	15.32	19.59	0.36	18.32
40	48.47	0.48	46.71	51.55	0.90	48.74
50	115.56	1.00	112.14	119.40	1.79	114.11
60	254.66	1.84	248.88	257.78	3.35	248.51

(c) Bird-penguin

#	w. domain predicates			w/o domain predicates		
	wall clock	ground	solve	wall clock	ground	solve
5	0.06	<0.005	0.01	0.08	0.02	0.01
10	0.14	<0.005	0.08	1.32	1.12	0.10
11	0.27	<0.005	0.19	2.85	2.43	0.27
12	0.32	<0.005	0.23	6.05	5.53	0.26
13	0.69	0.01	0.60	12.70	11.76	0.61
14	0.66	<0.005	0.57	28.17	26.70	0.73
15	1.66	0.01	1.49	59.73	57.14	1.46
16	1.69	0.01	1.53	139.47	131.87	1.92
17	3.83	0.01	3.57	—	—	n/a
18	4.34	0.01	4.08	—	—	n/a
19	10.07	0.01	9.56	—	—	n/a
20	11.36	0.01	10.87	—	—	n/a
24	95.60	0.01	93.35	—	—	n/a
25	—	0.01	—	—	—	n/a

(d) Merge Sort

#	w. domain predicates			w/o domain predicates		
	wall clock	ground	solve	wall clock	ground	solve
5	0.22	0.04	0.10	0.10	0.01	0.04
6	1.11	0.33	0.54	0.10	0.01	0.04
7	9.84	4.02	4.42	0.11	0.01	0.05
8	115.69	61.97	42.30	0.12	0.01	0.05
9	—	—	n/a	0.14	0.01	0.07
10	—	—	n/a	0.15	0.08	0.01
15	—	—	n/a	0.23	0.14	0.01
20	—	—	n/a	0.47	0.35	0.02
25	—	—	n/a	1.90	1.58	0.06
30	—	—	n/a	4.11	3.50	0.12
35	—	—	n/a	20.98	18.45	0.51
40	—	—	n/a	61.94	54.62	1.46
45	—	—	n/a	144.22	133.99	2.26
50	—	—	n/a	—	—	n/a

(e) Argumentation

#	monolithic			greedy		
	wall clock	ground	solve	wall clock	ground	solve
4	0.57	0.11	0.38	0.25	0.01	0.18
5	2.12	0.67	1.26	0.44	0.01	0.37
6	18.93	7.45	10.86	0.88	0.01	0.80
7	237.09	170.12	65.12	1.65	0.01	1.57
8	—	—	n/a	3.13	0.01	3.05
9	—	—	n/a	7.41	0.02	7.31

#	monolithic			greedy		
	wall clock	ground	solve	wall clock	ground	solve
10	—	—	n/a	15.92	0.02	15.81
11	—	—	n/a	31.19	0.02	31.05
12	—	—	n/a	63.16	0.02	62.95
13	—	—	n/a	172.75	0.03	172.38
14	—	—	n/a	256.60	0.01	256.44
15	—	—	n/a	290.01	<0.005	290.00

Bird-Penguin. We consider now a scenario using the DL-plugin for DLVHEX, which integrates description logics (DL) knowledge bases and nonmonotonic logic programs. The DL-plugin allows to access an ontology. We consider the ontology on the right, which encodes that penguins are birds and do not fly, and the logic program on the left which implements the rule that birds fly unless the contrary is derivable.

DL-Program:

$$\mathit{birds}(X) \leftarrow DL[\mathit{Bird}](X).$$
$$\mathit{flies}(X) \leftarrow \mathit{birds}(X), \text{not } \mathit{neg\_flies}(X).$$
$$\mathit{neg\_flies}(X) \leftarrow \mathit{birds}(X), DL[\mathit{Flier} \uplus \mathit{flies}; \neg \mathit{Flier}](X).$$

Ontology:

$$\mathit{Flier} \sqsubseteq \neg \mathit{NonFlier}$$
$$\mathit{Penguin} \sqsubseteq \mathit{Bird}$$
$$\mathit{Penguin} \sqsubseteq \mathit{NonFlier}$$

Intuitively, $DL[\mathit{Flier} \uplus \mathit{flies}; \neg \mathit{Flier}]$ requests all individuals in $\neg \mathit{Flier}$ under the assumption that Flier is extended by the elements in the extension of flies.

The third rule uses the domain predicate $\mathit{birds}(X)$, which is necessary under strong safety conditions, but with liberal de-safety, we might drop it because finiteness is guaranteed by finiteness of the ontology. The results are shown in Table 1c. The DL-

atom in the third rule is nonmonotonic and appears in a cycle, which is the worst case which cannot be avoided by the greedy heuristics. The results show a slowdown for the encoding without domain predicates. It is mainly caused by the grounding, but also solving becomes slightly slower without domain predicates due to caching effects.

Recursive Processing of Data Structures. This benchmark shows how data structures can be recursively processed. As an example we implement the merge sort algorithm using external atoms for *splitting a list in half* and *merging two sorted lists*, where lists are encoded as constants consisting of elements and delimiters. However, this is only a showcase and performance cannot be compared to native merge sort implementations.

In order to implement the application with strong safety, one must manually add a domain predicate with the set of all instances of the data structures at hand as extension, e.g., the set of all permutations of the input list. This number is factorial in the input size and thus already unmanagable for very small instances. The problems are both due to grounding and solving. Similar problems arise with other recursive data structures when strong safety is required (e.g., trees, for the pushdown automaton from [5], where the domain is the set of all strings up to a certain length). However, only a small part of the domain will ever be relevant during computation, hence the new grounding algorithm for liberally de-safe programs performs quite well, as shown in Table 1d.

Argumentation. This benchmark demonstrates the advantage of our new *greedy heuristics*, which is compared to the evaluation without splitting (*monolithic*). We compute ideal set extensions for randomized instances of abstract argumentation frameworks [2] of different sizes. External atoms are used for checking candidate extensions. Additionally, we perform a processing of the arguments in each extension, e.g., by using an external atom for generating LaTeX code for the visualization of the framework and its extensions. Without program decomposition, this is the worst case for our grounding algorithm because the code generating atom is nonmonotonic and receives input from the same component. But then our grounding algorithm calls it for exponentially many extensions, although only few of them are actually extensions of the framework.

We use random instances with an argument count from 1 to 20, and an edge probability from $\{0.30, 0.45, 0.60\}$; we use 10 instances for each combination. We can observe that grounding the whole program in a single pass causes large programs wrt. grounding time and size. Since the grounding is larger, also the solving step takes much more time than with our new decomposition heuristics, which avoids the worst case, cf. Table 1e.

Summary. Our new grounding algorithm allows for grounding liberally de-safe programs. Instances that can be grounded by the traditional algorithm as well, usually require domain predicates to be manually added (often cumbersome and infeasible in practice, as for recursive data structures). Our algorithm does not only relieve the user from writing domain predicates, but in many cases also has a significantly better performance. Nonmonotonic external atoms might be problematic for our new algorithm. However, the worst case can mostly be avoided by our new decomposition heuristics.

6 Conclusion

In this paper we presented a new grounding algorithm for the recent class of liberally domain-expansion safe HEX-programs [5]. In contrast to previous grounding techniques

for HEX-programs, it can handle all such programs directly and does not rely on a program decomposition. This is an advantage, as splitting has negative effects for learning techniques introduced in [4]. However, in the worst case the new algorithm requires exponentially many calls to external sources to determine the relevant constants for grounding. We thus developed a novel heuristics for program evaluation that aims at avoiding this worst case while retaining the positive features of the new algorithm, and we have extended the current HEX evaluation framework for its use. An experimental evaluation of our implementation on synthetic and real applications shows a clear benefit.

Future work includes *refinements* of our algorithm and the heuristics. In particular, we plan to exploit *meta-information* about external sources to identify classes of programs that allow for a better grounding, and in particular reduce worst case inputs for our algorithm. Furthermore, ongoing work investigates logic programs with *existentially quantified variables* in rule heads, which might be realized by a variant of our algorithm.

References

1. Calimeri, F., Cozza, S., Ianni, G.: External Sources of Knowledge and Value Invention in Logic Programming. Ann. Math. Artif. Intell. 50(3–4), 333–361 (2007)
2. Dung, P.M.: On the acceptability of arguments and its fundamental role in nonmonotonic reasoning, logic programming and n-person games. Artif. Intell. 77(2), 321–357 (1995)
3. Eiter, T., Fink, M., Ianni, G., Krennwallner, T., Schüller, P.: Pushing efficient evaluation of HEX programs by modular decomposition. In: LPNMR'11. pp. 93–106. Springer (2011)
4. Eiter, T., Fink, M., Krennwallner, T., Redl, C.: Conflict-driven ASP solving with external sources. Theor. Pract. Log. Prog. 12(4-5), 659-679 (2012)
5. Eiter, T., Fink, M., Krennwallner, T., Redl, C.: Liberal Safety Criteria for HEX-Programs. In: AAAI'13. AAAI Press (2013), http://www.kr.tuwien.ac.at/staff/tkren/pub/2013/aaai2013-liberalsafety.pdf, to appear
6. Eiter, T., Ianni, G., Schindlauer, R., Tompits, H.: A Uniform Integration of Higher-Order Reasoning and External Evaluations in Answer-Set Programming. In: IJCAI'05. pp. 90–96. Professional Book Center (2005)
7. Eiter, T., Ianni, G., Schindlauer, R., Tompits, H.: Effective Integration of Declarative Rules with External Evaluations for Semantic-Web Reasoning. In: ESWC'06. pp. 273–287. Springer (2006)
8. Faber, W., Leone, N., Pfeifer, G.: Semantics and complexity of recursive aggregates in answer set programming. Artif. Intell. 175(1), 278–298 (2011)
9. Gebser, M., Kaminski, R., Kaufmann, B., Schaub, T.: Answer Set Solving in Practice. Morgan & Claypool Publishers (2012)
10. Gebser, M., Kaufmann, B., Schaub, T.: Conflict-driven answer set solving: From theory to practice. Artif. Intell. 187–188, 52–89 (2012)
11. Gelfond, M., Lifschitz, V.: Classical Negation in Logic Programs and Disjunctive Databases. New Generat. Comput. 9(3–4), 365–386 (1991)
12. Palù, A.D., Dovier, A., Pontelli, E., Rossi, G.: Gasp: Answer set programming with lazy grounding. Fund. Inform. 96(3), 297–322 (2009)
13. Schüller, P.: Inconsistency in Multi-Context Systems: Analysis and Efficient Evaluation. Dissertation, Vienna University of Technology, Vienna, Austria (August 2012)
14. Syrjänen, T.: Omega-restricted logic programs. In: LPNMR'01. pp. 267–279 (2001)

IDP^3: Combining Symbolic and Ground Reasoning for Model Generation

Broes De Cat*, Joachim Jansen, Gerda Janssens

Department of Computer Science, KU Leuven
broes.decat, joachim.jansen, gerda.janssens@cs.kuleuven.be

Abstract. IDP^3 is a knowledge-base system, offering a rich, declarative knowledge representation language, a range of inferences and built-in interaction with a procedural language. In this paper, we give an overview of the system and show how multiple inferences are combined to obtain state-of-the-art model generation.

Keywords: declarative modeling, model expansion, program transformation

1 Introduction

IDP^3 is a knowledge-base system (KBS) [22], a system[1] aimed at (1) allowing natural representation of knowledge over an application domain in a formal language (a *logic*), (2) offering a range of inferences to tackle diverse reasoning tasks, *reusing* the same knowledge as much as possible, (3) providing a procedural interface to combine multiple inferences to solve complex/compound tasks in the application domain. An example is a KBS storing knowledge about course scheduling at a university. By applying suitable forms of inference, schedules can be generated automatically at the start of the year, hand-made schedules can be verified, existing schedules can be revised, etc., all using the same knowledge.

As knowledge representation language, IDP^3 offers the logic FO$(\cdot)^{\text{IDP}}$ [8], a logic that extends full First-Order Logic (FO) with aggregate functions (sum, product, etc.), integer arithmetic, partial functions, a type system and inductive definitions. The logic is closely related to the languages developed in the fields of SAT Modulo Theories (SMT) [21] and Answer Set Programming (ASP) [18], to the Constraint Programming (CP) language Zinc[19] and the Alloy language[15]. A more formal comparison to ASP can be found in [7].

A number of different forms of inference are already available in the system (and more are part of current research), such as model generation (find (optimal) models of a logic theory), entailment, query evaluation and simulation of dynamic systems. Model expansion for FO$(\cdot)^{\text{IDP}}$ is closely related to answer set generation

* Broes De Cat is funded by the Institute for Innovation through Science and Technology Flanders (IWT).

[1] The name IDP stands for Imperative-Declarative Programming.

and to solving Constraint Satisfaction Problems (CSPs), for example in the systems Gringo[12]-Clasp[11], DLV[17], Comet[20] and Gecode[13].

The procedural interface is available through the language Lua.

In this paper, we present IDP^3, the newest version of the IDP system, which turned the model expansion system IDP^2 into a real KBS. We specifically show how IDP^3 works towards one of our main research goals, namely to reduce the dependence of inference performance on the way knowledge is specified. To that end, we zoom in on the model expansion inference in IDP^3 (Section 3), improving over IDP^2 by, among others, definition evaluation using XSB Prolog and search on ground instead of propositional theories (see further).

2 The Knowledge-Base System IDP^3

The language FO$(\cdot)^{\text{IDP}}$ We focus on the aspects of FO$(\cdot)$ that are relevant for this paper; more details can be found in [6], [1] and modeling examples at `dtai.cs.kuleuven.be/krr/software/idp-examples`. An FO$(\cdot)$ *formula* differs from FO formulas in two ways. Firstly, FO$(\cdot)$ is a many-sorted logic: every variable has an associated *type* and every type an associated domain. Moreover, it is order-sorted: types can be subtypes of others; the types *string* and *int* are built-in basic types. Secondly, the standard terms in FO are extended with aggregate terms: functions over a set of domain elements and associated numeric values which map to the sum, product, cardinality, maximum or minimum value of the set. A *definition* is a set of *rules* of the form $\forall \bar{x} : p(\bar{x}) \leftarrow \phi[\bar{x}]$. where $\phi[\bar{x}]$ is an FO$(\cdot)$ formula. They are interpreted according to the well-founded semantics[24], which naturally capture e.g., mathematical definitions.

An FO$(\cdot)$ *specification* consists of a set of named logical components, such as vocabularies, theories and structures. A *vocabulary* declares a set of types and typed symbols. A *theory* consists of sentences and definitions over a vocabulary. A *structure* specifies factual data over some vocabulary, by providing a (possibly partial) interpretation of the symbols in its vocabulary. As an illustration, an FO$(\cdot)$ specification of NQueens is shown in the upper part of Figure 1.

The model expansion inference Model expansion in IDP^3 takes as input a theory $\mathcal{T}$ over a vocabulary Σ, a (partial) interpretation $\mathcal{I}$ over Σ and an "output" vocabulary $\Sigma_{out} \subseteq \Sigma$ (respectively T, V, I and V_{out} in the running example). The task is to search for interpretations of Σ_{out} such that an extension exists to Σ that also extends $\mathcal{I}$ and is a model of $\mathcal{T}$.

The procedural interface IDP^3 combines declarative and procedural specifications by making all (named) FO$(\cdot)^{\text{IDP}}$ components available as first-class citizens and providing all inferences as method calls in the procedural interface. This integration is available both from the Lua scripting language (as in the example) and from C++; an interactive shell is also available from Lua. An example is shown in Figure 1, where a procedure *solve*() is defined such that it generates a model and afterwards prints the interpretation of *queen* in that model.

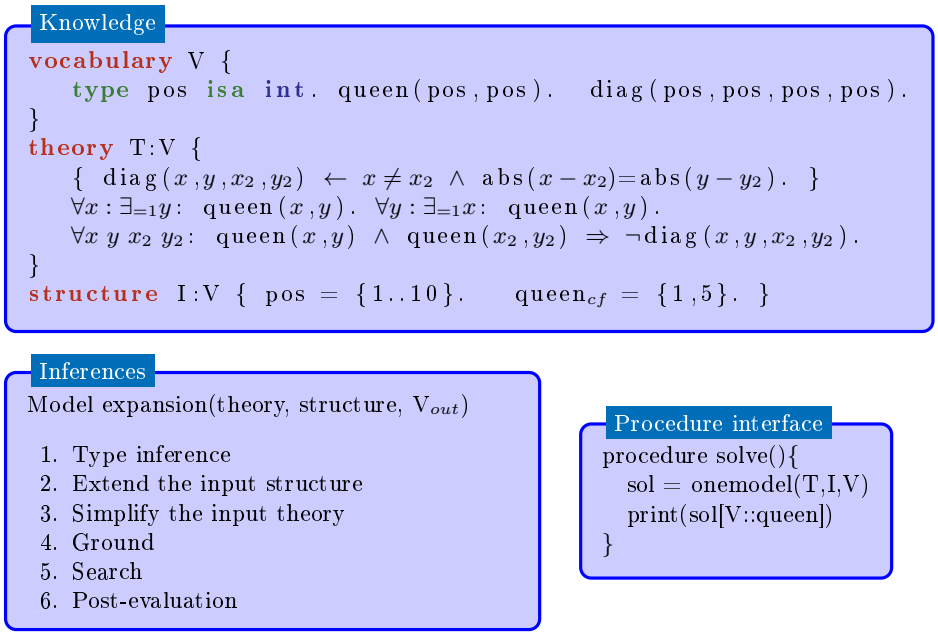

Fig. 1. Running example: NQueens with n = 10, where no queen (cf stands for "certainly false") is placed at $(1, 5)$.

3 Approach to model expansion

The "Inferences" part in the running example shows the basic components of our model expansion workflow. In this section, each of these is discussed in more detail. If inconsistency is derived at any point during the different stages (e.g., when extending $\mathcal{I}$ with entailed information), model expansion is aborted.

1. Type Inference [25] An important consideration for natural modelings is the ability to drop type specification wherever clear. For example for the sentence $\forall x : \exists_{=1} y : queen(x, y)$., it is clear that both x and y are *intended* to be typed as positions (pos). This could have been specified at the quantification (e.g., $\forall x[pos]$), however the system is often able to derive the intended type itself, because the vocabulary is fully typed. The type inference attempts to derive the intented types as follows[2]: for any untyped variable, its type is the order-minimum type which is a supertype of the types of all argument positions in which the variable occurs; for every variable, such a supertype is required to exist. In our example, x and y are typed pos as they both only occur in a position typed as pos ($queen(pos, pos)$). The result of the type inference step is a fully typed theory.

[2] Type inference is a non-equivalence preserving transformation: the sentence $\forall x[pos] : \exists_{=1} y[pos] : queen(x, y)$ is not logically equivalent with $\forall x : \exists_{=1} y : queen(x, y)$.

2. Structure Transformations The ideal structure input to the grounding step would be the *most precise* structure that captures all models of $\mathcal{T}$ that are more precise than $\mathcal{I}$. Additionally, we would like the structure to store its interpretation symbolically if possible, as the concrete structure can become very large (e.g., any structure that contains *int* is in fact infinite). A more precise structure allows us to create a smaller grounding and increase search performance. Symbolic representations are important as we might not need to completely enumerate an interpretation, but e.g., only check whether it contains a tuple of domain elements. We present the different symbolic representations IDP^3 supports and how some can be derived automatically from $\mathcal{T}$ and $\mathcal{I}$.

First, types and unary symbols can be specified as integer ranges, see e.g., *pos* in the running example. Second, an n-ary predicate symbol P can be specified as a Lua procedure that takes tuples $\overline{d}$ of length n that return true iff $P(\overline{d})$ is true (and similarly for function symbols). In the running example, we could have defined *diag* by a procedural call instead of by a definition. These types of symbolic representation are the responsibility of the modeler.

Third, for the different forms of sentences and definitions, the effects of Unit Propagation can be expressed as FO$(\cdot)$ definitions themselves[26][23]. For a given theory and structure, these FO$(\cdot)$ definitions can then be calculated using XSB [16]. The calculated results can then be used to make the structure more precise.

For example, if a queen is present on (x, y) in $\mathcal{I}$, it can be derived that there are no other queens in column x and row y, which can be expressed as the definition $\{queen_{cf}(x, y) \leftarrow \exists y' : queen_{ct}(x, y') \wedge y \neq y'\}$ (ct stands for "certainly true"). Instead of computing this definition in advance, which can be very expensive, a lifted version of BDDs[14] can be used to symbolically execute the propagation a number of times, resulting in an approximation of the full propagation[26]. For each symbol P, two such BDDs are derived (one for P_{ct} and one for P_{cf}), which are then added to the interpretation of P in $\mathcal{I}$.

3. Theory Transformations Next, the question is how the (now typed) theory can be improved to obtain a smaller grounding. The approaches used are (i) to reduce the type of quantified variables using the (symbolic) structure, and (ii) to exploit functional dependencies to drop quantified variables.

Reducing the quantification size [26] We first note that a quantification over a variable x of type T can also be seen as an instantiation of x with all values in the set $\{x \mid T(x)\}$ evaluated in $\mathcal{I}$. This can straightforwardly be generalized to a set of variables. Given a formula $\forall \overline{x} \in \{\overline{x} \mid T_1(x_1) \wedge \ldots \wedge T_n(x_n)\} : \varphi$, the size of the quantification can be reduced by replacing the set condition with the formula $T_1(x_1) \wedge \ldots \wedge T_n(x_n) \wedge \varphi$, where symbols in φ are replaced by their appropriate ct/cf symbolic interpretation in $\mathcal{I}$. This formula is then simplified to balance the cost of its evaluation against the (expected) reduction in number of answers.

For example, for the constraint $\forall x\ y\ x_2\ y_2 : queen(x, y) \wedge queen(x_2, y_2) \Rightarrow \neg diag(x, y, x_2, y_2)$, a possible associated set would be: $\{x\ y\ x_2\ y_2 \mid \neg queen_{cf}(x, y) \wedge \neg queen_{cf}(x_2, y_2) \wedge diag(x, y, x_2, y_2)\}$. This set can be further improved by replacing *diag* with its symbolic interpretation, resulting in $\{x\ y\ x_2\ y_2 \mid \neg queen_{cf}(x, y) \wedge$

$\neg queen_{cf}(x_2, y_2) \wedge x \neq x_2 \wedge abs(x - x_2) = abs(y - y_2)\}$. In the structure in the running example, this would for example eliminate all instantiations where $x = 1$ and $y = 5$, as $queen(1, 5)$ is false in $\mathcal{I}$, and instantiations where $x = x_2$.

Function Detection [4] It often happens that functional dependencies are present between arguments of the same symbol that are not explicitly modeled as a function. In our running example, the theory entails that *queen* is a bijection between rows and columns. We detect implicit functional dependencies by using a theorem prover to prove that the associated constraints are entailed by the theory. If this is the case, the theory is rewritten to explicitly introduce new function symbols, as this can significantly reduce the size of the grounding.

For example, we might replace $queen(pos, pos)$ with a function $f_q(pos) \mapsto pos$, mapping the first argument of *queen* to the second. The constraint $\forall x : \exists_{=1} y : f_q(x) = y$ is then dropped as it is trivially true, and the constraint $\forall x\ y\ x_2\ y_2 : queen(x, y) \wedge queen(x_2, y_2) \Rightarrow \neg diag(x, y, x_2, y_2)$ is replaced with $\forall x\ x_2 : \neg diag(x, f_q(x), x_2, f_q(x_2))$. The interpretation of *queen* in any model is then taken care of by calculating the definition $\{queen(x, y) \leftarrow f_q(x) = y\}$.

Definition stratification [16] Definitions in the theory are stratified whenever possible (if there are no loops between rules in the resulting definitions). A definition for which all open symbols are two-valued in $\mathcal{I}$ (in that case a unique expansion of the defined symbols exists) is translated into a Prolog program of which the answers are the interpretation of the defined symbols. This program is executed using XSB-Prolog, $\mathcal{I}$ is extended with the results and the definition is removed from the theory. This is repeated until a fixpoint is reached.

4. Grounding [27] The effective grounding phase basically consists of replacing all variables by all their matching instantiations, interleaved (for speed) with bringing the theory into a basic normal form using Tseitin variable introduction. The result is a ground FO($\cdot$) theory.

The grounding algorithm visits the theory in a depth-first, top-down fashion. One advantage is that the symbolic interpretation can be used *lazily*: types are generated one tuple at a time and atoms are only evaluated when they would effectively occur in the grounding. The main complexity in the algorithm stems from storing enough information to introduce a *minimal number* of Tseitin variables, which would otherwise reduce search performance. When returning from visiting a formula, the grounder creates a Tseitin variable representing that formula if needed. This is for example not necessary if a conjunction with a false subformula or a disjunction with a true subformula was visited, in which case "false", respectively "true" is returned instead. When creating Tseitin variables, the grounder reuses the same Tseitin variable for identical subformulas.

5. Search [3] Model expansion in IDP^3 relies on the state-of-the-art search implementation MiniSAT(ID) for ground FO($\cdot$) theories. The latest version of MiniSAT(ID) uses a search algorithm that combines the SAT-solver MiniSAT

with propagation algorithms for inductive definitions, uninterpreted functions (encoded lazily into SAT) and finite domain constraints such as aggregates.

6. Post-evaluation [16] In fact, as part of phase 3 (theory transformations), the theory can be further reduced, resulting in an even smaller grounding, by removing all rules of which the defined symbol occurs nowhere else in the theory (until fixpoint). These rules do not have to be considered for search as the interpretation of the symbols they define can be computed in polynomial time, given a model of the remaining theory, by transforming them into a Prolog program and evaluating it using XSB (cfr. definition simplification). An example is the definition added by function detection, where *queen* occurs nowhere else in the (resulting) theory. In fact, if a symbol does not occur in the output vocabulary, it does not even have to be computed (if no other definition depends on it).

4 Implementation and Applications

Both the KBS IDP^3 and the search algorithm MiniSAT(ID) are open-source, they are available from `dtai.cs.kuleuven.be/krr/software/`.

As discussed earlier, model expansion for FO$(\cdot)^{\text{IDP}}$ is closely related to answer set generation and solving CSPs. As such, it also shares applications with those domains, examples of which are scheduling, planning, verification and configuration problems. The performance of IDP has been demonstrated for example in the ASP competition series [9] and [2] (IDP^2) and the 2013 iteration (IDP^3)[3].

IDP is also used as a didactic tool in various logic-oriented courses, a.o. at KU Leuven, and an IDE for IDP is available at `dtai.cs.kuleuven.be/krr/software/idp-ide`.

5 Conclusion and future research

In this paper, we presented the architecture of the IDP^3 system, a next-generation of knowledge-base system, together with the workflow of its state-of-the-art model expansion inference. The system is an important step towards our aim of reducing the dependence of inference performance on the way knowledge is modeled. To this end, results and tools from different fields have been combined into one model expansion workflow. Important components are automated (i) derivation of smaller bounds on quantifications, (ii) detection of functional dependencies using theorem proving and subsequent theory simplification, (iii) stratification of definitions and efficient evaluation using tabled Prolog and (iv) a search algorithm combining the state-of-the-art techniques from SAT, ASP and CP. Future research is focusing on all aspects of KBS development, both the language, the inference and the interface between them. Concerning model expansion, we are looking into for example optimizing definitions automatically for efficient evaluating using Prolog and reducing the size of the grounding on-the-fly during search[5].

[3] Results of the 4^{th} ASP competition are not yet available at the time of writing.

References

1. Hendrik Blockeel, Bart Bogaerts, Maurice Bruynooghe, Broes De Cat, Stef De Pooter, Marc Denecker, Anthony Labarre, Jan Ramon, and Sicco Verwer. Modeling machine learning and data mining problems with FO($\cdot$). In Dovier and Santos Costa [10], pages 14–25.
2. Francesco Calimeri, Giovambattista Ianni, and Francesco Ricca. The third open answer set programming competition. *CoRR*, abs/1206.3111, 2012.
3. Broes De Cat, Bart Bogaerts, Jo Devriendt, and Marc Denecker. Model expansion with finite domain, uninterpreted functions. Accepted for the IEEE International Conference on Tools with Artificial Intelligence (ICTAI) - 2013.
4. Broes De Cat and Maurice Bruynooghe. Detection and exploitation of functional dependencies for model generation. Accepted for the 29th International Conference On Logic Programming.
5. Broes De Cat, Marc Denecker, and Peter Stuckey. Lazy model expansion by incremental grounding. In Dovier and Costa [10], pages 201–211.
6. Stef De Pooter, Johan Wittocx, and Marc Denecker. A prototype of a knowledge-based programming environment. In *International Conference on Applications of Declarative Programming and Knowledge Management*, 2011.
7. Marc Denecker, Yulia Lierler, Miroslaw Truszczynski, and Joost Vennekens. A tarskian informal semantics for asp. In *Technical Communications of the 28th International Conference on Logic Programming*, 2012.
8. Marc Denecker and Eugenia Ternovska. A logic of nonmonotone inductive definitions. *ACM Transactions on Computational Logic (TOCL)*, 9(2):14:1–14:52, April 2008.
9. Marc Denecker, Joost Vennekens, Stephen Bond, Martin Gebser, and Mirosław Truszczyński. The second Answer Set Programming competition. In *LPNMR*, pages 637–654, 2009.
10. Agostino Dovier and Vítor Santos Costa, editors. *Technical Communications of the 28th International Conference on Logic Programming, ICLP 2012, September 4-8, 2012, Budapest, Hungary*, volume 17 of *LIPIcs*. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2012.
11. Martin Gebser, Benjamin Kaufmann, and Torsten Schaub. Conflict-driven answer set solving: From theory to practice. *Artif. Intell.*, 187:52–89, 2012.
12. Martin Gebser, Torsten Schaub, and Sven Thiele. GrinGo : A new grounder for answer set programming. In Chitta Baral, Gerhard Brewka, and John S. Schlipf, editors, *LPNMR*, volume 4483 of *LNCS*, pages 266–271. Springer, 2007.
13. Gecode Team. Gecode: Generic constraint development environment, 2013. Available from `http://www.gecode.org`.
14. Jean Goubault. A BDD-based simplification and skolemization procedure. *Logic Journal of IGPL*, 3(6):827–855, 1995.
15. Daniel Jackson. Alloy: a lightweight object modelling notation. *ACM Transactions on Software Engineering and Methodology (TOSEM'02)*, 11(2):256–290, 2002.
16. Joachim Jansen, Albert Jorissen, and Gerda Janssens. Compiling input$*$ FO($\cdot$) inductive definitions into tabled prolog rules for IDP^3. Accepted for publication in Theory and Practice of Logic Programming, Special Issue ICLP-2013.
17. Nicola Leone, Gerald Pfeifer, Wolfgang Faber, Thomas Eiter, Georg Gottlob, Simona Perri, and Francesco Scarcello. The DLV system for knowledge representation and reasoning. *ACM Transactions on Computational Logic (TOCL)*, 7(3):499–562, 2006.

18. Victor W. Marek and Mirosław Truszczyński. Stable models and an alternative logic programming paradigm. In Krzysztof R. Apt, Victor W. Marek, Mirosław Truszczyński, and David S. Warren, editors, *The Logic Programming Paradigm: a 25-Year Perspective*, pages 375–398. Springer-Verlag, 1999.
19. Kim Marriott, Nicholas Nethercote, Reza Rafeh, Peter J. Stuckey, Maria Garcia de la Banda, and Mark Wallace. The design of the Zinc modelling language. *Constraints*, 13(3):229–267, 2008.
20. Laurent Michel and Pascal Van Hentenryck. The Comet programming language and system. In Peter van Beek, editor, *CP*, volume 3709 of *LNCS*, pages 881–881. Springer, 2005.
21. Robert Nieuwenhuis, Albert Oliveras, and Cesare Tinelli. Solving SAT and SAT modulo theories: From an abstract Davis–Putnam–Logemann–Loveland procedure to DPLL(T). *Journal of the ACM*, 53(6):937–977, 2006.
22. Stef De Pooter, Johan Wittocx, and Marc Denecker. A prototype of a knowledge-based programming environment. *CoRR*, abs/1108.5667, 2011.
23. Pashootan Vaezipoor, David Mitchell, and Maarten Mariën. Lifted unit propagation for effective grounding. *CoRR*, abs/1109.1317, 2011.
24. Allen Van Gelder. The alternating fixpoint of logic programs with negation. *Journal of Computer and System Sciences*, 47(1):185–221, 1993.
25. Johan Wittocx. *Finite Domain and Symbolic Inference Methods for Extensions of First-Order Logic*. PhD thesis, Department of Computer Science, K.U.Leuven, Leuven, Belgium, May 2010.
26. Johan Wittocx, Marc Denecker, and Maurice Bruynooghe. Constraint propagation for first-order logic and inductive definitions. *ACM Transactions on Computational Logic*, 2013. Accepted.
27. Johan Wittocx, Maarten Mariën, and Marc Denecker. Grounding FO and FO(ID) with bounds. *Journal of Artificial Intelligence Research*, 38:223–269, 2010.

Learning Non-Ground Rules for Answer-Set Solving

Antonius Weinzierl

Institute of Information Systems
Vienna University of Technology

Abstract. Today there exist many efficient solvers to compute the answer-sets of a logic program. The traditional approach is to first compute the grounding $gr(P)$ of a given ASP program P and then run a propositional ASP solver on $gr(P)$. But $gr(P)$ may be exponentially larger than P and in general this blow-up cannot be prevented. This poses a major problem, which is also known as the "grounding bottleneck".
In recent years, the approach of grounding on-the-fly has been proposed to circumvent the need for pre-grounding and to eliminate the grounding bottleneck. The Omiga solver follows this approach and it can compete with state of-the-art ASP solvers in a certain classes of problems due to the integration of a Rete network for efficient propagation. In this paper, we extend the solving capabilities of Omiga by introducing methods for conflict-driven learning of non-ground rules. Conceptually, this is similar to clause-learning in SAT solving: an implicit implication graph is used for the application of rule-unfolding to obtain (sound) rules that contain variables. Initial experiments show that the approach can dramatically reduce the amount of necessary choices.

1 Introduction

Grounding can be a major obstacle in Answer-Set Programming (ASP) and different approaches exist to mediate or overcome this issue. In this paper we extend the solving capabilities of the Omiga solver for ASP programs by methods for conflict-driven learning.

Given an ASP program P with rules that contain variables, the traditional approach to compute the answer-sets $\mathcal{AS}(P)$ of P has two phases: first, the ground version $gr(P)$ of P is computed, second, a propositional solver finds all answer-sets $\mathcal{AS}(gr(P))$ of the ground program. This approach has the important advantage that the solving takes place on a fully ground program, i.e., no variables have to be taken into consideration. Two of the answer-set solvers which scored highest on several ASP benchmarking competitions, namely Clasp [4] and DLV [7], follow this pre-grounding methodology, since it allows many techniques to make the computation of answer-sets efficient.

An issue with this two-step approach, however, is the fact that the grounded program $gr(P)$ may be exponentially larger than P, even if only a small part of $gr(P)$ is needed to compute all answer-sets. To mitigate this issue, there has

been a considerable amount of work to keep the size of $gr(P)$ small, e.g. by intelligent grounding [2]. Since this grounding bottleneck cannot be overcome by intelligent grounding alone, it still is an open issue and it limits the applicability of ASP, especially for large problem instances.

In [8] the notion of a persistent computation is introduced to characterize the answer-sets of a program. Conceptually a persistent computation is a sequence of partial interpretations which are revised in such a manner that, when a consistent fixpoint is reached, this fixpoint corresponds to an answer-set. The revision basically applies all rules that are applicable under the current interpretation (the propagation step), and if no more rules are applicable, then a choice is made whether a rule with the following properties fires or not (the guessing step): the rule has a nonempty negative body, its positive body is applicable under the current interpretation, and the negative body is not contradicted.

This notion then lead to the development of ASP solvers which compute persistent computations (or computation sequences) in such a manner that a rule is grounded only when it fires. This grounding is interwoven with the answer-set search, therefore no pre-grounding is necessary and the grounding bottleneck vanishes. In [5, 6] the ASPeRiX solver is introduced where this approach to grounding is called grounding on-the-fly and in [9] the GASP solver is introduced where the same approach is called lazy-grounding. Although these solvers are able to overcome the grounding issue, they are unable to compete with Clasp or DLV on instances where grounding is not the hardest part, i.e., such programs where the solvers have to do a lot of guessing.

In [1] we introduce the Omiga solver, which follows the grounding on-the-fly approach. To improve efficiency of propagation, Omiga is built upon a Rete network, which is utilized for finding variable assignments to non-ground rules such that the resulting ground rules are applicable in the current partial interpretation. Note that the problem of finding such an applicable rule in itself is NP-complete; as [1] shows, the use of Rete improves performance.

But the grounding on-the-fly solvers have sever issues with ASP programs where many guessing steps are required, since they apply depth-first search without any optimizations. This is due to the fact that optimizations used by solvers with pre-grounding are not fit for use in non-ground scenarios. Conflict-driven learning (cf. [10]), for example, contributes much to the solving capabilities of Clasp, and similar techniques are used by DLV. But it is not known how such techniques could be applied to non-ground ASP solving. Although such conflict-driven learning of non-ground rules can have dramatic effects on the number of guesses needed to find an answer-set, there has not been (to the best of our knowledge) any research into transferring this principle to the non-ground case.

Therefore, the main contribution of this paper is the introduction of conflict-driven learning for grounding on-the-fly solvers. Our approach yields (correct) rules with variables and as initial experiments with Omiga show, these rules can have a huge impact on the number of necessary guesses.

Summarizing, the contributions of this paper are:

1. a novel approach for grounding on-the-fly ASP solving to learn new non-ground rules during answer-set search. Hence, we transfer conflict-driven learning to the non-ground case;
2. a description of how ASP solvers for non-ground programs can employ a Rete network for more efficient the propagation; and
3. an implementation of the introduced approaches and an initial experimental evaluation of the resulting Omiga solver for ASP programs.

2 Preliminaries

Given a finite set $\mathcal{C}$ of constants, a set $\mathcal{V}$ of variables, and a finite set $\mathcal{P}$ of predicates such that these sets are all disjoint, then $a(t_1, \ldots, t_n)$ is an atom if $a \in \mathcal{P}$ and $t_1, \ldots, t_n \in \mathcal{C} \cup \mathcal{V}$. We say the atom is ground, if $t_1, \ldots, t_n \in \mathcal{C}$, otherwise it is a non-ground atom. A rule r is of form

$$a_0 \leftarrow a_1, \ldots, a_k, not\, a_{k+1}, \ldots, not\, a_n. \tag{1}$$

where every a_i is an atom, $0 \leq i \leq n$. We denote the head of r by $head(r) = a_0$, the positive body by $body^+(r) = \{a_1, \ldots, a_k\}$, the negative body by $body^-(r) = \{a_{k+1}, \ldots, a_n\}$, and the whole body by $body(r) = body^+(r) \cup body^-(r)$. By $var(a)$ we denote the set of variables occurring in the atom a, by $var(r)$, $var(body(r))$, $var(body^+(r))$, respectively $var(body^-(r))$, we denote the set of variables occurring in r, $body(r)$, $body^+(r)$, respectively $body^-(r)$. A rule r is non-ground iff $var(r) \neq \emptyset$, ground otherwise. Wlog. assume that each $a \in body(r)$ is unique.

Furthermore, for a set $A = \{b_1, \ldots, b_m\}$ of atoms, we write $not(A)$ to denote the set $\{not\, b_1, \ldots, not\, b_m\}$, so $r = head(r) \leftarrow body^+(r) \cup not(body^-(r))$. An ASP program P is a finite set of rules of form (1). Let the Herbrand universe $\mathcal{HU}_P$ and Herbrand base $\mathcal{HB}_P$ wrt. P be defined in the usual way. Given a set $A \subseteq \mathcal{HB}_P \cup not(\mathcal{HB}_P)$ of literals (containing negative and positive atoms), we write $A^+ = A \cap \mathcal{HB}_P$, respectively $A^- = A \cap not(\mathcal{HB}_P)$, to denote all positive, respectively negative, atoms of A. We assume that double negation eliminates itself, i.e. $not(not\, a) = a$ and $not(not\, A) = A$ where a is an atom and A is a set of atoms.

A (partial) interpretation I for P is any set $I \subseteq \mathcal{HB}_P \cup not(\mathcal{HB}_P)$. An interpretation I is partial, if $A^+ \cup not(A^-) \neq \mathcal{HB}_P$, it is total otherwise; I is contradictory, if $A^+ \cap not(A^-) \neq \emptyset$. Let answer-sets of P be defined in the usual way, we denote by $\mathcal{AS}(P)$ the set of all answer-sets of P over $\mathcal{HB}_P$. Given a (partial) interpretation I, we say that I' is an extension of I iff $I \subseteq I'$.

Let $x_1, \ldots, x_k \in \mathcal{V}$ and $c_1, \ldots, c_k \in \mathcal{C}$, then $\theta = \{x_1 \mapsto c_1, \ldots, x_k \mapsto c_k\}$ is a variable assignment. Given two variable assignments θ and ρ, we say that they are *compatible* iff for any $x \in \mathcal{V}$ such that $x \mapsto c \in \theta$ and $x \mapsto c' \in \rho$ it follows that $c = c'$. For compatible variable assignments θ, ρ, their join $\theta\rho$ is $\theta \cup \rho$. Variable assignments which are not compatible can not be joined. Note that our definition implies that variable assignments always map variables to constants.

Let a be an atom with $var(a) = \{x_1, \ldots, x_m\}$ and let $\theta = \{x_1 \mapsto s_1, \ldots, x_m \mapsto s_m\}$ be a variable assignment, then $a\theta$ is the ground atom obtained by replacing for every $1 \leq i \leq m$ each occurrence of x_i with s_i. Let r be a non-ground

rule with $var(r) = \{x_1, \ldots, x_m\}$ and let θ be an assignment for each variable of $var(r)$, then $r\theta$ is a grounding substitution of r, which is obtained by replacing each atom $a \in body(r)$ with $a\theta$. Let $x_1, \ldots, x_k \in \mathcal{V}$ and let $c_1, \ldots, c_k \in \mathcal{V} \cup \mathcal{C}$, then $\theta = \{x_1 \mapsto c_1, \ldots, x_k \mapsto c_k\}$ is a non-ground variable assignment; let a and b be two (non-ground) atoms, we say that the non-ground variable assignment θ is a unifier of a and b iff $a\theta = b\theta$. Let the most general unifier be defined in the usual way, then $unify(a, b)$ is a most general unifier of a and b.

For space reasons, we refer the reader to [5] and [6] for an overview of the grounding on-the-fly approach taken by ASPeRiX. Its main characteristics are that the solving is an iteration of (a) computing the fixpoint of propagation, and (b) guessing on an applicable, not blocked ground-rule $r\theta$ whether it fires or not. Note that $r\theta$ is applicable wrt. an interpretation I iff $body^+(r\theta) \subseteq I^+$, and it is not blocked iff $body^-(r\theta) \cap I^+ = \emptyset$. If $r\theta$ is applicable and $body^-(r\theta) \subseteq I^-$, then $r\theta$ fires by propagation already.

3 Representing ASP Rules in Rete

Rete [3] is a well-known technique in rule-based production systems which has been proposed as a technique to solve the many-pattern many-object problem. A Rete network is a directed graph where nodes represent different parts of a pattern (in our case a pattern is a rule body). Each node comes with a working memory (WM) where all instances are stored that match the pattern that is represented by the node. Directed edges in the Rete network indicate the flow of matching instances, i.e., if a node receives an instance from another node, it tests whether that instance matches the pattern it represents, if it matches, then the node stores the instance in its own WM and also forwards the instance to all other nodes it has out-going edges to. Observe that we use instance and (partial) variable assignment interchangeably here.

Example 1. For some intuition on Rete, we consider a simplified example. Given some rule $r = p(X, Y) \leftarrow q(X, Y), r(Y, a)$ and a (partial) interpretation $I = \{q(1, a), r(a, a)\}$, then Rete contains a (basic) node $BN(p)$, $BN(q)$, and $BN(r)$ for p, q, and r, respectively. Initially, the WM of $BN(p)$ is empty, the WM of $BN(q)$ is $\{(1, a)\}$ and the WM of $BN(r)$ is $\{(a, a)\}$. To represent r, there is one selection node SN representing $r(Y, a)$, and one join node JN representing $q(X, Y), r(Y, a)$. There is, furthermore, an edge from $BN(q)$ to JN, one from $BN(r)$ to SN, and one from SN to JN. Finally, there also is a node $HN(r)$ representing the head of r which is the only child of JN.

When propagation is started on the Rete, then every node pushes all instances of its WM to those children which have not yet received that instance. So $BN(r)$ pushes (a, a) to SN, where it matches (Y, a) and therefore it is stored in the WM of SN. Furthermore, $BN(q)$ pushes $(1, a)$ to JN, where JN tries to find a join-partner for $(1, a)$ in the WM of SN. It finds (a, a), hence JN stores $(1, a, a)$ in its WM and sends it to HN. HN then unifies $(1, a, a)$ with $p(X, Y)$, obtains $p(1, a)$ and pushes $(1, a)$ into the WM of $BN(p)$. Now one round of propagation is finished and the next round is started, until a fixpoint is reached.

Since a grounding on-the-fly solver has to match partial interpretations with non-ground rules to find new applicable ground rules, the Rete network seems ideal for realizing such a solver, since the partial instantiations of a non-ground rule are stored in the WM of the respective join node. So when a new atom is derived, the Rete network ensures that this atom is pushed to all partial rule instantiations where this atom might contribute to. As is shown in [1], Rete indeed speeds-up the computation.

Before going into detail we first consider the algorithm underlying the evaluation in Omiga. The algorithm is generic and its main functionality is relayed to Rete, which supplies functionality for propagation, doing choices, and backtracking. Propagation is a straightforward implementation as in many Rete implementations; it is a fixpoint computation as sketched in Example 1. Omiga follows the concept of doing choices from ASPeRiX, but it employs the Rete network to do so. Let I be the partial interpretation represented by contents of the WMs of the Rete network, to find a rule r and a variable assignment θ such that $body^+(r\theta) \subseteq I^+$, $body^-(r\theta) \cap I^+ = \emptyset$, and $head(r\theta) \notin I^+$ hold, Omiga only needs to select an entry θ from the WM of the so-called choice node of r (more details below). Backtracking also is straightforward: every variable assignment (instance in some WM) is associated a decision level and for backtracking all respective variable assignments are deleted from the WM of all nodes.

Algorithm 1 shows the generic algorithm employed by Omiga; basically it is an exhaustive depth-first search algorithm. Note that *Rete.doChoice* returns true, if there was a rule to guess on, it returns false if no more guesses can be done, hence it signals that a potential answer-set has been computed. For positive programs, there are no choices and therefore the answer-set is already computed by the first *Rete.propagate*() in line 4.

Since any node of a Rete network comes with a WM where (partial) variable instantiations are stored, we identify in the following each node with the set of variable instantiations in its WM. To illustrate the working of our Rete, we consider an arbitrary partial interpretation I of a program P, which is represented by the Rete. Given a rule $r \in P$ of form (1) the Rete representing this rule is as follows. For every predicate $p \in \mathcal{P}$ which occurs in r, there are two nodes: a so-called basic node $BN^+(p)$ storing all derived instances of p, and a basic node $BN^-(p)$ storing all instances of p which are known to not occur in I, i.e., $BN^+(p) = \{\theta \mid p(t_1, \ldots, t_n)\theta \in I^+\}$ and $BN^-(p) = \{\theta \mid p(t_1, \ldots, t_n)\theta \in not(I^-)\}$. Note that we treat negation in ASP by a compilation to specific nodes; negation in Rete networks usually is different, but this compilation allows to retain the negation-as-failure semantics of ASP rules.

Each atom $a_i \in body(r)$ is represented by a so-called selection node $SN(a_i)$. Let $a_i = p(t_1, \ldots, t_n) \in body^+(r)$, then the task of $SN(a_i)$ is to extract all matching variable assignments from $BN^+(p)$, i.e., $SN(a_i) = \{\theta \mid a_i\theta \in BN^+(p)\}$. For $a_i \in body^-(r)$, $SN^-(a_i)$ is defined accordingly, i.e., $SN(a_i) = \{\theta \mid a_i\theta \in BN^-(p)\}$. To achieve that, $SN(a_i)$ is a child node of the respective basic node. Hence, if some instance arrives at the basic node, it is then pushed to $SN(a_i)$.

Algorithm 1: The generic algorithm employed by Omiga.

```
   Input  : A safe ASP program P.
   Output: AS(P)
 1 AS ← ∅
 2 Build Rete network R
 3 Populate R with facts
 4 Rete.propagate()
 5 if Rete.isInconsistent() then
 6 |  finished = true
 7 end
 8 while !finished do
 9 |  if Rete.doChoice()                  // execute guess, increase decisionLevel
10 |  then
11 |  |  Rete.propagate()
12 |  |  if Rete.isInconsistent() then
13 |  |  |  Rete.backtrack()
14 |  |  end
15 |  else
16 |  |  if !Rete.isInconsistent() then
17 |  |  |  A ⟵ ⋃_{p∈P} BN+(p)          // extract answer-set from Rete
18 |  |  |  AS ⟵ AS ∪ A
19 |  |  end
20 |  |  if Rete.decisionLevel > 0 then
21 |  |  |  Rete.backtrack()
22 |  |  else
23 |  |  |  finished = true
24 |  |  end
25 |  end
26 end
```

Then follows a series of join nodes, where each join node represents a specific part of the rule body. Given two parents n_1 and n_2 of a join node, its working memory is $JN(n_1, n_2) = \{\theta\rho \mid \theta \in n_1, \rho \in n_2,$ and both are compatible$\}$. For our rule r there are selection nodes $SN(a_1)$ and $SN(a_2)$, and a join node $N_1 = JN(SN(a_1), SN(a_2))$, which represents the first two positive literals of r. Furthermore, there are join nodes $N_2 = JN(N_1, SN(a_3))$, $N_3 = JN(N_2, SN(a_4))$, until $N_{k-1} = JN(N_{k-2}, SN(a_k))$ which each represent (increasing) parts of $body^+(r)$. Observe that N_{k-1} represents the whole positive body of r, hence any variable assignment $\theta \in N_{k-1}$ is such that $a_i\theta \in I^+$ holds for every $1 \leq i \leq k$. This aids when making a guess, since *Rete.doChoice* needs to find r and θ such that $body^+(r)\theta \in I^+$; all instances in N_{k-1} have this property.

To aid in the selection of rules and instances for guessing, we introduce another type of node, the choice node $CN(r)$, which is a child of the node N_{k-1} where N_{k-1} is the join node that represents the positive body of r. Since a choice requires an r such that $body^+(r)\theta \in I^+$ and $body^-(r)\theta \cap I^+ = \emptyset$, one

could expect that $CN(r)$ also gets input from every node representing a negative literal of r. Since this information is only needed once, i.e., when an instance is selected from $CN(r)$, the choice node receives such information not regardless if it is needed, but the choice node specifically requests it from the corresponding basic node. Hence methods outside the usual Rete approach are used here.

We continue with the Rete structures to represent the negative literals of the rule r above. Recall that we only consider safe programs, i.e., programs where all variables of the head of r and in $body^-(r)$ also occur in $body^+(r)$. For our Rete structure this means that the join node N_{k-1} representing the positive body of r contains a binding for each variable that occurs in r. To that end, we introduce a new type of node for negative joins, denoted by $nJN(n_1, n_2) = \{\theta \mid \theta \in n_1, \rho \in n_2, \rho \subseteq \theta\}$. Let N_{k-1} be the join node representing $body^+(r)$, then we have that $N_k = nJN(N_{k-1}, SN^-(a_{k+1}))$, $N_{k+1} = nJN(N_k, SN^-(a_{k+2}))$, until $N_n = nJN(N_{n-1}, SN^-(a_n))$, which represents $body(r)$. Note that for negative literals we use SN^-, i.e., the basic nodes representing I^-, those atoms of the partial interpretation I which are known to not occur in any extension I' of I.

The negative join nodes also serve the additional purpose of so-called closing: assume that we arrive at a partial interpretation J where it is clear due to an analysis of the dependency graph of the program P, that for some predicate p, no more atoms over that predicate can be derived in any extension J' of J; then, for any negative selection node $SN^-(a)$ where $a = p(t_1, \ldots, t_m)$ and m is the arity of p, the behavior of the negative selection nodes can be switched to not consider the explicit $BN^-(p)$, but to consider $BN^+(p)$ in reversed form. Formally, if $SN^-(a)$ is closed, then $SN^-(a) = \{\theta \mid \theta \notin BN^+(p)\}$. So the closing of p conceptually is the completion of J wrt. p, i.e., $\{p(t_1, \ldots, t_m) \in (J^+ \cup \neg J^-)\} = \{p(t_1, \ldots, t_m) \in \mathcal{HB}_P\}$. Of course, it is not efficient to store this (large) amount of information explicitly. Therefore, all negative join nodes can be triggered to query the closed negative selection nodes. By safety of rules, all variables are already bound, so the negative selection node does not generate new assignments, but just confirms that an atom is contained in J^-.

Finally, there is one sort of node remaining, the head node $HN(r)$, which takes the variable assignments from the body and generates the respective atoms wrt. the head of r. Let N_m be the (negative) join node representing $body(r)$, then $HN(r) = \{a_0\theta \mid \theta \in N_m\}$. These instances now are immediately stored in the respective basic node, i.e, in $BN^+(p)$ where $a_0 = head(r) = p(t_1, \ldots, t_m)$ and m is the arity of p. In the case that r is a constraint, i.e., $head(r) = \bot$, then inconsistency is signaled if $HN(r) \neq \emptyset$. Since head nodes serve no other purpose, there is actually no need for them to have a WM, since any instance arriving at $HN(r)$ either becomes immediately added to $BN^+(p)$, or inconsistency is triggered immediately. In the implementation therefore there is no WM for head nodes.

In what follows, we will consider a new type of rules, namely such rules r' where the head node adds instances not to $BN^+(p)$, but to $BN^-(p)$ instead. We call the corresponding head node $nHN(r')$ a negative head node. If we also consider that some atoms must-be-true, but there is not yet a rule of P which

derives that atom, then this leads to another type of head node, one for must-be-true atoms, i.e., $mbtHN(r)$. For space reasons, however, we cannot go into detail of must-be-true propagation here.

4 Learning Non-Ground Rules

Unfolding: Conflict-driven learning of many SAT- or ASP-solvers is rooted in propositional resolution. Our approach for non-ground conflict-driven learning is based on similar ideas, but instead of full first-order resolution, we employ rule unfolding to derive new rules that are implied by the original program.

In essence, if the body of a rule r_1 contains a literal b which can be derived by a rule r_2 (i.e., if the head of r_2 unifies with b), then we can create a new rule r_ℓ where b is replaced by the body of r_2, i.e., $body(r_\ell) = (body(r_1) \setminus \{b\}) \cup body(r_2)$ and $head(r_\ell) = head(r_1)$ modulo the correct renaming of variables to incorporate the unification of b and $head(r_2)$. Now the addition of r_ℓ does not change the semantics of the original program, since r_ℓ can only derive atoms which also can be derived using r_2 and r_1. But r_ℓ can derive some information without the need to fire r_2 first.

To realize the unfolding, the unification of b and $head(r_2)$ is required. For simplicity, we introduce additional equality predicates that establish the unification and to avoid name clashes among variables, we rename all variables of r_2.

Definition 1. *Let r_1 be a rule and r_2 be a rule such that there is $a \in body(r_1)$, and $head(r_2)$ unifies with a. Let $r' = nV(r_2)$ be the renaming of all variables of r_2 with new variables, i.e., $var(r') \cap var(r_1) = \emptyset$, and let $unify(a, head(r')) = \theta$. Then, the* unfolding-step *of r_1 with r_2 by a is*

$$unfold(r_1, r_2, a) = body(r_1) \setminus \{a\} \cup body(r') \cup \{(X = S) \mid (X \mapsto S) \in \theta\}.$$

Given a program P and a sequence U of unfolding steps such that

$$U = (unfold(r_0, r_1, a_1), unfold(r'_1, r_2, a_2), \ldots, unfold(r'_{n-1}, r_n, a_n))$$

where it holds that $r_0, \ldots, r_n \in P$ and $unfold(r'_{i-1}, r_i, a_i) = r'_i$ holds for every $1 \leq i \leq n-1$, then we say that U is an unfolding-sequence. Finally, r_U is the *U-unfolding* of r iff $r_U = r'_n = unfold(r'_{n-1}, r_n, a_n)$ and $r = r_0$ holds.

Let P be a program, c be a constraint, and r_U be the U-unfolding of c for some sequence U of unfolding steps. Then, an *unfolding* of c is the constraint

$$\bot \leftarrow r_U.$$

One can check that any answer-set of P which satisfies c also satisfies any unfolding of c. The following proposition shows that this even holds for partial interpretations and their extensions to possible answer-sets.

Proposition 1. *Given a program P, a constraint $c \in P$, some unfolding c' of c, and a partial interpretation I for P. If c' is violated by I, then there is no answer-set I' such that $I \subseteq I'$.*

Proof (Sketch). Towards contradiction assume that there is an answer-set I' with $I \subseteq I'$ and c is not violated by I', i.e., $I' \not\models \mathit{body}(c)$, but c' is violated by I. Since I violates c', it holds that $I \models \mathit{body}(c)$. By definition of the unfolding, there must be a rule r_1 whose body is a subset of the body of c, hence it holds that $\mathit{head}(r_1) \in I'$ since I' is an answer-set. By induction this holds for all rules that were used in the unfolding of c. Consequently, it holds that $I' \models \mathit{body}(c)$, which contradicts the assumption and shows that no $I' \supseteq I$ can satisfy c, hence I' cannot be an answer-set of P.

Implication Graphs and Unfolding: We now show how the unfolding can be guided by conflicts (i.e., constraint violations). For that the notion of an implication graph, which is implicitly built during search of an answer-set, is useful. Nodes in the implication graph represent truth assignments to ground literals. Different from propositional implication graphs, where directed edges represent unit-propagation, our edges represent applicability of a ground rule. If a ground rule r fires, then the implication graph contains an edge from every literal $b \in \mathit{body}(r)$ to the head literal $a \in \mathit{head}(r)$. Furthermore, the label of a node indicate at which decision level the value was assigned and which non-ground rule was used to derive that value.

Definition 2. *An* implication graph *wrt. a partial interpretation I is an acyclic graph $G = (V, E)$ where each node $v \in V$ represents a truth assignment of a ground atom, i.e., v is assigned a label of the form: $p(t_1, \ldots, t_n) = f@L$ where $p \in \mathcal{P}$, $t_1, \ldots, t_n \in \mathcal{C}$, $f \in \{0, 1\}$ indicates whether $f \in I^+$ or I^-, and L is the decision level at which v was derived.*

Let r be the rule which fired at decision level L and derived v, i.e., r is a rule such that there exists a variable assignment θ with $\mathit{head}(r)\theta = p(t_1, \ldots, t_n)$ and for every $a \in \mathit{body}(r)$ holds that $a\theta \in I$, where I is the partial interpretation represented by G. Then, there exists an edge $e = (v', v)$ for every $a \in \mathit{body}(r)$ where $a\theta@L'$ is the label of v' and $L' \leq L$ holds. Furthermore, e is labeled by $r : \theta$.

Note that for any implication graph, all in-edges of a node have the same label. Also observe that the implication graph $G = (V, E)$ wrt. a partial interpretation I is implicitly stored in the Rete network wrt. I. In the description of Section 3, the information regarding nodes V is already present. To represent edges E, the basic nodes of the Rete are adapted to not only store instances, but also store the rule and its instantiation which derived the given instance.

If a constraint is violated, then an implication graph contains a conflict node κ, representing the empty head of the violated constraint. If an implication graph G contains a conflict node, then G is a *conflict-graph*.

For conflict-driven learning, we now use the conflict-graph to consider only such a U-unfolding where each unfolding-step is guided by the conflict-graph. If there is a r-labeled edge from some node to κ, then r may be unfolded recursively.

Definition 3. *Let $G = (V, E)$ be a conflict graph with conflict node κ at decision level L. Let $G' = (V', E')$ be the smallest subgraph of G such that*

1. $\kappa \in V'$
2. *if* $v \in V'$ *and* $e = (v', v) \in E$ *where* v' *is labeled* $a = f@L$ *for some* a *and* f, *then* v' *in* V' *and* $e \in E'$.

The unfolding traversal *induced by* G *then is a traversal of* G' *starting at* κ. *It is inductively defined as follows:*

$$U_o = (body(r), \kappa) \text{ such that } e = (v, \kappa) \in E', \text{and } e \text{ is labeled } r : \theta \tag{2}$$

$$U_{i+1} = (unfold(r_i, r', a), v) \text{ if there exists } v \text{ with } e = (v, v') \in E', \tag{3}$$
$$U_i = (r_i, v'), e \text{ is labeled } r'\theta \text{ and } v \text{ is labeled } a@f = L \text{ for some } f;$$

otherwise

$$U_{i+1} = (unfold(r_i, r', a), v) \text{ if there exists } v \text{ with } e = (v, v') \in E', \tag{4}$$
$$U_j = (r_j, v') \text{ for some } j \leq i, \text{ and } v' \notin \{v_\ell \mid U_\ell = (r_\ell, v_\ell), 1 \leq \ell \leq i\}$$

Let $U_o, U_1, \ldots, U_n$ be the unfolding traversal that is induced by G and let $U_i = (unfold(r_i, r_{i+1}, a_i), v_i)$ for $1 \leq i \leq n$, then

$$U = (unfold(r_1, r_2, a_1), \ldots, unfold(r_{n-1}, r_n, a_{n-1}))$$

is an unfolding sequence, which represents an unfolding of all rules, on the same decision level as κ, whose firing contributes to the conflict.

Shifting: By unfolding, one can derive new constraints from a program P. These constraints already can be useful for ASP solvers which rely on some form unit-propagation. In Omiga, however, only rules with non-empty head allow to derive more information. Therefore, we now consider a method to obtain new rules from a constraint. Since this technique is tailored for the Omiga solver, the created rules can be processed by Omiga, but these rules are not normal in the strict sense. Note however, that the resulting propagation is also employed in other ASP solvers, e.g., DLV.

The intuition of the shifting approach is as follows: assume that the body of a constraint c is almost satisfied, i.e., all but one literal $a \in body(c)$ are satisfied by a partial interpretation I. If there exists an answer set I' with $I \subseteq I'$, then c also must not be violated by I', which means that I' must falsify a. Hence, we can derive the falsity of a directly at I, i.e., the constraint c implies for each $a \in body(c)$ a rule $not\, a \leftarrow body(c) \setminus \{a\}$.

It is important to note that such a rule requires special treatment of $not\, a$ by the solver: if $a \in body^-(c)$, then the head, $not\, not\, a$, of the created rule must be interpreted as so-called must-be-true to preserve answer-set semantics. Hence it is still necessary to derive a using a normal rule of the input program P. On the other hand, if $a \in body^+(c)$, then the head of the created rule $(\neg a)$ can directly extend the negative part I^- of any partial interpretation I. This behavior is correct since assuming some atom to be not in an answer-set requires no further conditions to hold, but assuming some atom to be in an answer-set requires that the atom is supported and founded by rules of the original program.

Definition 4. *Given a constraint* c, *the set of* constraint-induced *rules is*

$$ind(c) = \{not\, a \leftarrow body(c) \setminus \{a\}. \mid a \in body(c)\}.$$

Note that not all constraint-induced rules are safe, since the head of a constraint-induced rule might contain variables that are not bound in its body. On the other hand, a constraint-induced rule only needs to derive additional information; which is different from normal rules where a negative body can trigger more choice points. Therefore, we consider constraint-induced rules to fire only when the positive and negative body is fulfilled. Hence, the notion of safety for such rules is less strict, i.e., a constraint-induced rule r is safe iff $var(head(r)) \subseteq var(body(r))$.

Properties: One important property, which is based on Proposition 1, is the correctness of the presented learning approach, i.e., every learned rule is implied by the original program.

Proposition 2. *(Correctness) Let $r \in ind(c)$ be a safe constraint-induced rule, and let c be the unfolding of some constraint $c' \in P$, then $\mathcal{AS}(P \cup \{r\}) = \mathcal{AS}(P)$.*

Assume that one ASP encoding of a problem is used for multiple problem instances (like in the ASP competitions). Furthermore, assume that the conflict-graph guided U-unfolding only used rules from the problem encoding (and not from the problem instance), then this learned non-ground rule can be used for any of the other problem instance, too. This holds, because for every other problem instance, the learned non-ground rule again can be derived.

Proposition 3. *Assume a rule r is learned based on a subset T of the rules of P_1 while solving a program P_1. If P_2 is such that $T \subseteq P_2$, then $\mathcal{AS}(P_2 \cup \{r\}) = \mathcal{AS}(P_2)$.*

5 Evaluation

We evaluate our approach of learning in two ways. First, we consider all benchmarks from [1] and run Omiga once with learning enabled and once without, to see the impact and cost of learning on computation speed. Second, we pick one of the hard instances for Omiga (3-colorability) and analyze the impact on the number of decisions in more detail.

As Table 1 shows, the performance of Omiga on the benchmark instances of [1] does not change much compared to the previous version without learning. This is mostly due to the fact that these benchmark instances require very few guessing, hence the positive effects of learning are small, but the additional overhead is visible. Note the cutedge sample, where the performance of Omiga with learning is surprisingly bad. Further analysis showed that here a rule is learned which requires large joins, such that the current (naive) join evaluation can not cope with it efficiently.

To give a detailed account on the effects of learning non-ground rules, we also consider a 3-colorability benchmark with varying number of nodes (10 and

[2] Benchmark results for all solvers but Omiga with learning taken from [1]. All benchmarks used the same machine and the same instances, only the OS version differs.

	reach		3col				stratProg				cutedge			
	1	4	10-18		20-38		200		400		100-30		100-50	
c	0.33	5.00	0.00	**0.00**	0.00	**0.00**	0.46	**0.46**	2.06	**2.05**	25.85	**27.34**	75.06	**79.26**
d	0.44	4.56	0.00	**0.00**	0.00	**0.00**	5.88	**5.67**	46.93	**47.78**	107.07	**214.67**	301.54	**600.08**
a	2.84	—	0.01	**1.06**	—	—	0.01	**0.08**	0.07	**0.33**	1.70	**16.70**	4.62	**46.02**
o	1.20	15.53	0.16	0.35	1.97	5.37	0.38	0.65	0.61	1.32	0.77	3.05	0.85	3.53
o+	1.14	20.64	1.12	**0.47**	2.28	**8.63**	0.37	**0.70**	0.56	**1.06**	0.97	**35.88**	0.90	—

Table 1. Runtime evaluation of common solvers (**c**: clingo, **d**: DLV, **a**: ASPeRix, **o**: Omiga, **o+**: Omiga with learning), time in seconds.[2]

	Nodes	10			25			
	Answer Sets	1	2	10	1	2	10	20
With learning	Overall decisions	31	52	686	58	240	42738	168894
	Max decision level	32	33	37	59	63	72	75
	Rules learned	2	9	19	5	19	19	19
No learning	Overall decisions	50	1048642	—	125	—	—	—
	Max decision level	51	51	—	126	—	—	—

Table 2. The effect of learning on number of decisions and maximum decision level.

25) and requested answer-sets (1,2, 10, and 20). The encoding and the instances stem from a training instance of the 2013 ASP-competition. Overall decisions counts the overall number of times that Omiga guessed that some rule fires.

As Table 2 shows, the effect of learning dramatically improves the solving capabilities of Omiga. While for a small graph, there is not much difference, if learning is disabled, but the effect of learning is fully visible when the first two answer-sets are requested: with learning, 52 decisions are necessary, while without learning, it takes 1.048.642 decisions to compute the first two answer-sets. In case of 10 requested answer-sets, with learning there are about 700 decisions required, while without learning, the solver is unable to compute them within 10 minutes. Considering a graph a graph with 25 nodes, we see again that the first answer-set can be computed with ease regardless if learning is disabled. But here, without learning, even the computation of 2 answer-sets is not possible within 10 minutes. With learning, the first 20 answer-sets can be computed.

One can also see that after the second answer-set, there are no more rules learned by Omiga. This shows that learning non-ground rules is able to deduce all implicit information in a very concise and efficient manner. Where solvers with pre-grounding have to forget learned information, this is far less of a problem if non-ground rules are learned. On the other hand, the number of necessary decisions rises exponentially after those 19 rules are learned. Here, other learning strategies (e.g., first-UIP learning like clasp [4]) might be better than the one currently implemented. The experiments also showed that Omiga learns many rules twice, which requires a more detailed analysis of the underlying reasons and is left for future work.

In summary, the performance of Omiga with learning is still far away from the performance of state of-the-art solvers like clasp and DLV on NP-hard

benchmark problems. But conflict-driven learning can be a significant step towards that performance. Also note that clasp and DLV use many optimization techniques and heuristics, while Omiga just implements one specific version of conflict-driven learning.

6 Conclusion and Future Work

In this paper we presented a new approach to learn non-ground rules during answer-set search. The approach transfers techniques for conflict-driven learning to the non-ground case. As our initial experiments show, the impact for ASP programs with many choices (e.g., in NP-hard problems like 3-colorability), is dramatic: even for small instances, the necessary number of guesses can drop from over a million down to a few dozen with only a handful of newly learned non-ground rules. Due to the additional overhead of book-keeping, which is required for learning, the Omiga solver with learning runs a bit slower on instances with very few guesses.

For future work, we plan do improve on the underlying Rete, e.g. make Rete re-use nodes when possible, improve the performance of the working memory (WM) also for rules with many variables. The learning approach itself also can be improved, e.g., by employing other learning strategies like first-UIP learning.

References

1. Dao-Tran, M., Eiter, T., Fink, M., Weidinger, G., Weinzierl, A.: Omiga : An open minded grounding on-the-fly answer set solver. In: JELIA. pp. 480–483 (2012)
2. Faber, W., Leone, N., Perri, S.: The intelligent grounder of DLV. In: Erdem, E., Lee, J., Lierler, Y., Pearce, D. (eds.) Correct Reasoning. Lecture Notes in Computer Science, vol. 7265, pp. 247–264. Springer (2012)
3. Forgy, C.: Rete: A fast algorithm for the many patterns/many objects match problem. Artif. Intell. 19(1), 17–37 (1982)
4. Gebser, M., Kaufmann, B., Neumann, A., Schaub, T.: *clasp* : A conflict-driven answer set solver. In: LPNMR. pp. 260–265 (2007)
5. Lefèvre, C., Nicolas, P.: A first order forward chaining approach for answer set computing. In: LPNMR. pp. 196–208 (2009)
6. Lefèvre, C., Nicolas, P.: The first version of a new asp solver : Asperix. In: LPNMR. pp. 522–527 (2009)
7. Leone, N., Pfeifer, G., Faber, W., Eiter, T., Gottlob, G., Perri, S., Scarcello, F.: The DLV system for knowledge representation and reasoning. ACM Transactions on Computational Logic 7, 499–562 (2002)
8. Liu, L., Pontelli, E., Son, T.C., Truszczynski, M.: Logic programs with abstract constraint atoms: The role of computations. In: ICLP. pp. 286–301 (2007)
9. Palù, A.D., Dovier, A., Pontelli, E., Rossi, G.: Gasp: Answer set programming with lazy grounding. Fundam. Inform. 96(3), 297–322 (2009)
10. Silva, J.P.M., Sakallah, K.A.: Grasp: A search algorithm for propositional satisfiability. IEEE Trans. Computers 48(5), 506–521 (1999)

On the Automated Selection of ASP Instantiators*

Marco Maratea[1], Luca Pulina[2], and Francesco Ricca[3]

[1] DIBRIS, Univ. degli Studi di Genova, Viale F. Causa 15, 16145 Genova, Italy
marco@dist.unige.it
[2] POLCOMING, Univ. degli Studi di Sassari, Viale Mancini 5, 07100 Sassari, Italy
lpulina@uniss.it
[3] Dip. di Matematica ed Informatica, Univ. della Calabria, Via P. Bucci, 87030 Rende, Italy,
ricca@mat.unical.it

Abstract. Answer Set Programming (ASP) is a powerful language for knowledge representation and reasoning. ASP is exploited in real-world applications and is also attracting the interest of industry thanks to the availability of efficient implementations. ASP systems compute solutions relying on two modules: an *instantiator* (or *grounder*) that produces, by removing variables from the rules, a ground program equivalent to the input one; and a *model generator* (or *solver*) that computes the solutions of such propositional program. In this paper we make a first step toward the exploitation of automated selection techniques to the grounding module. We rely on two well-known ASP grounders, namely the grounder of the DLV system and GRINGO, and we leverage on automated classification algorithms to devise and implement an automatic procedure for selecting the "best" grounder for each problem instance. An experimental analysis, conducted on benchmarks and solvers from the 3rd ASP Competition, shows that our approach improves the evaluation performance independently from the solver associated with our grounder selector.

1 Introduction

In the last decade, Answer Set Programming (ASP) [4, 8, 14, 15, 27, 29] is increasingly attracting the interest of industry thanks to the availability of efficient implementations – see, e.g., [16]. Thus, the ability of ASP systems to compute solutions in an efficient way is a factor of paramount importance, especially when they deal with industrial level problems. Given a non-ground ASP program, ASP systems compute solutions relying on two main modules: an *instantiator* (or *grounder*) that produces a propositional ASP program, and a *model generator* (or *solver*) that takes as input a propositional program and returns a solution.

Recently, the application of automated algorithm selection techniques to ASP solving [11, 3, 17, 25, 33] has noticeably improved the performance of ASP systems. The approaches cited above are often obtained by importing to ASP techniques already applied to Constraint Satisfaction problems, propositional satisfiability (SAT) or Quantified SAT (see [30, 37, 32] for details). The drawback about the adoption of such tech-

* This paper is an early version of the work submitted at the 13rd Conference of the Italian Association for Artificial Intelligence (AI*IA 2013).

niques is that they are confined to the model generator module, mainly because the research fields mentioned deal with inherently ground problems.

In ASP it is well-established that limiting the choice to only one grounder could avoid the exploitation of several optimization techniques, possibly applied to different problem class (e.g., Polynomial (*P*) and *NP* problem classes as classified in the 3rd ASP Competition [6]), implemented only in one grounder (e.g. Magic sets [1] in the presence of queries).

In this paper we make a first step toward the exploitation of automated selection techniques to the grounding module. We rely on two well-known ASP grounders, namely the grounder of the DLV system [22] (DLV-G in the following) and GRINGO [13], and we leverage on automated classification algorithms to automatically select the "best" grounder. More in details, our starting point is an experimental analysis conducted on the domains of benchmarks belonging to both *P* and *NP* classes of the 3rd ASP Competition, involving state-of-the-art ASP solvers and the aforementioned grounders. We then applied classification methods by relying on characteristics of the various encoding (*features*), with the aim of automatically select the most appropriate grounder. We then implemented a system based on these ideas, and the results of our experimental analysis show that the performance of the considered solvers are boosted by the usage of the proposed system.

To sum up, the main contributions of this paper are:

- the application of automated selection techniques to the grounding module in ASP computation, to complement a recent body of research only focused on solvers;
- the implementation of a system based on these techniques; and
- an experimental analysis of the new system, involving a large variety of benchmarks, grounders and solvers, that shows the benefits of the approach.

The paper is structured as follows. Section 2 introduces needed preliminaries about ASP. Section 3 shows the main answer set computation methods, with focus on the grounding module. Section 4 then describes the ideas we have applied to reach the above-mentioned goals. Section 5 presents implementation details of the system implemented along the line reported in the previous section, and its results. The paper ends with some conclusions in Section 6.

2 Answer Set Programming

In this section we recall Answer Set Programming syntax and semantics.

Syntax. A variable or a constant is a *term*. An *atom* is $p(t_1, ..., t_n)$, where p is a *predicate* of arity n and $t_1, ..., t_n$ are terms. A *literal* is either a *positive literal* p or a *negative literal* `not` p, where p is an atom. A *(disjunctive) rule* r has the following form:

$$a_1 \vee \ldots \vee a_n \text{ :- } b_1, \ldots, b_k, \texttt{ not } b_{k+1}, \ldots, \texttt{ not } b_m. \quad (1)$$

where $a_1, \ldots, a_n, b_1, \ldots, b_m$ are atoms. The disjunction $a_1 \vee \ldots \vee a_n$ is the *head* of r, while the conjunction $b_1, \ldots, b_k, \texttt{not } b_{k+1}, \ldots, \texttt{not } b_m$ is the *body* of r. A rule having

precisely one head literal (i.e. $n = 1$) is called a *normal rule*. If the body is empty (i.e. $k = m = 0$), it is called a *fact*, and the $:-$ sign is usually omitted. A rule without head literals (i.e. $n = 0$) is usually referred to as an *integrity constraint*. A rule r is *safe* if each variable appearing in r appears also in some positive body literal of r.

An *ASP program* $\mathcal{P}$ is a finite set of safe rules. A `not` -free (resp., $\vee$-free) program is called *positive* (resp., *normal*). A term, an atom, a literal, a rule, or a program is *ground* if no variables appear in it.

Hereafter, we denote by $H(r)$ the set $\{a_1, \ldots, a_n\}$ of the head atoms, and by $B(r)$ the set $\{b_1, \ldots, b_k, \texttt{not}\ b_{k+1}, \ldots, \texttt{not}\ b_m\}$ of the body literals. $B^+(r)$ (resp., $B^-(r)$) denotes the set of atoms occurring positively (resp., negatively) in $B(r)$. A predicate p is referred to as an *EDB* predicate if, for each rule r having in the head an atom whose name is $p \in H(r)$, r is a fact; all others predicates are referred to as *IDB* predicates. The set of facts in which *EDB* predicates occur, denoted by $EDB(\mathcal{P})$, is called *Extensional Database (EDB)*, the set of all other rules is the *Intensional Database (IDB)*.

Semantics. Let $\mathcal{P}$ be an ASP program. The *Herbrand universe* of $\mathcal{P}$, denoted as $U_\mathcal{P}$, is the set of all constants appearing in $\mathcal{P}$. In the case when no constant appears in $\mathcal{P}$, an arbitrary constant is added to $U_\mathcal{P}$. The *Herbrand base* of $\mathcal{P}$, denoted as $B_\mathcal{P}$, is the set of all ground atoms constructable from the predicate symbols appearing in $\mathcal{P}$ and the constants of $U_\mathcal{P}$. Given a rule r occurring in a program $\mathcal{P}$, a *ground instance* of r is a rule obtained from r by replacing every variable X in r by $\sigma(X)$, where σ is a substitution mapping the variables occurring in r to constants in $U_\mathcal{P}$. We denote by $Ground(\mathcal{P})$ the set of all the ground instances of the rules occurring in $\mathcal{P}$.

An *interpretation* for $\mathcal{P}$ is a set of ground atoms, that is, an interpretation is a subset I of $B_\mathcal{P}$. A ground positive literal A is true (resp., false) w.r.t. I if $A \in I$ (resp., $A \notin I$). A ground negative literal $\texttt{not}\ A$ is true w.r.t. I if A is false w.r.t. I; otherwise $\texttt{not}\ A$ is false w.r.t. I. Let r be a rule in $Ground(\mathcal{P})$. The head of r is true w.r.t. I if $H(r) \cap I \neq \emptyset$. The body of r is true w.r.t. I if all body literals of r are true w.r.t. I (i.e., $B^+(r) \subseteq I$ and $B^-(r) \cap I = \emptyset$) and otherwise the body of r is false w.r.t. I. The rule r is *satisfied* (or true) w.r.t. I if its head is true w.r.t. I or its body is false w.r.t. I. A *model* for $\mathcal{P}$ is an interpretation M for $\mathcal{P}$ such that every rule $r \in Ground(\mathcal{P})$ is true w.r.t. M. A model M for $\mathcal{P}$ is *minimal* if there is no model N for $\mathcal{P}$ such that N is a proper subset of M. The set of all minimal models for $\mathcal{P}$ is denoted by $\mathrm{MM}(\mathcal{P})$. In the following, the semantics of ground programs is first given, then the semantics of general programs is given in terms of the answer sets of its instantiation. Given a *ground* program $\mathcal{P}$ and an interpretation I, the *reduct* of $\mathcal{P}$ w.r.t. I is the subset $\mathcal{P}^I$ of $\mathcal{P}$ obtained by deleting from $\mathcal{P}$ the rules in which a body literal is false w.r.t. I.[4] Let I be an interpretation for a ground program $\mathcal{P}$. I is an *answer set* (or stable model) for $\mathcal{P}$ if $I \in \mathrm{MM}(\mathcal{P}^I)$ (i.e., I is a minimal model for the program $\mathcal{P}^I$) [9].

Queries. A program $\mathcal{P}$ can be coupled with a *query* in the form $q?$, where q is a literal. Let $\mathcal{P}$ be a program and $q?$ be a query, $q?$ is *true* iff for any answer set A of $\mathcal{P}$ it holds that $q \in A$. Basically, the semantics of queries corresponds to cautious reasoning, since a query is true if the corresponding atom is true in all answer sets of P.

[4] This definition, introduced in [9], is equivalent to the one of Gelfond and Lifschitz [15].

3 Answer Sets Computation

In this section we overview the evaluation of ASP programs, and recall the available solutions mentioning the techniques underlying the state of the art implementations.

The evaluation of ASP programs is traditionally carried out in two phases: program instantiation and model generation. As a consequence, an ASP system usually couples two modules: the *grounder* or *instantiator* and the *ASP model generator* or *solver*. In the following we provide a more detailed description of the instantiation, since the target of this work is improving performance of this phase.

ASP Program Instantiation. In general, an ASP program $\mathcal{P}$ contains variables, and the process of *instantiation or grounding* aims to eliminate these variables in order to generate a propositional ASP program equivalent to $\mathcal{P}$. Note that, the full theoretical instantiation $Ground(\mathcal{P})$ introduced in previous section contains all the ground rules that can be generated applying every possible substitution of variables. A modern instantiator module does not produce the full ground instantiation $Ground(\mathcal{P})$ (which is unnecessarily huge in size), but employees several techniques to produce one that is both equivalent and usually much smaller than $Ground(\mathcal{P})$. Notice that grounding is an EXPTIME-hard task, indeed in general it may produce a program that is of exponential size w.r.t. the input program. Thus, having an instantiator able to produce a comparatively small program in a reasonable time is crucial to achieve good (or even acceptable) performance in evaluating ASP programs. For instance, DLV-G generates a ground instantiation that has the same answer sets as the full one, but is much smaller in general [22].

In order to generate a small ground program equivalent to $\mathcal{P}$, a modern instantiator usually exploits some structural information on the input program. The evaluation proceeds bottom-up, starting from the information contained in the facts and evaluating the rules according to the positive body-to-head dependencies. Such dependencies can be identified by mean of the *Dependency Graph* (DG) of $\mathcal{P}$. The DG of $\mathcal{P}$ is a directed graph $G(\mathcal{P}) = \langle N, E \rangle$, where N is a set of nodes and E is a set of arcs. N contains a node for each IDB predicate of $\mathcal{P}$, and E contains an arc $e = (p, q)$ if there is a rule r in $\mathcal{P}$ such that q occurs in the head of r and p occurs in a positive literal of the body of r. The graph $G(\mathcal{P})$ induces a subdivision of $\mathcal{P}$ into subprograms (also called *modules*) allowing for a modular evaluation. We say that a rule $r \in \mathcal{P}$ *defines* a predicate p if p appears in the head of r. For each strongly connected component (SCC) C of $G(\mathcal{P})$, the set of rules defining all the predicates in C is called *module* of C and is denoted by $\mathcal{P}_c$.

The DG induces a partial ordering among its SCCs which is followed during the evaluation. Basically, this order allows to perform a layered evaluation of the program; one module at time in such a way that data needed for the instantiation of a module C_i have been already generated by the instantiation of the modules preceding C_i. This way, ground instances of rules are generated using only atoms which can possibly be derived from $\mathcal{P}$, and thus avoiding the combinatorial explosion that may occur in the case of a full instantiation [21]. Modules containing recursive rules are evaluated according to fix-point techniques originally introduced in the field of deductive databases [36]. In turn, each rule in a module is processed by applying a variable to constant matching

procedure, which is basically implemented as a backtracking algorithm. Actually, modern instantiators implement a backjumping technique [21]. Note that the instantiation of a rule is very similar to the evaluation of a conjunctive query, which is a process exponential both in the size of the query (number of elements in the body) and in the number of variables. An additional aspect influencing the cost of evaluating a rule, is the way variables are bound (trough joins or builtin operators), as it also happens for conjunctive queries [36].

At the time of this writing, two are the most prominent instantiators for ASP programs, which are capable of parsing the core language employed in the 3rd ASP competition, namely DLV-G and GRINGO. These two grounders are both based on the above-mentioned techniques, but employ specific variants and heuristics which are described in the related literature [22, 31, 13], and that will be outlined in Section 5 when the results are analyzed.

Answer Set Solving. The subsequent computations, which constitute the non-deterministic part of ASP programs evaluation, are then performed on the ground instantiation by an *ASP solver*. ASP solvers employ algorithms very similar to SAT solvers, i.e., specialized variants of DPLL [7] search.

There are several different approaches to ASP solving that range from native solvers (i.e., implementing ASP-specific techniques), to rewriting-based solutions (e.g., rewriting programs into SAT and calling a SAT solver). Among the ones that participated to the 3rd ASP Competition, we recall the native ASP solvers SMODELS [34], DLV [22] and CLASP [12]. SMODELS is one the first ASP systems made available; and DLV is one of the first robust implementations able to cope with disjunctive programs. Both feature look-ahead based techniques and ASP-specific search space pruning operators. CLASP is a native ASP solver relying on conflict-driven nogood learning. Among the rewriting-based ASP solvers we mention CMODELS [23], IDP [28], and the LP2SAT [18] family that resort on a translation to SAT. There are also proposals, like the LP2DIFF [19] family, rewriting ASP in difference logic and calling a Satisfiability Modulo Theories solver to compute answer sets.

4 Automated selection of grounding algorithm

In our previous work [26, 25], our aim was to build an efficient ASP solver on top of state-of-the-art systems, leveraging on machine learning techniques to automatically choose the "best" available solver on a per-instance basis. Our analysis focused on *ground* instances, and, to do that, we ran each non-ground instance with GRINGO – the same setting used in the 3rd ASP Competition –, letting our system ME-ASP to choose the best solver to fire. In order to extend ME-ASP to cope with non-ground instances, and considering that in [26] we report that ME-ASP was not able to cope with a number of instances due to GRINGO failures during the grounding stage, to obtain a more efficient system we investigate the application of algorithm selection techniques to the grounding phase, by relying on DLV-G and GRINGO.

Considering the grounders described above, it is not clear which one represents the choice that allows to reach the best possible performance. In the context of the

Problem	Class	Problem	Class
DisjunctiveScheduling	*NP*	HydraulicLeaking	*P*
HydraulicPlanning	*P*	GrammarBasedIE	*P*
GraphColouring	*NP*	HanoiTower	*NP*
KnightTour	*NP*	MazeGeneration	*NP*
Labyrinth	*NP*	MCSQuerying	*NP*
Numberlink	*NP*	PackingProblem	*NP*
PartnerUnitsPolynomial	*P*	Reachability	*P*
SokobanDecision	*NP*	Solitaire	*NP*
StableMarriage	*P*	WeightAssignmentTree	*NP*

Table 1. Pool of ASP problems involved in the reported experiments. Notice that "GrammarBasedIE" and "MCSQuerying" are shorthands for the problems named "GrammarBasedInformationExtraction" and "MultiContextSystemQuerying", respectively.

3rd ASP Competition, GRINGO has been used as grounder for all participant solvers, mainly because it features an easy numeric format (i.e. the one of LPARSE [35]), that all participant solvers can read and employ.

In order to investigate this point, we design an experiment aimed to highlight the performance of a pool of state-of-the-art ASP solvers on a pool of problem instances. Concerning the solvers, we selected the pool comprised in the multi-engine solver MEASP, namely CLASP, CMODELS, DLV and IDP. As reported in [26][5], these solvers are representative of the state-of-the-art solver (SOTA), i.e., considering a problem instance, the oracle that always fares the best among available solvers.

The benchmarks considered for the experiment belong to the suite of the 3rd ASP Competition. This is a large and heterogeneous suite of benchmarks encoded in ASP-Core, which was already employed for evaluating the performance of state-of-the-art ASP solvers. That suite includes planning domains, temporal and spatial scheduling problems, combinatorial puzzles, graph problems, and a number of application domains, i.e., database, information extraction and molecular biology field [6]. In more detail, we have employed the encodings used in the System Track of the competition of all evaluated problems belonging to the categories *P* and *NP*, and all the problem instances evaluated at the competition[7]. Notice that with *instance* we refer to the complete input program (i.e., encoding+facts) to be fed to a solver for each instance of the problem to be solved. In Table 1 we report the problems involved in our experiment. We evaluated 10 instances per problem – the same ones evaluated at the System Track of the 3rd ASP Competition –, for a total amount of 180 instances.

In Table 2 we report the results of the experiment described above. All the experiments ran on a cluster of Intel Xeon E31245 PCs at 3.30 GHz equipped with 64 bit Ubuntu 12.04, granting 600 seconds of CPU time for the whole process (grounding + solving) and 2GB of memory to each system. We present Table 2 in two parts – top

[5] A preliminary version is available for download at [24].

[6] An exhaustive description of the benchmark problems can be found in [5].

[7] Both encodings and problem instances are available at the competition website [5].

	CLASP				CMODELS			
	DLV-G		GRINGO		DLV-G		GRINGO	
	#	Time	#	Time	#	Time	#	Time
DisjunctiveScheduling	10	425.20	5	75.45	9	977.82	4	947.17
GrammarBasedIE	10	2323.72	10	254.33	10	2344.88	10	266.69
GraphColouring	3	20.90	3	144.55	4	423.54	4	357.90
HanoiTower	6	751.65	7	1058.87	7	314.03	7	137.11
HydraulicLeaking	10	2095.85	7	2819.15	10	2087.08	7	2850.30
HydraulicPlanning	10	883.17	10	154.30	10	878.80	10	164.57
KnightTour	7	148.76	7	82.52	6	71.19	6	18.50
Labyrinth	9	720.53	10	237.50	8	399.77	9	580.40
MazeGeneration	10	35.54	10	4.94	10	120.03	10	5.94
MCSQuerying	10	136.38	10	130.82	10	155.84	10	133.55
Numberlink	8	254.06	7	9.64	4	161.19	4	353.41
PackingProblem	9	2285.40	–	–	9	2247.19	–	–
PartnerUnitsPolynomial	8	263.70	2	63.94	8	262.93	–	–
Reachability	9	528.69	6	110.50	8	427.97	5	439.69
SokobanDecision	10	425.09	10	512.59	10	814.07	10	893.03
Solitaire	3	206.73	2	81.51	4	303.09	3	488.84
StableMarriage	1	61.47	–	–	–	–	–	–
WeightAssignmentTree	8	416.69	1	15.62	5	1100.25	–	–

	DLV				IDP			
	DLV-G		GRINGO		DLV-G		GRINGO	
	#	Time	#	Time	#	Time	#	Time
DisjunctiveScheduling	1	35.09	1	44.60	10	415.90	5	69.36
GrammarBasedIE	10	2020.14	10	280.51	10	2285.07	10	280.53
GraphColouring	–	–	–	–	3	510.15	3	531.83
HanoiTower	–	–	–	–	8	408.31	9	474.39
HydraulicLeaking	10	1936.64	7	2830.51	10	2130.24	7	2843.08
HydraulicPlanning	10	837.25	10	164.01	10	889.13	10	173.13
KnightTour	5	711.54	5	15.81	9	1002.47	10	1092.94
Labyrinth	3	71.48	3	71.36	5	49.66	6	21.92
MazeGeneration	8	629.47	8	472.77	10	37.24	10	7.16
MCSQuerying	10	31.08	10	160.17	10	138.18	6	56.58
Numberlink	4	5.50	4	10.08	8	130.06	8	80.31
PackingProblem	8	1910.86	–	–	9	2215.05	–	–
PartnerUnitsPolynomial	1	443.06	–	–	8	264.21	–	–
Reachability	10	58.89	4	69.51	5	302.53	5	1201.39
SokobanDecision	6	182.42	6	280.80	10	1306.88	9	926.54
Solitaire	4	85.50	–	–	5	216.67	5	109.06
StableMarriage	–	–	–	–	–	–	–	–
WeightAssignmentTree	10	549.09	–	–	5	113.23	1	12.07

Table 2. Results of a pool of solvers using different grounders on the instances evaluated at the 3rd ASP Competition. Notice that "GrammarBasedIE" and "MCSQuerying" are shorthands for the problems named "GrammarBasedInformationExtraction" and "MultiContextSystemQuerying", respectively.

and bottom – organized as follows. The first column reports the problem name, and it is followed by two group of columns. Each group is labeled with the considered solver name, and it is composed of two sub-groups, denoting the grounder (groups "DLV-G" and "GRINGO"). Finally, each sub-group is composed of two columns, in which we report the total amount of solved instances and the total CPU time (in seconds) spent to solve them (columns "#" and "Time", respectively).

Looking at Table 2, concerning the results of CLASP, we report that it was able to solve 141 (out of 180) instances using the DLV-G grounder, while it tops to 107 using GRINGO. In particular, looking at the table, we can see that CLASP mainly benefits from the usage of DLV-G – in terms of total amount of solved instances – in DisjunctiveScheduling, PackingProblem, PartnerUnitsPolynomial, and WeightAssignmentTree problems. Looking at the performance of CMODELS, we can see a very similar picture; notice that DLV-G +CMODELS solves 132 instances, while GRINGO +CMODELS stops at 99. While we can find the same picture looking at IDP performance (it solves 135 formulas if coupled with DLV-G, while 104 if coupled with GRINGO), the picture changes in a noticeable way if we look at the performance of DLV, because – as expected – it performs better using its native grounder (in terms of total amount of solved instances) instead of GRINGO – it solves 100 instances instead of 68. Looking at results in which a solver solves the same number of instances with both grounders, we report that in most cases the usage of GRINGO leads to lower CPU times.

To investigate this phenomenon and possibly getting advantage on this picture, we computed some problem characteristics called *features*. This is the first proposal of *non-ground* ASP features in ASP solving: our choice is to compute "simple" but meaningful features, that are cheap-to-compute, and that can help to discriminate among problems and/or classes, in order to employ the "best" grounder on each instance considered.

Most of the features we extract are related to peculiarities of ASP that can lead to select the most appropriate grounders, e.g. if the program contains a query we want to choose DLV-G given it implements specialized techniques to deal with ASP programs with queries [1]. Such features are: fraction of non-ground rules, either normal or disjunctive, presence of queries, ground and partially grounded queries; maximum Strongly Connected Components (SCC) size, number of Head-Cycle Free (HCF) and non-HCF components, degree of support for non-HCF components; features indicating if the program is recursive, tight and stratified; and number of builtins. This set of ASP peculiar features is complemented with features that take into account other characteristics such as problem size, balancing measures and proximity to horn, and are the following: number of predicates, maximum body size, ratio of positive and negative literals in each body of non-ground rules, and its reciprocal, fraction of unary, binary and ternary non-ground rules, and fraction of horn rules.

In order to highlight the differences between DLV-G and GRINGO, we extract the features described above on the instances *submitted* at the 3rd ASP Competition, i.e., a pool of more than one thousand instances related to the problems listed in Table 1, from which we discarded the instances involved in the experiment of Table 2. In Figure 1 we report the boxplots related to the distribution of two features. The differences between DLV-G and GRINGO herewith reported motivates the work presented in the next section.

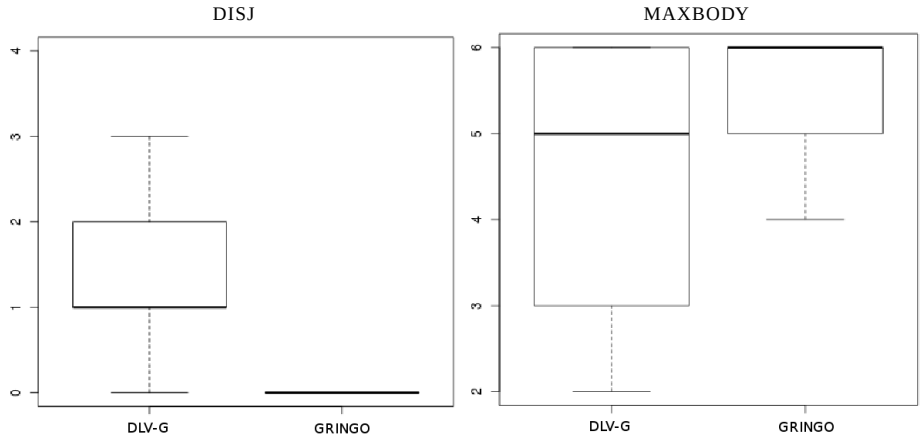

Fig. 1. Distributions of the features: number of disjunctive rules (DISJ) (left) and maximum body size (MAXBODY) (right) considering problems *submitted* to the 3rd ASP Competition for which DLV-G allows a better performance with respect to GRINGO (distribution on the left of each plot) and vice-versa (distribution of the right of each plot). For each distribution, we show a box-and-whiskers diagram representing the median (bold line), the first and third quartile (bottom and top edges of the box), the minimum and maximum (whiskers at the top and the bottom) of a distribution.

5 Implementation and Experiments

In this section we show the implementation of our automated grounder selector, representing a first attempt towards the exploitation of automated selection techniques to predict both grounder and solver, given an ASP non-ground instance. Our current implementation is composed of two main elements, namely a feature extractor able to analyze non-ground ASP programs, and a decision making module. In particular, the latter has been implemented as an if-then-else decision list, computed with the support of the PART decision list generator [10], a classifier that returns a human readable model based on if-then-else rules.

The resulting model highlights some specific cases in which there is a clear difference in performance between two grounders. DLV-G is always chosen when one has to deal with queries. GRINGO is usually preferable for instantiating non-disjunctive and recursive encodings with many components, and is preferable, in particular, when most of the rules of the encoding feature a short body. DLV-G is usually the right choice also when the encoding contains rules having large bodies (say, bodies with 4 or more literals) and the program has a simple structure (few components). The reasons behind this result can be found both in the techniques implemented in the two grounders and in some more specific implementation choices. For instance queries are processed far better by DLV-G, which exploits specific techniques like magic sets [2],

Solver	Grounder	*P*		*NP*		**Total**	
		#	Time	#	Time	#	Time
CLASP	DLV-G	48	128.26	93	62.65	141	84.99
	GRINGO	35	97.21	72	32.69	107	53.80
	SELECTOR	48	70.94	95	59.64	143	63.43
CMODELS	DLV-G	46	130.47	86	82.42	132	99.16
	GRINGO	32	116.29	67	58.44	99	77.14
	SELECTOR	46	70.60	87	80.72	133	77.22
DLV	DLV-G	41	129.17	59	71.39	100	95.08
	GRINGO	31	107.89	37	28.53	68	64.71
	SELECTOR	41	71.26	59	69.50	100	70.22
IDP	DLV-G	43	136.54	92	71.13	135	91.96
	GRINGO	32	140.57	72	46.97	104	75.77
	SELECTOR	43	74.16	94	70.19	137	71.43

Table 3. Performance of a pool of ASP solvers combined with DLV-G, GRINGO, and SELECTOR.

while this is a feature that is not well supported in GRINGO.[8] GRINGO is a newer implementation which (we argue) was initially optimized do deal with non-disjunctive encodings, since also CLASP (the solver developed by the same team) does not support disjunction. Finally, we argue that DLV-G performs better with rules having comparatively long bodies because it features both a more sophisticated indexing technique (w.r.t. GRINGO) and effective join ordering heuristics [20]; indeed, these techniques are expected to pay off especially in these cases. The grounder selector presented in this paper is available for download at `http://www.mat.unical.it/ricca/downloads/GR-SELECTOR-AIIA.zip`.

Aim of our next experiment is to test the performance of the pool of solvers introduced in Section 4 using the proposed tool as grounder (called SELECTOR in the following). Table 3 shows the results of the experiment described above on the benchmark instances evaluated at the 3rd ASP Competition. The table is structured as follows: "Solver" and "Grounder" report the solver and the grounder name, respectively; for each grounder+solver S, "P", "NP" report the number of instance solved by S ("#") and the average CPU time (in seconds) spent on such instances ("Time") related to the benchmark instances comprised in the classes *P* and *NP*, respectively.

Looking at Table 3, we can see that all the considered solvers benefit from the usage of SELECTOR as grounder. Looking at the performance of CLASP, we can see that SELECTOR+CLASP it is able to solve 2 instances more DLV-G +CLASP, and 36 instances more that GRINGO +CLASP. The increase of performance is noticeable concerning *NP* instances, while if we look at the performance of *P* instance, we report that SELECTOR+CLASP is able to solve the same instances of DLV-G +CLASP, also if in this case

[8] In the competition, as well as in our experiment, only ground queries are used and when calling GRINGO a straight technique is employed to handle propositional queries: $q?$ is replaced by the constraint `:- not` q, concluding the q is cautiously true when the resulting program has no answer set.

the average CPU time per solved instance related to SELECTOR+CLASP is 55% of the one related to DLV-G +CLASP.

Looking now at the performance of CMODELS, we can see that the picture is very similar to the one related to CLASP. SELECTOR+CMODELS solves 34 instances more that GRINGO +CMODELS, while the gap with DLV-G +CMODELS stops to 1. Similar considerations can be reported in the case of IDP. Finally, considering the performance of DLV, we can see that the picture slightly changes. In this case, SELECTOR+DLV is never superior to DLV-G +DLV– in terms of total amount of solved instances – but we report better performance in terms of average CPU time per solved instance, i.e., SELECTOR+DLV is about 25% faster than DLV-G +DLV.

6 Conclusions

ASP systems are obtained combining a grounder, which eliminates variables, and a solver, that computes the answer sets. It is well known that both components play a central role in the performance of the system. Algorithm selection techniques, up to now, have been applied only on the second component. In this paper we make a first step toward the exploitation of automated selection techniques to the grounding component.

In particular, we implemented a new system able to automatically select the most appropriate grounder for solving the instance at hand out of two state-of-the-art ASP instantiators. An experimental analysis, conducted on benchmarks and solvers from the 3rd ASP Competition, shows that our grounder selector improves the evaluation performance independently from the solver associated.

As far as future work is concerned we are exploring the possibility to implement a selector that is able to predict the best grounder+solver pair among a set of possible combinations.

References

1. Alviano, M., Faber, W., Greco, G., Leone, N.: Magic sets for disjunctive datalog programs. Artificial Intellegence 187, 156–192 (2012)
2. Alviano, M., Faber, W., Leone, N.: Disjunctive asp with functions: Decidable queries and effective computation. Theory and Practice of Logic Programming, 26th Int'l. Conference on Logic Programming (ICLP'10) Special Issue 10(4–6), 497–512 (2010)
3. Balduccini, M.: Learning and using domain-specific heuristics in ASP solvers. AI Communications – The European Journal on Artificial Intelligence 24(2), 147–164 (2011)
4. Baral, C.: Knowledge Representation, Reasoning and Declarative Problem Solving. Cambridge University Press, Tempe, Arizona (2003)
5. Calimeri, F., Ianni, G., Ricca, F.: The third answer set programming system competition (since 2011), `https://www.mat.unical.it/aspcomp2011/`
6. Calimeri, F., Ianni, G., Ricca, F., Alviano, M., Bria, A., Catalano, G., Cozza, S., Faber, W., Febbraro, O., Leone, N., Manna, M., Martello, A., Panetta, C., Perri, S., Reale, K., Santoro, M.C., Sirianni, M., Terracina, G., Veltri, P.: The Third Answer Set Programming Competition: Preliminary Report of the System Competition Track. In: Proc. of LPNMR11. pp. 388–403. LNCS Springer, Vancouver, Canada (2011)

7. Davis, M., Putnam, H.: A Computing Procedure for Quantification Theory. Journal of the ACM 7, 201–215 (1960)
8. Eiter, T., Gottlob, G., Mannila, H.: Disjunctive Datalog. ACM Transactions on Database Systems 22(3), 364–418 (Sep 1997)
9. Faber, W., Leone, N., Pfeifer, G.: Recursive aggregates in disjunctive logic programs: Semantics and complexity. In: Alferes, J.J., Leite, J. (eds.) Proceedings of the 9th European Conference on Artificial Intelligence (JELIA 2004). Lecture Notes in AI (LNAI), vol. 3229, pp. 200–212. Springer Verlag (Sep 2004)
10. Frank, E., Witten, I.H.: Generating accurate rule sets without global optimization. In: Machine Learning: Proceedings of the Fifteenth International Conference (ICML'98). p. 144. Morgan Kaufmann Pub (1998)
11. Gebser, M., Kaminski, R., Kaufmann, B., Schaub, T., Schneider, M.T., Ziller, S.: A portfolio solver for answer set programming: Preliminary report. In: Delgrande, J.P., Faber, W. (eds.) Proc. of the 11th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR). LNCS, vol. 6645, pp. 352–357. Springer, Vancouver, Canada (2011)
12. Gebser, M., Kaufmann, B., Neumann, A., Schaub, T.: Conflict-driven answer set solving. In: Twentieth International Joint Conference on Artificial Intelligence (IJCAI-07). pp. 386–392. Morgan Kaufmann Publishers, Hyderabad, India (Jan 2007)
13. Gebser, M., Schaub, T., Thiele, S.: GrinGo : A New Grounder for Answer Set Programming. In: Logic Programming and Nonmonotonic Reasoning, 9th International Conference, LPNMR 2007. Lecture Notes in Computer Science, vol. 4483, pp. 266–271. Springer, Tempe, Arizona (2007)
14. Gelfond, M., Lifschitz, V.: The Stable Model Semantics for Logic Programming. In: Logic Programming: Proceedings Fifth Intl Conference and Symposium. pp. 1070–1080. MIT Press, Cambridge, Mass. (1988)
15. Gelfond, M., Lifschitz, V.: Classical Negation in Logic Programs and Disjunctive Databases. New Generation Computing 9, 365–385 (1991)
16. Grasso, G., Leone, N., Manna, M., Ricca, F.: Asp at work: Spin-off and applications of the dlv system. In: Balduccini, M., Son, T.C. (eds.) Logic Programming, Knowledge Representation, and Nonmonotonic Reasoning - Essays Dedicated to Michael Gelfond on the Occasion of His 65th Birthday. Lecture Notes in Computer Science, vol. 6565, pp. 432–451. Springer (2011)
17. Hoos, H., Kaminski, R., Schaub, T., Schneider, M.T.: ASPeed: Asp-based solver scheduling. In: Technical Communications of the 28th International Conference on Logic Programming (ICLP 2012). LIPIcs, vol. 17, pp. 176–187. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik (2012)
18. Janhunen, T.: Some (in)translatability results for normal logic programs and propositional theories. Journal of Applied Non-Classical Logics 16, 35–86 (2006)
19. Janhunen, T., Niemelä, I., Sevalnev, M.: Computing Stable Models via Reductions to Difference Logic. In: Erdem, E., Lin, F., Schaub, T. (eds.) Logic Programming and Nonmonotonic Reasoning. Lecture Notes in Computer Science, vol. 5753, pp. 142–154. Springer Berlin / Heidelberg (2009), http://dx.doi.org/10.1007/978-3-642-04238-6_14
20. Leone, N., Perri, S., Scarcello, F.: Improving asp instantiators by join-ordering methods. In: LPNMR. LNCS, vol. 2173, pp. 280–294. Springer (2001)
21. Leone, N., Perri, S., Scarcello, F.: BackJumping Techniques for Rules Instantiation in the DLV System. In: Proceedings of the 10th International Workshop on Non-monotonic Reasoning (NMR 2004), Whistler, BC, Canada. pp. 258–266 (2004)
22. Leone, N., Pfeifer, G., Faber, W., Eiter, T., Gottlob, G., Perri, S., Scarcello, F.: The DLV System for Knowledge Representation and Reasoning. ACM Transactions on Computational Logic 7(3), 499–562 (Jul 2006)

23. Lierler, Y.: Disjunctive Answer Set Programming via Satisfiability. In: Baral, C., Greco, G., Leone, N., Terracina, G. (eds.) Logic Programming and Nonmonotonic Reasoning — 8th International Conference, LPNMR'05, Diamante, Italy, September 2005, Proceedings. Lecture Notes in Computer Science, vol. 3662, pp. 447–451. Springer Verlag (Sep 2005)
24. Maratea, M., Pulina, L., Ricca, F.: Applying machine learning techniques to ASP solving. p. 21. No. CVL 2012/003, University of Sassari Tech. Rep. (March 2012), http://eprints.uniss.it/7292/
25. Maratea, M., Pulina, L., Ricca, F.: The Multi-Engine ASP Solver ME-ASP. In: Proceedings of the 13th European Conference on Logics in Artificial Intelligence (JELIA 2012). Lecture Notes in Computer Science, vol. 7519, pp. 484–487. Springer (2012)
26. Maratea, M., Pulina, L., Ricca, F.: A Multi-Engine approach to Answer Set Programming. Theory and Practice of Logic Programming (To Appear)
27. Marek, V.W., Truszczyński, M.: Stable models and an alternative logic programming paradigm. CoRR cs.LO/9809032 (1998)
28. Mariën, M., Wittocx, J., Denecker, M., Bruynooghe, M.: Sat(id): Satisfiability of propositional logic extended with inductive definitions. In: Proc. of the 11th International Conference on Theory and Applications of Satisfiability Testing (SAT). pp. 211–224. LNCS, Springer, Guangzhou, China (2008)
29. Niemelä, I.: Logic Programs with Stable Model Semantics as a Constraint Programming Paradigm. In: Niemelä, I., Schaub, T. (eds.) Proceedings of the Workshop on Computational Aspects of Nonmonotonic Reasoning. pp. 72–79. Trento, Italy (May/June 1998)
30. O'Mahony, E., Hebrard, E., Holland, A., Nugent, C., O'Sullivan, B.: Using case-based reasoning in an algorithm portfolio for constraint solving. In: Proc. of the 19th Irish Conference on Artificial Intelligence and Cognitive Science (2008)
31. Perri, S., Scarcello, F., Catalano, G., Leone, N.: Enhancing DLV instantiator by backjumping techniques. Annals of Mathematics and Artificial Intelligence 51(2–4), 195–228 (2007)
32. Pulina, L., Tacchella, A.: A self-adaptive multi-engine solver for quantified boolean formulas. Constraints 14(1), 80–116 (2009)
33. Silverthorn, B., Lierler, Y., Schneider, M.: Surviving solver sensitivity: An asp practitioner's guide. In: Technical Communications of the 28th International Conference on Logic Programming (ICLP 2012). LIPIcs, vol. 17, pp. 164–175. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik (2012)
34. Simons, P., Niemelä, I., Soininen, T.: Extending and Implementing the Stable Model Semantics. Artificial Intelligence 138, 181–234 (Jun 2002)
35. Syrjänen, T.: Lparse 1.0 User's Manual (2002), http://www.tcs.hut.fi/Software/smodels/lparse.ps.gz
36. Ullman, J.D.: Principles of Database and Knowledge Base Systems. Computer Science Press (1989)
37. Xu, L., Hutter, F., Hoos, H.H., Leyton-Brown, K.: SATzilla: Portfolio-based algorithm selection for SAT. JAIR 32, 565–606 (2008)

List of Authors