TLP **16** (5–6): 552–569, 2016. © Cambridge University Press 2016
doi:10.1017/S1471068416000338

The power of non-ground rules in Answer Set Programming

MANUEL BICHLER, MICHAEL MORAK and STEFAN WOLTRAN
TU Wien, Vienna, Austria
(*e-mails:* {bichler,morak,woltran}@dbai.tuwien.ac.at)

submitted 6 May 2016; revised 8 July 2016; accepted 22 August 2016

Abstract

Answer set programming (ASP) is a well-established logic programming language that offers an intuitive, declarative syntax for problem solving. In its traditional application, a fixed ASP program for a given problem is designed and the actual instance of the problem is fed into the program as a set of facts. This approach typically results in programs with comparably short and simple rules. However, as is known from complexity analysis, such an approach limits the expressive power of ASP; in fact, an entire NP-check can be encoded into a single large rule body of bounded arity that performs both a guess and a check within the same rule. Here, we propose a novel paradigm for encoding hard problems in ASP by making explicit use of large rules which depend on the actual instance of the problem. We illustrate how this new encoding paradigm can be used, providing examples of problems from the first, second, and even third level of the polynomial hierarchy. As state-of-the-art solvers are tuned towards short rules, rule decomposition is a key technique in the practical realization of our approach. We also provide some preliminary benchmarks which indicate that giving up the convenient way of specifying a fixed program can lead to a significant speed-up.

KEYWORDS: answer set programming, rewriting, non-ground rules, rule decomposition

1 Introduction

Answer set programming (ASP) (Marek and Truszczyński 1999; Brewka *et al.* 2011; Gebser *et al.* 2012) is a well-established logic programming paradigm based on the stable model semantics of logic programs. Its main advantage is an intuitive, declarative language, and the fact that generally, each answer set of a given logic program describes a valid answer to the original question. Moreover, ASP solvers—see e.g. (Gebser *et al.* 2012; Alviano *et al.* 2013; Elkabani *et al.* 2005; Alviano *et al.* 2010)—have made huge strides in efficiency. A logic program usually consists of a set of logical implications (called rules) and a set of facts. Deciding the consistency problem, that is, whether a given disjunctive logic program has an answer set, is $\text{NExpTime}^{\text{NP}}$-complete in the combined complexity, where both the rules and facts are treated as input, and Σ_{P}^{2}-complete in the data complexity, where the set of rules is fixed; cf. (Dantsin *et al.* 2001).

https://doi.org/10.1017/S1471068416000338 Published online by Cambridge University Press

In practice, when problems are modelled using the ASP logic programming language, the usual goal is to write a fixed program (i.e. set of rules) that solves the general problem. The concrete input is then supplied as a set of (ground) facts. The answer set solver then takes the fixed program, plus the ground facts, and computes the answer sets, that is, the solutions to the original problem, as described earlier. Most ASP programs written in this way contain only small rules. This is for two main reasons: firstly, current solvers are much better at evaluating smaller rules, and secondly, large rules tend not to be as human-readable, and are thus often avoided. However, restricting to fixed programs prohibits exploiting the additional expressive power of large rules. The following example illustrates the usage of rules we have in mind.

Example 1

Consider a graph G over vertices a, b, c, d with edges (a, b), (b, c), (c, d), (a, d), and (b, d). The problem of three-colorability of this graph can be solved by a single, non-ground ASP rule. Let program Π contain the facts $e(r, g)$, $e(r, b)$, $e(g, b)$, $e(g, r)$, $e(b, r)$, and $e(b, g)$ plus the rule

$$\bot \leftarrow e(A, B), e(B, C), e(C, D), e(D, A), e(B, D),$$

where A, B, C, D are variables representing the graph vertices. It is easy to verify that the body of this rule is true (and thus, Π has no answer set) iff each variable can be mapped to one of the constants r, g, and b, representing the three colors, such that no two neighboring vertices are colored the same. Clearly, a corresponding transformation to the above can be applied to any graph, not just to G, resulting in a rule whose size depends on the graph.

We postulate that large rules in the spirit of Example 1 can be useful when encoding hard problems into answer set programs, at the expense of having a single, fixed program solving the problem. When encoding a problem P into ASP in this way, clearly we no longer have a fixed logic program Π solving P. Instead, a rewriting is used that transforms a specific instance I of P into a non-ground program Π_I, with the intention that every answer set of Π_I represents a specific solution to I. The rewriting algorithm is used separately for each concrete instance I of P, and thus can use arbitrarily large, instance-specific rules, that may even encode an NP-check (cf. Example 1, where a concrete instance of 3-colorability is rewritten into a corresponding rule).

It is the aim of this paper to propose a general rewriting paradigm that encodes problems into ASP by making explicit use of large, non-ground rules. Such rules can, in general, encode NP-hard checks, even when predicate arities are bounded (this follows immediately from the NP-hardness of answering conjunctive queries over databases (Chandra and Merlin 1977)). As shown by Eiter *et al.* (2007), the combined complexity of solving arbitrary ASP programs of bounded arity increases to Σ^3_{P}, but drops back to Σ^2_{P} for normal (i.e. non-disjunctive) programs. Such programs therefore fall well within the realm of practically solvable ASP instances, and we can harness the power of advanced ASP solvers to solve problems above Σ^2_{P}.

https://doi.org/10.1017/S1471068416000338 Published online by Cambridge University Press

Unfortunately, the use of large rules causes problems for current ASP solvers, since the input program needs to be grounded first (i.e. all the variables in each rule are replaced by all possible, valid combinations of constants). This grounding step can be done in polynomial time for fixed ASP encodings, but generally requires exponential time for rules of arbitrary size. In practice, the grounding time thus becomes prohibitively large very quickly. However, our encoding paradigm requires the use of large rules. In order to deal with this problem, we employ a rule decomposition algorithm that splits large rules into multiple smaller ones. This approach, based on hypertree decompositions (Gottlob *et al.* 2009) of ASP rules, was first proposed by Morak and Woltran (2012). We extend this approach to cover the full ASP syntax as specified by the ASP Standardization Working Group (ASP-Core-2 2015). When such an approach is used, the size of the largest rules is often reduced drastically, and a traditional grounding/solving approach with existing ASP solvers becomes feasible for problem encodings according to our paradigm. In fact, our encodings offers competitive performance when compared to traditional ASP encodings.

Constructions with large rules, similar to our proposed approach, have been used in several related areas—usually to establish relevant complexity results. In (Gottlob and Papadimitriou 2003), the complexity of single rule Datalog programs (sirups) has been investigated. Also, in the world of ontological reasoning and description logics, rewritings into Datalog (see e.g. (Gottlob and Schwentick 2012)) typically yield rules of large size. Eiter *et al.* (2010) present several problem encodings with large rules in order to illustrate that current solvers are not well-equipped to handle them and propose a polynomially space-bounded solving approach. The grounding bottleneck of state-of-the-art ASP solvers is widely recognized and several methods and systems have been proposed that either work directly on the non-ground rules via resolution-based methods (Bonatti *et al.* 2008), forward-chaining algorithms (Lefèvre *et al.* 2015), or lazy grounding approaches where grounding is only performed as needed (Palù *et al.* 2009; de Cat *et al.* 2012).

The main results of this paper can be summarized as follows:

- We propose a novel paradigm for problem encoding into ASP that makes use of the power of large, non-ground rules. In contrast to the classical ASP approach where a concrete problem instance I is supplied as ground facts, our method transforms I into a non-ground program making use of arbitrarily large non-ground rules. This allows us to solve problems above the class Σ_{P}^{2}, and exhibits good practical performance when compared to classical fixed encodings.
- In order to make our paradigm work well in practice, we extend the rule decomposition algorithm proposed in (Morak and Woltran 2012) to the full ASP-Core-2 syntax. This allows to feed smaller rules to existing ASP solvers, which is a key for good performance.
- We provide a number of case studies that show our proposed encoding paradigm at work. We exhibit encodings for solving several logic-related problems, including 2- and 3-QBF solving (borrowed from the hardness proofs in (Eiter *et al.* 2007)), as well as an encoding for solving ground disjunctive programs via (non-ground)

https://doi.org/10.1017/S1471068416000338 Published online by Cambridge University Press

normal programs. Finally, we consider the Σ_P^3-complete problem of stable cautious abduction (Eiter *et al.* 1997).

- We give preliminary benchmark results for 2- and 3-QBF solving, comparing the performance of the ASP encodings of the case studies to the performance of the classical (fixed) ASP encoding, as well as a state-of-the-art QBF solver. Our approach turns out to be surprisingly competitive in practice, even when compared to the dedicated QBF solver.

2 Preliminaries

Answer Set Programming. A *ground logic program* (also called answer set program, or program, for short) is a pair $\Pi = (\mathscr{A}, \mathscr{R})$, where $\mathscr{A}$ is a set of propositional (i.e. ground) atoms and $\mathscr{R}$ is a set of rules of the form:

$$a_1 \vee \cdots \vee a_l \leftarrow a_{l+1}, \ldots, a_m, \neg a_{m+1}, \ldots, \neg a_n, \tag{1}$$

where $n \geqslant m \geqslant l$ and $a_i \in \mathscr{A}$ for all $1 \leqslant i \leqslant n$. A rule $r \in \mathscr{R}$ of form (1) consists of a head $H(r) = \{a_1, \ldots, a_l\}$ and a body given by $B^+(r) = \{a_{l+1}, \ldots, a_m\}$ and $B^-(r) = \{a_{m+1}, \ldots, a_n\}$. A rule is called *normal* iff $l \leqslant 1$. A set $M \subseteq \mathscr{A}$ is a called a model of r if $B^+(r) \subseteq M$ together with $B^-(r) \cap M = \emptyset$ implies that $H(r) \cap M \neq \emptyset$. We denote the set of models of r by $models(r)$ and the models of a program $\Pi = (\mathscr{A}, \mathscr{R})$ are given by $models(\Pi) = \bigcap_{r \in \mathscr{R}} models(r)$. The reduct Π^I of a program Π with respect to a set of atoms $I \subseteq \mathscr{A}$ is the program $\Pi^I = \big(\mathscr{A}, \{H(r) \leftarrow B^+(r) \mid r \in \mathscr{R}, B^-(r) \cap I = \emptyset\}\big)$. Following Gelfond and Lifschitz (1988), $M \subseteq \mathscr{A}$ is an *answer set* of a program Π if $M \in models(\Pi)$ and for no $N \subseteq M$, $N \in models(\Pi^M)$. The set of answer sets of a program Π is denoted $AS(\Pi)$. The *consistency problem* of ASP (decide whether for a given program Π, $AS(\Pi) \neq \emptyset$) is Σ_2^P-complete (Eiter and Gottlob 1995).

General *non-ground, disjunctive logic programs* differ from ground programs in that variables may occur in rules. Such rules are $\forall$-quantified first-order implications of the form $H_1 \vee \cdots \vee H_k \leftarrow P_1, \ldots, P_n, \neg N_1, \ldots, \neg N_m$ where H_i, P_i and N_i are (non-ground) atoms, called head, positive and negative body atoms, respectively. An *atom* A is of the form $a(\mathbf{X}, \mathbf{c})$ and consists of a predicate name a, as well as a sequence of variables $\mathbf{X}$ and a sequence of constants $\mathbf{c}$ from the domain Δ, with $|\mathbf{X}| + |\mathbf{c}|$ being the arity of a. We will sometimes treat $\mathbf{X}$ and $\mathbf{c}$ as sets. For zero-arity atoms, we simply write a instead of $a()$. Let $var(A)$ denote the set of variables $\mathbf{X}$ in atom A. This notation naturally extends to sets of atoms. We will denote variables by capital letters, constants and predicates by lower-case words. A variable occurring in the positive body of a rule is said to be *safe*. A rule is said to be safe if every variable in it is safe. If not stated otherwise, we always assume that rules are safe. A non-ground rule can be seen as an abbreviation for all possible instantiations of the variables with domain elements from Δ. In ASP, this step is usually explicitly performed by a grounder that transforms a (non-ground) disjunctive logic program into a set of ground rules of the form given in (1). This process is called *grounding*. Note that in general, such a ground program can be exponential in the size of the non-ground program.

https://doi.org/10.1017/S1471068416000338 Published online by Cambridge University Press

Tree Decomposition and Treewidth. A *tree decomposition* of a graph $G = (V, E)$ is a pair $\mathcal{T} = (T, \chi)$, where T is a rooted tree and χ is a labelling function over nodes t, with $\chi(t) \subseteq V$—we call $\chi(t)$ the *bag* of t—such that the following holds: (i) for each $v \in V$, there exists a node t in T, such that $v \in \chi(t)$; (ii) for each $\{v, w\} \in E$, there exists a node t in T, such that $\{v, w\} \subseteq \chi(t)$; and (iii) for all nodes r, s, and t in T, such that s lies on the path from r to t, we have $\chi(r) \cap \chi(t) \subseteq \chi(s)$. The *width* of a tree decomposition is defined as the cardinality of its largest bag minus one. The *treewidth* of a graph G, denoted by $tw(G)$, is the minimum width over all tree decompositions of G. To decide whether a graph has treewidth at most k is NP-complete (Arnborg *et al.* 1987). For an arbitrary but fixed k, however, this problem can be solved (and a tree decomposition constructed) in linear time (Bodlaender 1996). Given a non-ground ASP rule r, we can construct its *Gaifman graph* $G = (V, E)$ as follows: Let V be the set of variables occurring in r. Let there be an edge (X, Y) in E iff X and Y occur together in the head of r, or in a body atom of r. We refer to a tree decomposition of G as the *tree decomposition of rule r*.

3 A new ASP encoding paradigm

In this section, we propose a general encoding paradigm allowing to encode hard problems into non-ground ASP. We first discuss the general method and illustrate the underlying idea by extending the coloring problem from the introduction. Secondly, we elaborate on the concept of rule decomposition which is a key ingredient towards practical efficiency.

3.1 Encoding hard problems into ASP using rule bodies

Classically, when solving problems with ASP, the idea is to find a fixed problem encoding, that is, given a problem P, we write a fixed ASP encoding Π_P that solves P. An instance I of P is then transformed into a set of ASP facts (i.e. into an input database) D_I, and an ASP solver is then used to solve the ASP program given by $\Pi_P \cup D_I$. The resulting answer sets usually represent the solutions of the original problem instance I.

In contrast, our proposed encoding paradigm facilitates the use of arbitrarily large non-ground rules. Instead of obtaining a fixed encoding for a problem P, we directly encode the instance I of P into ASP. This requires a problem-dependent rewriting algorithm, that takes an instance I of P, and generates an ASP program Π_I for I. Such a rewriting algorithm may now use, and is in fact encouraged to use, arbitrarily large rule bodies, that may encode an NP-hard check. Note that model-checking for a rule in ASP is already NP-hard (Gottlob and Papadimitriou 2003), which follows from the fact that the body can be viewed as a conjunctive query, and answering conjunctive queries is known to be NP-complete (Chandra and Merlin 1977). In the literature, constructions like this were, to date, mainly used to show theoretic complexity or expressiveness results by polynomial-time reductions, and not to solve problems in practice. The following example shows how Example 1,

https://doi.org/10.1017/S1471068416000338 Published online by Cambridge University Press

which deals with a concrete 3-colorability instance, can be extended to a rewriting algorithm that deals with all possible graphs.

Example 2
The problem of 3-colorability of an arbitrary input graph $G = (V, E)$ can be solved by a single, non-ground ASP rule. Let an ASP program Π contain the three facts $col(r).col(g).col(b).$ representing the tree colors, as well as the three facts $e(r, g).e(g, b).e(b, r).$ and their respective inverse, representing valid color pairs. Let rule r be constructed from G as follows:

$$\bot \leftarrow \bigcup_{v \in V} \{col(X_v)\} \cup \bigcup_{(v,w) \in E} \{e(X_v, X_w)\},$$

where for each vertex $v \in V$, a variable X_v represents its color. Note that after the first large union, every vertex is guaranteed to be assigned a color. The second large union encodes the structure of the graph. If vertex colors can be selected such that the graph structure can be mapped onto the valid color pairs, then the body of the rule is true, and the graph is 3-colorable. Thus, an answer set exists for Π_G iff no valid 3-coloring exists.

From the above example, we can see several ingredients needed to encode problems using our paradigm. Firstly, a (usually fixed) set of ASP facts D contains the domain of our guess (this is done by the $col(\cdot)$ facts in the above example—in particular, we want to guess colors). Further, the database contains a (usually fixed) set of facts C, representing valid, local combinations of domain elements (this is done by the $e(\cdot, \cdot)$ facts in the above example, which represent the valid color pairs that are allowed to be next to each other). Finally, we have the large rules that check for the required properties. Normally, such rules will contain at least one variable for each item to be guessed. In the above example, we want to guess a color for each vertex, and thus one variable per vertex is introduced.

In general, such large rules should follow a distinct guess-and-check pattern for easy readability. The structure of such a large, rewritten rule can thus be specified as follows:

$$head \leftarrow B_{guess} \cup B_{check},$$

where B_{guess} is a set of body literals encoding the guess, and B_{check} is a set of body literals encoding the check. As already stated, such a rule will contain a variable for each item to be guessed (this can for example be a variable per vertex to guess colors for the 3-colorability problem, or a variable per atom to guess a truth assignment for the SAT problem). The guess-part B_{guess} will contain atoms that independently map each variable to one of the domain facts D of the program. In Example 2, B_{guess} contains an atom $col(X_v)$ for each vertex v of the graph. This represents a guess, since the only way to satisfy this part of the body is when every variable X_v is assigned a color. The second part of the body, B_{check}, contains the actual information about the relation of the guessed variables. The construction of B_{check} is highly problem-dependent, but generally the guessed variables will be mapped to the fixed set C of valid combinations of domain elements. In Example 2, this is done by the atoms $e(X_v, X_w)$ which check that two variables representing two neighbouring

https://doi.org/10.1017/S1471068416000338 Published online by Cambridge University Press

vertices are assigned different colors (since C in this case only contains valid color pairs for neighboring vertices). Clearly, such large rules can be combined with other, classical ASP encoding elements. To illustrate this, the following example further extends Example 2 to a second-level coloring problem.

Example 3
Let $G = (V = V_1 \cup V_2, E)$ be a graph with its vertices partitioned into two sets V_1 and V_2. It can be easily shown that deciding whether there exists a three-coloring C for the subgraph of G induced by V_1, such that there does not exist an extension of C to V_2 being a coloring of G, is Σ_{P}^2-hard. This problem can be solved via a logic program, using the same set of facts as Example 2, and, additionally, a set of facts with the predicate $vertex_1$ to represent the set V_1. This program consists of the following two rules:

$$c(X, r) \vee c(X, g) \vee c(X, b) \leftarrow vertex_1(X), \text{ and} \qquad \bot \leftarrow edge(X_1, X_2), c(X_1, C), c(X_2, C),$$

plus a third rule r_{col} which is obtained from the rule r of Example 2 by adding the body atoms $\{c(v, X_v) \mid v \in V_1\}$. These additional body atoms require that vertices from V_1 are assigned the same colors in r_{col} as in the classical guess-and-check part of the program. It can be verified that indeed the above logic program has an answer set iff there exists a coloring on V_1 that cannot be extended to V_2 (since the body of r_{col} is true precisely when such an extension exists).

Note that the above example is a combination of a classical, fixed program encoding together with large rules according to our paradigm, which illustrates that the two approaches can be easily combined. Moreover, the programs from Example 3 are head-cycle free (and thus can be easily transformed into equivalent normal programs via shifting) and their predicates have a fixed arity; hence, we transform a Σ_{P}^2-complete variant of graph-coloring into a class of programs for which consistency is of the same complexity; cf. (Eiter *et al.* 2007). A number of extended examples how this technique can be applied can be found in Section 4.

3.2 Rule decomposition

As we have seen, ASP rules obtained by rewritings as described above may generally have rule bodies with a large number of atoms (in order to facilitate an NP-check). However, state-of-the-art ASP solvers are not well equipped to handle these large rules. Classically, the program is first converted into a propositional program by a grounder, which may already lead to problems, since the number of ground instances of a rule with a large number of body atoms and variables may be prohibitively large, as the size of the grounding is exponential in the worst case.

We thus need a method to make it feasible for large rules to be evaluated with current grounders and solvers. An approach to split up large, non-ground rules has been proposed in (Morak and Woltran 2012). Generally speaking, this approach computes the tree decomposition of a rule, and then splits the rule up into multiple, smaller rules according to this decomposition. Morak and Woltran (2012) show that,

https://doi.org/10.1017/S1471068416000338 Published online by Cambridge University Press

while in the worst case this decomposition may not change the rule at all, in practice large rules can be split up very well. For instance, in Example 3, for graphs that are sparse (or, more generally speaking, are of low treewidth), the long rule r_{col} will be amenable for such a decomposition. Let us briefly recall the algorithm from (Morak and Woltran 2012). For a given rule r, the algorithm works as follows:

1. Compute a tree decomposition $\mathscr{T} = (T, \chi)$ of r with minimal width, with all variables occurring in the head of r contained in its root node.
2. For each node n, let $temp_n$ be a fresh predicate, and the same for each variable X in r and predicate dom_X. Let $\mathbf{Y}_n = \chi(n) \cap \chi(p_n)$, where p_n is the parent node of n. For the root node $root$, let $temp_{root}$ be the entire head of r, and, accordingly, $\mathbf{Y}_{root} = var(H(r))$. Now, for a node n, generate the following rule:

$$\begin{aligned} temp_n(\mathbf{Y}_n) \leftarrow \quad & \{A \in B(r) \mid var(A) \subseteq \chi(n)\} \\ & \cup \{dom_X(X) \mid A \in B^-(r), X \in var(A), var(A) \subseteq \chi(n)\} \\ & \cup \{temp_m(\mathbf{Y}_m) \mid m \text{ is a child of } n\}. \end{aligned}$$

3. For each $X \in var(B^-(r))$, for which a dom predicate is needed to guarantee safety of a rule generated above, pick an atom $A \in B^+(r)$, such that $X \in var(A)$ and generate a rule

$$dom_X(X) \leftarrow A.$$

Replacing the rule r by the set of generated rules according to the above algorithm now guarantees that the size of the grounding of r is no longer (at worst) exponential in the size of the rule, but only exponential in the treewidth of r. Note that the rule decomposition algorithm may also increase the arity of bounded arity programs, since the arity of temporary predicates depends on the treewidth. However, it can be shown that, since these atoms occur nowhere except non-negated in the generated rule bodies without internal recursion, the number of possible answer sets does not increase, and thus the complexity of bounded arity ASP programs is preserved.

Example 4

Given the rule $h(X, W) \leftarrow e(X, Y), e(Y, Z), \neg e(Z, W), e(W, X)$, we may compute the following tree decomposition consisting of two nodes: the root node, containing $\{h(X, W), e(X, Y), e(W, X)\}$, and a child node, containing $\{e(Y, Z), \neg e(Z, W)\}$. Based on this decomposition, the rule decomposition algorithm yields the following three rules:

- $dom_W(W) \leftarrow e(W, X)$;
- $t_1(Y, W) \leftarrow e(Y, Z), \neg e(Z, W), dom_W(W)$; and
- $h(X, W) \leftarrow e(X, Y), e(X, W), t_1(Y, W)$;

where t_1 is a fresh predicate not appearing anywhere else.

The above algorithm only deals with rules consisting of simple atoms. However, in practice, the full answer set syntax contains further, more complicated atoms; cf. (ASP-Core-2 2015). In the following, we will explain how the above algorithm

https://doi.org/10.1017/S1471068416000338 Published online by Cambridge University Press

can be extended to the full ASP syntax. Several extensions are straightforward (e.g. rules with upper or lower bounds on the body or head atoms). We will therefore focus on two extensions that warrant a separate explanation. Our implementation of the extended rule decomposition algorithm as described above can be found here: `http://dbai.tuwien.ac.at/proj/lpopt`; a detailed system description is given in (Bichler 2015). Note that, since computing an optimal tree decomposition w.r.t. width can be NP-hard, our implementation employs a heuristic approach for the first step of the above algorithm; more details can be found in (Dermaku *et al.* 2008).

Arithmetics. An atom in the full language of ASP may be an *arithmetic atom* of the form $X = \varphi$, where φ is an arithmetic expression between variables and numbers, connected with the mathematical connectives $+$, $-$, $\times$, and $\div$. Such an expression impacts the safety of the generated rules, since X is safe iff all variables in φ are safe. In order to deal with this, we change steps 2 and 3 of the decomposition algorithm above as follows.

In step 2, a *dom*-predicate is also added for all variables occurring on the right-hand side of an arithmetic expression. In step 3, it may now be the case that no atom $A \in B^+(r)$ exists that contains the variable X. In this case, pick the smallest arithmetic atom of the form $X = \varphi$. For each variable Y_i in φ, now repeat this procedure: Try to pick an atom from $B^+(r)$ to make Y_i safe. If no such atom exists, pick the smallest arithmetic atom of the form $Y_i = \psi$ not already selected. This procedure necessarily terminates, and selects a set of atoms $\mathbf{A}$ from r. We generate the rule $dom_X(X) \leftarrow \mathbf{A}$. It is easy to see that this rule is safe, and describes the possible domain of variable X, as required.

Aggregates. An aggregate is an atom A of the form $\#agg\{\mathbf{X} : \Psi(\mathbf{X}, \mathbf{Y}, \mathbf{Z})\}$, where Ψ is a conjunction of non-ground literals over the variables in $\mathbf{X}$, $\mathbf{Y}$ and $\mathbf{Z}$. Intuitively, the variables in $\mathbf{X}$ are used for aggregation. $\mathbf{Y}$ are those variables that also occur outside the aggregate, and $\mathbf{Z}$ are those variables appearing within the aggregate only. Such an aggregate atom A is rewritten as follows.

First, replace A by an aggregate atom A' that only preserves those atoms within A that have connections to the rest of the rule r containing A. To this end, let A' be of the form $\#agg\{\mathbf{X} : \Psi'(\mathbf{X}, \mathbf{Y}, \mathbf{W}), temp_A(\mathbf{X}, \mathbf{Y}, \mathbf{W})\}$, where $\Psi'(\mathbf{X}, \mathbf{Y}, \mathbf{W}) = \{B \in \Psi \mid var(B) \cap \mathbf{X} \neq \emptyset \vee var(B) \cap \mathbf{Y} \neq \emptyset\}$, $temp_A$ is a fresh predicate, and $\mathbf{W} = (\mathbf{Z} \cap var(\Psi'))$, that is, all those variables from $\mathbf{Z}$ occurring in Ψ'. Then, create a temporary rule $r_A = temp_A(\mathbf{X}, \mathbf{Y}, \mathbf{W}) \leftarrow (\Psi \setminus \Psi' \cup \Psi_{dom})$, that is, r_A contains all the atoms of A that have no connection to the rest of r, plus the set Ψ_{dom} containing dom_X-predicates for each otherwise unsafe variable X in r_A. Finally, recursively execute the rule decomposition algorithm on r_A. Since aggregates may be arbitrarily large, this allows us to decompose large aggregates to a smaller aggregate and a set of smaller rules.

4 Case studies

In this section we exhibit a number of interesting problems that can be solved using our proposed encoding paradigm. The problems include evaluating 2-QBF

https://doi.org/10.1017/S1471068416000338 Published online by Cambridge University Press

and 3-QBF formulas, as well as solving disjunctive ground ASP itself. The latter is then extended to an encoding for abduction.

4.1 Rewriting QSAT into logic programs

Before introducing the encodings according to our paradigm, we give a classic, fixed-program encoding for 2-QBFs in the spirit of the encoding used in the Σ_{P}^2-completeness proof from (Eiter and Gottlob 1995). This encoding will be used for comparison in the benchmarks in Section 5. Let Φ be a fully quantified 2-QBF of the form

$$\Phi = \forall x_1, \ldots, x_m \, \exists y_1, \ldots, y_n \;\; (c_1 \wedge \ldots \wedge c_k), \tag{2}$$

where clauses are of the form $c_i = L_{i,1} \vee L_{i,2} \vee L_{i,3}$ with literals $L_{i,j}$ over $\{x_1, \ldots, x_m, y_1, \ldots, y_n\}$. We construct Π^Φ which consists of the following fixed set of rules:

- $ass(X,1) \vee ass(X,0) \leftarrow var(X)$;
- $ass(X,0) \leftarrow sat, exists(X)$;
- $ass(X,1) \leftarrow sat, exists(X)$;
- $sat \leftarrow \bigcup_{i \leqslant 3} \{pos_i(C, X_i, A_i), ass(X_i, 1 - A_i)\}$;
- $\bot \leftarrow \neg sat$;

and facts derived from Φ as follows: a fact $var(x_i)$ for each variable x_i, a fact $exists(x_i)$ if x_i is existentially quantified, and a fact $pos_\ell(c_i, x_j, a_\ell)$ for each occurrence of a variable x_j in a clause c_i at position $\ell \in \{1,2,3\}$, where a_ℓ is 0 if x_j appears negated in c_i; otherwise a_ℓ is 1. It can be checked that Π^Φ has an answer set iff Φ is unsatisfiable.

A different reduction algorithm for QBFs of the form (2) has been proposed in (Eiter *et al.* 2007), which in fact makes use of the paradigm proposed in Section 3. Let Φ be a QBF as before. For ease of notation, let $X(c_i) = \{L \mid L \text{ is a literal of form } x_j \text{ or } \neg x_j \text{ in } c_i\}$, and let $Y(c_i)$ be the ordered tuple of the literals in c_i over the variables $\{y_1, \ldots, y_n\}$. Further, let $\eta(c_i)$ be the tuple obtained by replacing each literal of $Y(c_i)$ with a corresponding (ASP) variable, ignoring negation, and let $\overline{c_i}$ denote the tuple resulting from $Y(c_i)$ by replacing each positive literal in $Y(c_i)$ by 0 and each negative literal by 1. For example, for $c = \neg y_1 \vee x_2 \vee y_3$, we have $Y(c) = (\neg Y_1, Y_3)$, $\eta(c) = (Y_1, Y_3)$, and $\overline{c} = (1,0)$. Let Π^Φ contain the following rules:

- $t(x_i) \vee f(x_i)$, for each $i \in \{1, \ldots, m\}$;
- $c_i(\mathbf{t}) \leftarrow t(x_j)$, for each $i \in \{1, \ldots, k\}$, tuple $\mathbf{t} \in \{0,1\}^{|Y(c_i)|}$, and positive literal $x_j \in X(c_i)$;
- $c_i(\mathbf{t}) \leftarrow f(x_j)$, for each $i \in \{1, \ldots, k\}$, tuple $\mathbf{t} \in \{0,1\}^{|Y(c_i)|}$, and negative literal $\neg x_j \in X(c_i)$;
- $c_i(\mathbf{t})$, for each $i \in \{1, \ldots, k\}$ and tuple $\mathbf{t} \in \{0,1\}^{|Y(c_i)|} \setminus \{\overline{c_i}\}$; and
- $\bot \leftarrow c_1(\eta(c_1)), \ldots, c_k(\eta(c_k))$.

Note that predicate arities in the above construction are bounded by the constant 3 and the last rule is the only rule containing variables, and has a body with size linear in the size of Φ. Moreover, the program is head-cycle free and thus can equivalently

https://doi.org/10.1017/S1471068416000338 Published online by Cambridge University Press

be given as normal program. From (Eiter *et al.* 2007), we again have that Π^Φ has an answer set iff Φ is unsatisfiable.

In (Eiter *et al.* 2007) the above approach is then extended to solve third-level QBFs, effectively reducing a Σ_P^3-complete problem to disjunctive ASP. To this end, Φ be a 3-QBF as follows:

$$\Phi = \exists x_1, \ldots, x_l \forall x_{l+1}, \ldots x_m \exists y_1 \ldots y_n (c_1 \wedge \ldots \wedge c_k).$$

A program Π^Φ can be constructed from Φ as before (by treating all x_i variables as if they were universally quantified). By replacing the $\bot$ symbol with an atom *sat*, adding the rules $\bot \leftarrow \neg sat$ and $p \leftarrow sat$ for each atom $p \in \{t(x_i), f(x_i) \mid l < i \leqslant m\} \cup \{c_i(t) \mid 0 < i \leqslant k, t \in \{0,1\}^{|Y(c_i)|}\}$, the 2-QBF construction can be extended to 3-QBF formulas Φ, such that Π^Φ has an answer set iff Φ is valid.

4.2 Rewriting disjunctive ASP into normal logic programs

In this subsection, we use our method to rewrite disjunctive (ground) ASP programs to non-ground *normal* ASP programs. In order words, we "shift" the additional complexity caused by disjunctive rule heads into non-ground rule bodies. As in the previous subsection, our encodings require predicates of fixed arity only, thus we map a Σ_P^2 problem to another Σ_P^2 problem.

Let $\Pi = (\mathscr{A}, \mathscr{R})$ be a ground, disjunctive logic program. We construct a non-ground, normal logic program $\hat{\Pi}$ of bounded arity, such that every answer set of $\hat{\Pi}$ witnesses the existence of a corresponding answer set of Π. First, $\hat{\Pi}$ contains the following facts:

- $\{atom(a) \mid a \in \mathscr{A}\}$,
- $\{rule(r) \mid r \in \mathscr{R}\}$,
- $\{leq(0,0), leq(0,1), leq(1,1)\}$,
- $\{or(0,0,0), or(0,1,1), or(1,0,1), or(1,1,1)\}$,
- $\{head(r,a) \mid r \in \mathscr{R}, a \in H(r)\}$,
- $\{pos(r,a) \mid r \in \mathscr{R}, a \in B^+(r)\}$, and
- $\{neg(r,a) \mid r \in \mathscr{R}, a \in B^-(r)\}$.

Predicates *atom*, *rule*, *head*, *pos*, and *neg* describe the disjunctive ground program to be evaluated. Predicate *leq* represents the less-or-equal relation; predicate *or* is used to encode logical disjunction. The role of the latter two predicates will be clarified below. Next, we give a fixed set of six rules that guesses an assignment on the atoms of Π, and then checks that this assignment is a classical model of Π:

- $assign(A,1) \leftarrow atom(A), \neg assign(A,0)$;
- $assign(A,0) \leftarrow atom(A), \neg assign(A,1)$;
- $sat(R) \leftarrow head(R,A), assign(A,1)$;
- $sat(R) \leftarrow pos(R,A), assign(A,0)$;
- $sat(R) \leftarrow neg(R,A), assign(A,1)$; and
- $\bot \leftarrow rule(R), \neg sat(R)$.

Finally, we need to check the subset-minimality of the guessed assignment M w.r.t. the reduct Π^M. This is done with one large rule r_{reduct}, encoding the guess of a subset $M' \subset M$, as well as the check if this subset M' satisfies all the rules in Π^M. In that case, the guessed assignment cannot be an answer set and the constraint r_{reduct} fires.

https://doi.org/10.1017/S1471068416000338 Published online by Cambridge University Press

Let the last rule r_{reduct} be of the following form, where $\mathbf{X}$ and $\mathbf{Y}$ contain a variable X_a and Y_a for each atom $a \in \mathscr{A}$, representing the atom's truth assignment in M and M', respectively:

$$r_{reduct} = \bot \leftarrow B_{subset}(\mathbf{X}, \mathbf{Y}) \cup B_{neq}(\mathbf{X}, \mathbf{Y}) \cup B_{model}(\mathbf{X}, \mathbf{Y}),$$

where the literals in B_{subset} encode the fact that M' (represented by $\mathbf{Y}$) is a subset of M (represented by $\mathbf{X}$); B_{neq} encodes the fact that this subset relation is proper; and B_{model} encodes the model check of M' against the reduct Π^M. Since ASP does not allow disjunction in rule bodies, we will use the $or(\cdot,\cdot,\cdot)$ atoms to simulate disjunction. B_{subset} first guesses a truth value for each variable in $\mathbf{Y}$ that is smaller or equal to the corresponding variable in $\mathbf{X}$ as follows:

$$B_{subset}(\mathbf{X}, \mathbf{Y}) \equiv \bigcup_{a \in \mathscr{A}} \{assign(a, X_a), leq(Y_a, X_a)\}.$$

Then, in B_{neq}, we use our *or* atoms as disjunction to check that at least one truth value in $\mathbf{Y}$ is different from the one in $\mathbf{X}$, which, in combination with B_{subset}, guarantees that the true atoms in $\mathbf{Y}$ are a proper subset of the ones in $\mathbf{X}$:

$$B_{neq}(\mathbf{X}, \mathbf{Y}) \equiv \{N_0 = 0\} \cup \bigcup_{a \in \mathscr{A}} \{or(N_i, X_a - Y_a, N_{i+1})\} \cup \{N_{|\mathscr{A}|} = 1\}.$$

Finally, we check that the truth assignment stored in $\mathbf{Y}$ actually represents a model of the reduct. To this end, B_{model} checks, by again making use of the *or* atoms to encode disjunction, that in each rule there exists an atom that makes the rule true. We have:

$$B_{model}(\mathbf{X}, \mathbf{Y}) \equiv \bigcup_{r \in \mathscr{R}} \Big(\{R^r_0 = 0\} \cup \bigcup_{a \in H(r)} \{or(R^r_i, Y_a, R^r_{i+1})\}$$

$$\cup \bigcup_{a \in B^+(r)} \{or(R^r_j, 1 - Y_a, R^r_{j+1})\} \cup \bigcup_{a \in B^-(r)} \{or(R^r_k, X_a, R^r_{k+1})\} \cup \{R^r_{|r|} = 1\} \Big),$$

where for a rule $r \in \mathscr{R}$, $|r| = |H(r)| + |B^+(r)| + |B^-(r)|$. In the above, variables N and R are used to "glue" together the disjunctions. For instance, in B_{neq} we have that N_i switches from 0 to 1 for N_{i+1} as soon as there is at least one atom assigned true in M but false in M', i.e. when $M' \subset M$. For the sake of readability, we do not explicitly specify how i, j and k are determined, but we assume that these index numbers increment by one for each element added to the respective set. Clearly, this could easily be formalized by assuming an order over all the atoms and rules.

Note that in the above construction, we have a fixed, non-disjunctive logic program to check for (classical) satisfiability. Then, according to the encoding paradigm proposed in Section 3, we use a rule with a large body to encode the coNP-check for stability (i.e. checking that a given classical model is minimal w.r.t. its reduct). This construction actually works for any ground, disjunctive answer set program, and thus yields a kind of "meta-solver" for ASP: we can use the rewriting above, and an ASP solver, to solve ASP itself. Another interesting observation is that, if we run a grounder on this program, we obtain a ground, normal logic program Π' (i.e. without disjunction) that solves our original ground logic program Π (which may

https://doi.org/10.1017/S1471068416000338 Published online by Cambridge University Press

contain disjunction). Thus, we have a rewriting algorithm that eliminates disjunction from ground logic programs, at the cost of an exponential blowup in general (since the grounding of r_{reduct} may be exponential).

The idea of creating such a meta-solver using reification techniques is not new; see e.g. (Gebser *et al.* 2011) and the references therein. However, most of these approaches "stay" within the same class of programs, i.e. the meta-program to solve a disjunctive logic program is itself a disjunctive logic program. Some of the meta-programming approaches in fact take the other direction than we do here. For example, in (Eiter and Polleres 2006) a meta-solver is used to equip a problem formulated as a non-disjunctive program with an additional coNP-test, yielding a disjunctive program as a result. Similar approaches are used to handle optimization statements (Gebser *et al.* 2011; Brewka *et al.* 2015). As we have emphasized before, our aim here is the opposite direction, that is, translating a ground disjunctive program into a (non-ground) program from an easier class[1]. For existing ASP solvers, solving performance drops significantly when (full) disjunction comes into play. It will be thus interesting to see whether such a rewriting improves performance.

4.3 Rewriting stable cautious inference into disjunctive logic programs

Finally, we provide an encoding for the Σ^3_P-complete problem of *stable cautious inference* (Eiter *et al.* 1997), defined as follows: Given a tuple $\langle \Pi, H, M \rangle$, where $\Pi = (\mathscr{A}, \mathscr{R})$ is a ground (disjunctive) logic program, and $H \subseteq \mathscr{A}$ and $M \subseteq \mathscr{A}$ are sets of (ground) atoms called *hypotheses* and *manifestations*, respectively: decide whether there exist a subset $E \subseteq H$, such that for all answer sets S of $\Pi \cup E$ it holds that $M \subseteq S$. Note that the original definition also requires that $AS(\Pi \cup E) \neq \emptyset$ However, inspecting the proof in (Eiter *et al.* 1997) shows that even without this condition, Σ^3_P-hardness is preserved. Thus, to simplify our construction, we will omit it.

The encoding of this program will partly be an adaptation of the encoding from Section 4.2; in particular, we will reuse the idea of r_{reduct}. The aim is to find an assignment to the hypotheses, such that for any assignment of the remaining atoms *not* containing all manifestations it is the case that such a joint assignment is *not* an answer set. We do so by employing saturation. This also forces us to encode the test for classical satisfaction of the rules in a different way than in Section 4.2, i.e. we have to saturate as soon as one rule is not satisfied.

To this end, let $\langle \Pi, H, M \rangle$ be an instance of the stable cautious inference problem. We create a program $\widehat{\langle \Pi, H, M \rangle}$ that has an answer set iff the tuple $\langle \Pi, H, M \rangle$ represents a valid instance of stable cautious inference.

We need only the following facts: for each atom a in Π, $atom(a)$; and for each $a \in H$, $hyp(a)$. The fixed set of rules contains the following:

[1] The *GnT* system follows an alternative idea to evaluate disjunctive logic programs (Janhunen *et al.* 2006): it facilitates a generate-and-test method based on disjunction-free programs, thus relying on two calls of an ASP-solver; in the worst-case an exponential input for the second program needs to be generated.

https://doi.org/10.1017/S1471068416000338 Published online by Cambridge University Press

- $assign(A,1) \vee assign(A,0) \leftarrow atom(A)$;
- $assign(A,1) \leftarrow sat, atom(A), \neg hyp(A)$;
- $assign(A,0) \leftarrow sat, atom(A), \neg hyp(A)$; and
- $\bot \leftarrow \neg sat$.

Then, we have several rules for saturation. First, we saturate if all manifestations are in the guess:

$$sat \leftarrow \bigcup_{a \in M} \{assign(a,1)\}.$$

Second, we saturate if the assignment is not a model of the program Π (and thus not of $\Pi \cup E$). To this end, for every rule $r \in \mathscr{R}$ we add

$$sat \leftarrow \bigcup_{a \in H(r)} \{assign(a,0)\} \cup \bigcup_{b \in B^+(r)} \{assign(b,1)\} \cup \bigcup_{c \in B^-(r)} \{assign(c,0)\}.$$

Finally, we again need the reduct check. Let the last rule again be r_{reduct}, defined in the same way as in the construction in Section 4.2:

$$r_{reduct} = sat \leftarrow B_{subset}(\mathbf{X},\mathbf{Y}) \cup B_{neq}(\mathbf{X},\mathbf{Y}) \cup B_{model}(\mathbf{X},\mathbf{Y}).$$

In r_{reduct}, the last two sets remain the same as in Section 4.2. The first set, B_{subset}, however, needs to reflect that the guessed hypotheses E are indeed facts in the program $\Pi \cup E$ (i.e. if they are true in a model candidate M represented by the variables $\mathbf{X}$, they must also be true in the reduct model candidate $M' \subset M$ represented by the variables $\mathbf{Y}$) . Thus, B_{subset} is changed as follows:

$$B_{subset} \equiv \bigcup_{a \in \mathscr{A} \setminus H} \{assign(a,X_a), leq(Y_a,X_a)\} \cup \bigcup_{a \in H} \{assign(a,X_a), X_a = Y_a\}$$

This completes the construction. Now, if a subset E of hypotheses can be found, such that saturation is not applied (observe that we saturate over all non-hypotheses atoms), we know that there is an assignment extending E to all atoms that (i) does not contain all manifestations, (ii) is a model of $\Pi \cup E$, and (iii) there is no subset of that assignment that is a model of the reduct. Hence, we found an answer of $\Pi \cup E$ that does not contain all manifestations. In that case, due to saturation, and the rule $\bot \leftarrow \neg sat$, this particular guess for E will not yield an answer set of the encoding. Thus, an answer set S of the rewritten program $\widehat{\langle \Pi, H, M \rangle}$ represents a valid selection of hypotheses E from H, such that in every answer set of $\Pi \cup E$ we find all atoms from M.

5 Experimental evaluation

In this section, we give a preliminary experimental evaluation of how our proposed rewritings perform when compared to current encodings and state-of-the-art problem-specific solvers. We have implemented the QBF rewriting algorithms of Section 4.1 and integrated these rewritings with the extended rule decomposition tool, *lpopt*[2], as described in Section 3.2.

[2] The updated *lpopt* tool is available at `http://www.dbai.tuwien.ac.at/proj/lpopt`

https://doi.org/10.1017/S1471068416000338 Published online by Cambridge University Press

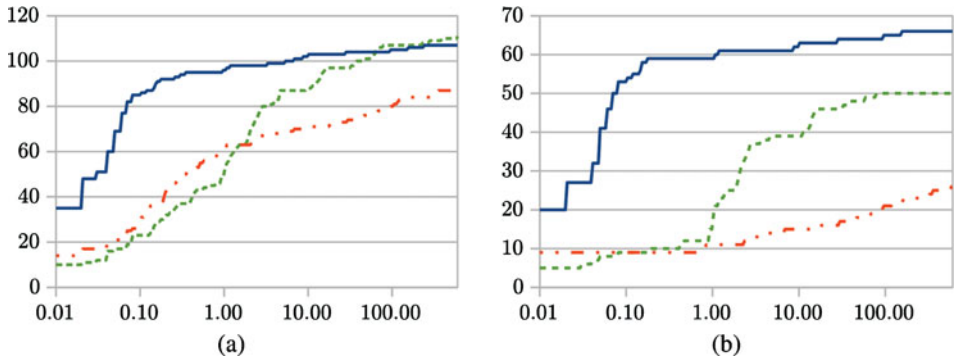


Fig. 1. Solved 2-QBF instances on the y-axis vs. time on the x-axis. Classic encodings in dotted-dashed red, encodings according to our paradigm in dashed green, and *DepQBF* in solid blue. (a) shows the overall benchmark results, (b) only the results for the "stmt" problem.

We used publicly available 2-QBF (∀∃) competition instances[3] and instances from the Eval-2012 data set[4] of the latest QBF competition for 3-QBF benchmarks. All benchmark inputs can be found online[5]. In the following, we compare the performance of (1) the classic encodings for 2-QBFs as described in Section 4.1; (2) the new encoding according to our paradigm as described in Section 4.1, where the long rules are decomposed with *lpopt*; and (3) a dedicated QBF solver. For (1) and (2), we first perform *DepQBF* preprocessing (Lonsing *et al.* 2015) and employ the well-known ASP solver *clingo*[6] on the preprocessed instances. For (3), we employ the state-of-the-art QBF solver *DepQBF*[7]. All tools were used in their most recent version. 3-QBF-benchmarks were run on a 16-core AMD Opteron machine with 2.1GHz, 224 GB of RAM, running Debian Linux. 2-QBF-benchmarks were run on an 8-core Intel Xeon machine with 2.33GHz and 48 GB of RAM. A global timeout was set at 600 seconds for 2-QBF, and 900 seconds for 3-QBF instances. All times measured are cumulative, that is, we sum up the times used for *DepQBF* preprocessing, rule decomposition, grounding, and solving. We measured the CPU time, thus, time wasted for I/O operations is not included in our measurements.

2-QBF Results. As can be seen in Figure 1(a), results for 2-QBF solving are very encouraging. In particular, for a runtime of about 2 seconds, we match the performance of the classical encoding, and for longer runtimes surpass it significantly overall. Where the classical encoding could only solve 88 instances in total, with our encoding according to the paradigm proposed in Section 3, *clingo* was able to solve 111 instances successfully within the given timeout window. Surprisingly, this was even better than *DepQBF*, a solver built specifically to evaluate QBF formulas. It was only able to solve 107 instances out of our benchmark set of 200 instances.

[3] See `http://www.qbflib.org/TS2010/2QBF.tar.gz`.
[4] Available at `http://qbf.satisfiability.org/gallery/eval2012r2.tgz`.
[5] `http://dbai.tuwien.ac.at/proj/lpopt/benchmarks.tgz`
[6] `http://potassco.sourceforge.net/`
[7] `http://lonsing.github.io/depqbf/`

https://doi.org/10.1017/S1471068416000338 Published online by Cambridge University Press

The largest single instance set within the benchmark set was the "stmt" problem. The benchmark results are shown in Figure 1(b). While here, *DepQBF* performed better than the ASP encodings, this still shows that our encoding can indeed outperform the classical encoding by a large margin, solving 51 instances compared to 26. Note that we do not claim to beat *DepQBF* in general in terms of performance when solving QBF problems. However, the benchmarks clearly show that our encoding paradigm can be by far superior to the classical, fixed ASP encoding.

3-QBF Results. In the case of solving third-level problems, results are sparse. For the 151 3-QBF instances, *DepQBF* was able to solve 47, while our approach was able to solve 18. However, out of these 18 instances, there were ten that our 3-QBF encoding was able solve, while *DepQBF* was unable to solve them within the time limit. This again indicates the viability of our encoding paradigm, using the power of readily available, state-of-the-art ASP solvers in order to solve problems on the third level of the polynomial hierarchy.

Let us conclude with a remark on the treewidth of the large rules in our encodings. Generally, current grounders tend to time out when a rule is too large. Thus, when encoding problems with large rules according to our paradigm, performance is best when the resulting rule has low treewidth, since then our rule decomposition tool can split the rule up into very small rules that are easy to process for the grounder. When the treewidth of the rule is high, however, the decomposed rules will still be very large, in which case the grounding bottleneck may still be a dominating factor. Rule treewidth in our instances ranged from 3 to over 300, and grounding sizes (and timeouts) varied accordingly, leading to expected grounder timeouts on instances with large, high-treewidth rules. However, we note that this is a property of the original input instance, and not of our rule decomposition approach which, in the worst case, will not change the rules in the program at all, and can thus not make things worse. Finally, since *lpopt* makes use of heuristics to compute tree decompositions (see (Dermaku *et al.* 2008) for details of the heuristics used), some variability in decomposition quality is expected. This can cause variations in grounding time and size for different decompositions of the same rule. However, these generally stayed within the range of less than 10%.

6 Conclusion

In this paper, we laid out a novel approach to encode problems into ASP. These rewritings make heavy use of large rules, which can be used to encode NP-checks. For disjunctive ASP, if the predicate arity is bounded, it becomes possible to solve problems of up to Σ_P^3-hardness with this approach, since the consistency problem for disjunctive ASP with bounded predicate arity is itself Σ_P^3-complete (Eiter *et al.* 2007). Because existing grounders are slow to ground large rules, we then present an extended rule decomposition algorithm to make our encodings solvable in practice. Finally, we provided several examples on how our approach can be used to solve problems on the second and third level of the polynomial hierarchy, and show that these encodings actually perform well in practice.

https://doi.org/10.1017/S1471068416000338 Published online by Cambridge University Press

Acknowledgments. This work was funded by the Austrian Science Fund (FWF): Y698, P25607.

References

Alviano, M., Dodaro, C., Faber, W., Leone, N. and Ricca, F. 2013. WASP: A native ASP solver based on constraint learning. In *Proc. LPNMR*, 54–66.

Alviano, M., Faber, W., Leone, N., Perri, S., Pfeifer, G. and Terracina, G. 2010. The disjunctive datalog system DLV. In *Datalog Reloaded. Revised Selected Papers*, 282–301.

Arnborg, S., Corneil, D. G. and Proskurowski, A. 1987. Complexity of finding embeddings in a k-tree. *SIAM J. Algeb. Discr. Meth. 8,* 2, 277–284.

ASP-Core-2 2015. ASP Core 2 Standard, v2.03c. `https://www.mat.unical.it/aspcomp2013/ASPStandardization`. Accessed: 2016-04-28.

Bichler, M. 2015. Optimizing non-ground answer set programs via rule decomposition. BSc Thesis, TU Wien. `http://dbai.tuwien.ac.at/proj/lpopt/thesis.pdf`.

Bodlaender, H. L. 1996. A linear-time algorithm for finding tree-decompositions of small treewidth. *SIAM J. Comput. 25,* 6, 1305–1317.

Bonatti, P. A., Pontelli, E. and Son, T. C. 2008. Credulous resolution for answer set programming. In *Proc. AAAI*, 418–423.

Brewka, G., Delgrande, J. P., Romero, J. and Schaub, T. 2015. asprin: Customizing answer set preferences without a headache. In *Proc. AAAI*, 1467–1474.

Brewka, G., Eiter, T. and Truszczynski, M. 2011. Answer set programming at a glance. *Commun. ACM 54,* 12, 92–103.

Chandra, A. K. and Merlin, P. M. 1977. Optimal implementation of conjunctive queries in relational data bases. In *Proc. STOC*, 77–90.

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

de Cat, B., Denecker, M. and Stuckey, P. J. 2012. Lazy model expansion by incremental grounding. In *Proc. ICLP*, 201–211.

Dermaku, A., Ganzow, T., Gottlob, G., McMahan, B. J., Musliu, N. and Samer, M. 2008. Heuristic methods for hypertree decomposition. In *Proc. MICAI*, 1–11.

Eiter, T., Faber, W., Fink, M. and Woltran, S. 2007. Complexity results for answer set programming with bounded predicate arities and implications. *Ann. Math. Artif. Intell. 51,* 2-4, 123–165.

Eiter, T., Faber, W. and Mushthofa, M. 2010. Space efficient evaluation of ASP programs with bounded predicate arities. In *Proc. AAAI*, 303–308.

Eiter, T. and Gottlob, G. 1995. On the computational cost of disjunctive logic programming: Propositional case. *Ann. Math. Artif. Intell. 15,* 3-4, 289–323.

Eiter, T., Gottlob, G. and Leone, N. 1997. Abduction from logic programs: Semantics and complexity. *Theor. Comput. Sci. 189,* 1-2, 129–177.

Eiter, T. and Polleres, A. 2006. Towards automated integration of guess and check programs in answer set programming: a meta-interpreter and applications. *Theory and Practice of Logic Programming 6,* 1-2, 23–60.

Elkabani, I., Pontelli, E. and Son, T. C. 2005. Smodels^{A} - A system for computing answer sets of logic programs with aggregates. In *Proc. LPNMR*, 427–431.

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

https://doi.org/10.1017/S1471068416000338 Published online by Cambridge University Press

Gebser, M., Kaminski, R. and Schaub, T. 2011. Complex optimization in answer set programming. *Theory and Practice of Logic Programming 11,* 4-5, 821–839.

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

Gelfond, M. and Lifschitz, V. 1988. The stable model semantics for logic programming. In *Proc. ICLP/SLP*, 1070–1080.

Gottlob, G., Miklós, Z. and Schwentick, T. 2009. Generalized hypertree decompositions: NP-hardness and tractable variants. *J. ACM 56,* 6, article no. 30.

Gottlob, G. and Papadimitriou, C. H. 2003. On the complexity of single-rule datalog queries. *Inf. Comput. 183,* 1, 104–122.

Gottlob, G. and Schwentick, T. 2012. Rewriting ontological queries into small nonrecursive datalog programs. In *Proc. KR*, 254–263.

Janhunen, T., Niemelä, I., Seipel, D., Simons, P. and You, J. 2006. Unfolding partiality and disjunctions in stable model semantics. *ACM Trans. Comput. Log. 7,* 1, 1–37.

Lefèvre, C., Béatrix, C., Stéphan, I., and Garcia, L. 2015. ASPeRiX, a first order forward chaining approach for answer set computing. *CoRR abs/1503.07717.*

Lonsing, F., Bacchus, F., Biere, A., Egly, U., and Seidl, M. 2015. Enhancing search-based QBF solving by dynamic blocked clause elimination. In *Proc. LPAR*, 418–433.

Marek, V. W. and Truszczyński, M. 1999. Stable Models – an Alternative Logic Programming Paradigm. In *The Logic Programming Paradigm – A 25-Year Perspective.* Springer, 375–398.

Morak, M. and Woltran, S. 2012. Preprocessing of complex non-ground rules in answer set programming. In *Proc. ICLP*, 247–258.

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

https://doi.org/10.1017/S1471068416000338 Published online by Cambridge University Press