Simulating Sets in Answer Set Programming

Sarah Alice Gaggl[1] , **Philipp Hanisch**[2] and **Markus Krötzsch**[2]
[1]Logic Programming and Argumentation Group, TU Dresden, Germany
[2]Knowledge-Based Systems Group, TU Dresden, Germany
{sarah.gaggl, philipp.hanisch1, markus.kroetzsch}@tu-dresden.de

Abstract

We study the extension of non-monotonic disjunctive logic programs with terms that represent sets of constants, called DLP(S), under the stable model semantics. This strictly increases expressive power, but keeps reasoning decidable, though cautious entailment is $\text{coNExpTime}^{\text{NP}}$-complete, even for data complexity. We present two new reasoning methods for DLP(S): a semantics-preserving translation of DLP(S) to logic programming with function symbols, which can take advantage of lazy grounding techniques, and a ground-and-solve approach that uses non-monotonic existential rules in the grounding stage. Our evaluation considers problems of ontological reasoning that are not in scope for traditional ASP (unless $\text{ExpTime} = \Pi_2^{\text{P}}$), and we find that our new existential-rule grounding performs well in comparison with native implementations of set terms in ASP.

1 Introduction

The success of answer set programming (ASP) is in no small part due to its declarative approach to computation, which allows users to encode complex problems in a clean, direct manner [Brewka *et al.*, 2011; Lifschitz, 2019]. Plain ASP can express problems up to the second level of the polynomial hierarchy [Dantsin *et al.*, 2001], and highly optimised ASP solvers often lead to efficient solutions in such cases. However, not all problems can be expressed in this way, and ASP has therefore been extended with more complex terms that support function symbols [Bonatti, 2004], list, and sets [Calimeri *et al.*, 2009].

The ability to encode sets (of constants) in terms is of particular interest here, since it preserves – in contrast to function symbols and lists – the decidability of reasoning. Moreover, sets and related predicates like $\in$ and $\subseteq$ support a very declarative modelling style. For example, the rules in Figure 1 define maximal strongly connected components (scc) on a graph described by edges $e(X, Y)$ and vertices $v(X)$. The rules for c show how sets can be constructed iteratively using suitable functions, while the last two rules provide an elegant description of maximal sets. Such elegance presumably was the main motivation for implementations of sets in ASP solvers [Calimeri *et al.*, 2009], but sets have further computational advantages in boosting expressive power, as noted already for the case of Datalog [Ortiz *et al.*, 2010; Carral *et al.*, 2019b].

$$
\begin{aligned}
e^{+}(X,Y) &\leftarrow e(X,Y)\\
e^{+}(X,Y) &\leftarrow e^{+}(X,Z) \wedge e(Z,Y)\\
c(\{X\}) &\leftarrow v(X)\\
c(S \cup \{Y\}) &\leftarrow c(S) \wedge X \in S \wedge e^{+}(X,Y) \wedge e^{+}(Y,X)\\
\mathit{subc}(S_1) &\leftarrow c(S_1) \wedge c(S_2) \wedge S_1 \subsetneq S_2 \wedge \textbf{not}\, S_2 \subseteq S_1\\
\mathit{scc}(S) &\leftarrow c(S) \wedge \textbf{not}\, \mathit{subc}(S)
\end{aligned}
$$

Figure 1: Strongly connected components in ASP with sets

Surprisingly, the extension of ASP with sets remains poorly understood, regarding both its theoretical properties and its practical potential. Most notably, Calimeri *et al.* [2008; 2009] discuss set terms in combination with functions and lists, and provide a first implementation in *DLV-complex*. However, to our knowledge, even the complexity of reasoning in ASP with sets has not been established yet, and its utility for solving computationally hard problems has never been discussed or evaluated. One may also wonder whether formalisms that are even more powerful, such as rules with function symbols or value invention, could be exploited to implement sets, but research on feasible implementation methods is similarly scarce. Significant potential for using ASP and related tools therefore remains unexplored.

To address this, we take a closer look at the extension of ASP with set terms, investigate possible implementation methods in theory, and evaluate their practical feasibility in solving problems that are beyond the expressive power of plain ASP. Our main contributions are:

- We define DLP(S), the extension of disjunctive logic programs with set terms, and establish relevant reasoning complexities under the stable model semantics.
- We show how DLP(S) reasoning can be reduced to reasoning in disjunctive logic programs with function symbols while still ensuring finite stable models.
- We show that existing lazy ASP solvers can handle these programs, whereas traditional ASP engines will fail.
- We develop another alternative grounding approach

based on a technique for modelling sets in existential rules [Carral *et al.*, 2019b].

- We evaluate the ability of our new methods and the existing set implementation DLV-complex to solve complex real-world problems from ontology engineering.

Full proofs, datasets, and source code for our prototype implementations can be accessed through the online repository https://github.com/knowsys/eval-2022-IJCAI-asp-with-sets.

2 ASP with Sets

We now introduce the extension of non-monotonic disjunctive logic programs with finite sets (DLP(S)). This will allow us to write rules as in Figure 1.

Syntax. Formally, we consider two *sorts*: a sort of *objects* $\mathbf{obj}$ and a sort of *sets* $\mathbf{set}$. A signature of DLP(S) is based on countably infinite sets of *object constants* $\mathbf{C_{obj}}$ (typically a, b, c), *object variables* $\mathbf{V_{obj}}$ (typically X, Y, Z), *set variables* $\mathbf{V_{set}}$ (typically S, U, V), and *predicate names* $\mathbf{P}$ including *special predicates* $\{\in, \subseteq\} \subseteq \mathbf{P}$. The *signature* of a predicate symbol p is a tuple $\mathit{sig}(p) = \langle \mathbf{S_1}, \ldots, \mathbf{S_n} \rangle$ of sorts, and n is called its *arity*. We have $\mathit{sig}(\in) = \langle \mathbf{obj}, \mathbf{set} \rangle$ and $\mathit{sig}(\subseteq) = \langle \mathbf{set}, \mathbf{set} \rangle$. An *object term* is any object constant or object variable. A *set term* is any set variable, the special constant $\emptyset$, an expression $\{t\}$ where t is an object term, or, recursively, an expression $(s_1 \cup s_2)$ where s_1, s_2 are set terms. We use $\{t_1, \ldots, t_n\}$ as abbreviation for $(\{t_1\} \cup (\{t_2\} \cup \ldots \cup \{t_n\} \ldots))$ and omit parentheses for $\cup$. An term or formula is *ground* if it contains no variables.

An *atom* is an expression $p(t_1, \ldots, t_n)$ where $p \in \mathbf{P}$ with $\mathit{sig}(p) = \langle \mathbf{S_1}, \ldots, \mathbf{S_n} \rangle$ and t_i is a term of sort $\mathbf{S_i}$. We write $\in(t, s)$ as $t \in s$, and $\subseteq(s_1, s_2)$ as $s_1 \subseteq s_2$. A *literal* is an atom α or a negated atom $\mathbf{not}\, \alpha$. We sometimes treat conjunctions or disjunctions of literals as sets of literals. For a formula F, $\mathrm{preds}(F)$ denotes the set of all predicates in F.

Definition 1. A *DLP(S) rule* is an expression r of the form $H \leftarrow B^+ \wedge B^-$, where the *head* H is a disjunction of atoms, the *positive body* B^+ is a conjunction of atoms, and the *negative body* B^- is a conjunction of negated atoms, such that:

1. every object variable in r occurs in B^+,
2. every set variable S in r occurs in B^+ as an argument $t_i = S$ $(1 \leq i \leq n)$ of an atom $p(t_1, \ldots, t_n) \in B^+$ with non-special predicate $p \in \mathbf{P} \setminus \{\in, \subseteq\}$, and
3. the special predicates $\in$ and $\subseteq$ do not occur in H.

A *fact* is a disjunction-free rule with empty body, i.e., a ground atom. A *DLP(S) program* P is a set of DLP(S) rules. A rule, fact, or program is DLP if it contains only object terms. We also consider the extension of DLP with arbitrary (object) function symbols, which we will denote DLP^f.

Some restrictions in Definition 1 could be relaxed without fundamentally changing the properties of DLP(S). Our selection is also motivated by practical concerns, e.g., since a rule $q(S) \leftarrow p(S \cup T)$ (which violates condition 2) would produce exponentially many derivations in the size of any set R for which $p(R)$ holds.

A substitution σ is a sort-preserving partial mapping from variables to terms. $F\sigma$ denotes the result of applying σ to all variables in the formula or term F for which it is defined.

Semantics. We define a stable model semantics for DLP(S) by interpreting set terms as finite sets over the (object) domain. Given a program P, let $\mathbf{obj}(P)$ be the set of all object constants in P (including those used in set terms), and let $\mathbf{set}(P)$ be a set of terms of the form $\{t_1, \ldots, t_n\}$ that bijectively correspond to the powerset of $\mathbf{obj}(P)$. The *grounding* $\mathrm{ground}(P)$ of P consists of all rules that can be obtained from a rule r of P by (1) uniformly replacing object and set variables in r by terms from $\mathbf{obj}(P)$ and $\mathbf{set}(P)$, respectively; and (2) replacing each of the resulting set terms s by the corresponding term from $\mathbf{set}(P)$ that represents the same set under the usual interpretation of $\emptyset$, $\{\cdot\}$, and $\cup$.

An *interpretation* $\mathcal{I}$ for P is a set of ground facts that only uses terms from $\mathbf{obj}(P)$ and $\mathbf{set}(P)$, and that contains exactly those facts $c \in t$ (resp. $s \subseteq t$) for which c occurs in the set term t (resp. for which all constants in s occur in t). $\mathcal{I}$ *satisfies* (or *is a model of*) a ground atom α if $\alpha \in \mathcal{I}$. $\mathcal{I}$ satisfies a ground conjunction B (disjunction H) if $B \subseteq \mathcal{I}$ $(H \cap \mathcal{I} \neq \emptyset)$, and a positive ground rule $H \leftarrow B$ if it satisfies H or does not satisfy B. The *reduct* $P^{\mathcal{I}}$ of P with respect to $\mathcal{I}$ is obtained from $\mathrm{ground}(P)$ by (1) deleting every negated atom $\mathbf{not}\, \alpha$ with $\alpha \notin \mathcal{I}$ and (2) deleting every rule with a negated atom $\mathbf{not}\, \alpha$ with $\alpha \in \mathcal{I}$. $\mathcal{I}$ is a *stable model* of P if it is a subset-minimal model of $P^{\mathcal{I}}$. A fact α is (cautiously) entailed by P if every stable model of P contains the fact α', obtained by replacing set terms in α with their representative in $\mathbf{set}(P)$.

Since there are exponentially many sets over a given object domain, groundings, reducts, and stable models can also be exponential for DLP(S), even in the size of the data:

Theorem 1. *Deciding whether P entails α is* $\textsc{coNExpTime}^{\mathrm{NP}}$*-complete. If P does not contain $\vee$, the problem is* $\textsc{coNExpTime}$*-complete. In either case, hardness holds even if only facts are allowed to vary while all other rules are fixed (data complexity).*

Proof sketch. The data complexities of the considered problems are Π_2^{P}-complete respectively $\textsc{coNP}$-complete for DLP [Dantsin *et al.*, 2001]. Hardness can be shown, e.g., by reducing from the word problem of polynomial-time alternating Turing machines with one quantifier alternation. The claims for DLP(S) follow by simulating an exponentially time-bounded alternating Turing machine instead.

The encoding is standard once an exponentially long chain (of time points or tape cells) is inferred. Given input facts $\mathit{succ}(1, 2), \ldots, \mathit{succ}(\ell - 1, \ell)$, a chain of length 2^ℓ is characterised by a predicate c, defined as follows:

$$\mathit{succ}^+(X, Y) \leftarrow \mathit{succ}(X, Y) \tag{1}$$

$$\mathit{succ}^+(X, Z) \leftarrow \mathit{succ}^+(X, Y) \wedge \mathit{succ}(Y, Z) \tag{2}$$

$$n(\emptyset, \{1\}, 1, \emptyset) \leftarrow \tag{3}$$

$$n(U, \{X\} \cup \grave{V}, X, \grave{V}) \leftarrow n(\_, U, X, \grave{U}) \wedge n(\grave{U}, \grave{V}, \grave{X}, \_) \wedge \mathit{succ}^+(\grave{X}, X) \tag{4}$$

$$n(U, \{Y\}, Y, \emptyset) \leftarrow n(\_, U, X, \grave{U}) \wedge n(\grave{U}, \_, X, \_) \wedge \mathit{succ}(X, Y) \tag{5}$$

$$c(S,T) \leftarrow n(S,T,\_,\_) \tag{6}$$

Here we use _ for anonymous variables. The encoding is adapted from Carral *et al.* [2019b]. Predicates $n(S,T,X,V)$ express that (1) T is the successor of S if both sets are taken as binary numbers with elements from $\{1,\ldots,\ell\}$ encoding active digits, (2) X is the largest number (most significant bit) in T, and (3) $T \setminus \{X\} = V$. With this in mind, (4) and (5) increment numbers with the highest bit staying the same or being increased by one, respectively.

Membership is also derived from known results by reducing the DLP(S) problems to propositional logic programming problems via grounding, which incurs an exponential blow-up in the presence of set terms. □

3 Reducing DLP(S) to DLP

Answer Set Programming over DLP(S) could be implemented by similar techniques as normal ASP, taking the semantics of sets into account when applying rules (operational extension APIs like clingo's *@-terms* or DLV's *external atoms* could be used). However, this is rarely done so far, and we therefore explore alternative approaches. A first idea is to reduce DLP(S) to DLP with arbitrary function symbols (DLP^f), which is known to be computationally very powerful [Bonatti, 2004; Eiter and Simkus, 2010]. In this section, we show how this can be done and establish the correctness of the translation. Its computational properties and possible implementation in current solvers are discussed later.

We encode sets using a constant $c_\emptyset$ (the empty set) and a function $f_\cup$, such that, e.g., $f_\cup(a, f_\cup(b, c_\emptyset))$ represents the set $\{a,b\}$. However, we cannot generally represent $\{c\} \cup s$ by $f_\cup(c,s)$, since this would lead to redundant representations like $f_\cup(a, f_\cup(a, c_\emptyset))$. Instead, we use facts of the form $su(c,s,t)$ – where su stands for *singleton union* – to express $\{c\} \cup s = t$, where t might be different from $f_\cup(c,s)$. In particular, $su(c,s,s)$ means $c \in s$, which we abbreviate by $in(c,s)$. Finally, to ensure that only necessary set encodings are derived, we use facts $get\_su(c,s)$ to express that a representation of $\{c\} \cup s$ is needed. The following DLP^f rules implement these ideas:

$$su(X,S,f_\cup(X,S)) \leftarrow get\_su(X,S) \wedge \textbf{not}\, in(X,S) \tag{7}$$
$$su(X,U,U) \leftarrow su(X,S,U) \tag{8}$$
$$su(Y,U,U) \leftarrow su(X,S,U) \wedge in(Y,S) \tag{9}$$
$$in(X,S) \leftarrow su(X,S,S) \tag{10}$$

We can exercise this machinery to define rules that compute arbitrary unions using analogous predicates u and get_u:

$$u(S,c_\emptyset,S) \leftarrow get\_u(S,c_\emptyset) \tag{11}$$
$$u(S,f_\cup(X,T),U) \leftarrow get\_u(S,f_\cup(X,T)) \wedge su(X,S,\acute{S}) \wedge u(\acute{S},T,U) \tag{12}$$
$$get\_su(X,S) \leftarrow get\_u(S,f_\cup(X,T)) \tag{13}$$
$$get\_u(\acute{S},T) \leftarrow get\_u(S,f_\cup(X,T)) \wedge su(X,S,\acute{S}) \tag{14}$$

Here, (11) is the base and (12) the recursive case, and (13) and (14) ensure the computation of necessary auxiliary facts.

We are now ready to transform a single DLP(S) rule $r = H \leftarrow B$ to a set of DLP^f rules $\mathrm{dlp^f}(r)$. For every DLP(S) predicate p of arity n, let $\hat{p}$ be a unique fresh predicate of arity n and signature $sig(p) = \langle \mathbf{obj}, \ldots, \mathbf{obj} \rangle$, and for every set term s, let V_s be a fresh object variable. We construct H' and B' from H and B, respectively, by replacing each atom $t \in s$ by $in(t,s)$, each atom $s \subseteq u$ by $sub(s,u)$, every predicate p by $\hat{p}$, and every set term s by V_s. Moreover, let B^+ be the conjunction of positive literals in B'. Now let $s_1, \ldots, s_k$ be a list of all terms and subterms in r of the form $\{t\}$ or $s \cup u$, ordered such that subterms occur before their superterms. Then $\mathrm{dlp^f}(r)$ consists of the following rules:

$$H' \leftarrow B' \wedge \textstyle\bigwedge_{i=1}^{k} \beta(s_i) \tag{15}$$
$$\alpha(s_{j+1}) \leftarrow B^+ \wedge \textstyle\bigwedge_{i=1}^{j} \beta(s_i) \quad j \in \{0,\ldots,k-1\} \tag{16}$$

where we define, for $v = \{t\}$, $\alpha(v) = get\_su(t, c_\emptyset)$ and $\beta(v) = su(t, c_\emptyset, V_v)$; and, for $v = s \cup u$, $\alpha(v) = get\_u(V_s, V_u)$ and $\beta(v) = u(V_s, V_u, V_v)$.

Example 1. Consider the DLP(S) rule $r : p(S) \leftarrow q(S,x) \wedge r(S \cup \{x\}) \wedge \textbf{not}\, x \in S$. The required list of set terms is $s_1 = \{x\}, s_2 = S \cup \{x\}$. We have $H' = \hat{p}(V_S)$, $B^+ = \hat{q}(V_S,x) \wedge \hat{r}(V_{S\cup\{x\}})$, and $B' = B^+ \wedge \textbf{not}\, in(x, V_S)$. Now $\mathrm{dlp^f}(r)$ consists of the rules:

$$H' \leftarrow B' \wedge su(x, c_\emptyset, V_{\{x\}}) \wedge u(V_S, V_{\{x\}}, V_{S\cup\{x\}})$$
$$get\_su(x, c_\emptyset) \leftarrow B^+$$
$$get\_u(V_S, V_{\{x\}}) \leftarrow B^+ \wedge su(x, c_\emptyset, V_{\{x\}})$$

Note how B^+ is used to ensure that the auxiliary rules that define only some of the variables V_s are safe in the sense of Definition 1 (1).

As Example 1 suggests, we could sometimes use get_su and su instead of get_u and u. This optimisation can be useful in practice since it may make rules (11)–(12) obsolete, but we omit it from our formal description for simplicity.

The transformation needs one more ingredient, since our set representations are not unique. For example, $f_\cup(a, f_\cup(b, c_\emptyset))$ and $f_\cup(b, f_\cup(a, c_\emptyset))$ both represent $\{a,b\}$. We therefore explicitly compute equality of sets as follows:

$$sub(c_\emptyset, c_\emptyset) \leftarrow \tag{17}$$
$$sub(c_\emptyset, S) \leftarrow in(X,S) \tag{18}$$
$$sub(T,U) \leftarrow su(X,S,T) \wedge sub(S,U) \wedge in(X,U) \tag{19}$$
$$eq(S,T) \leftarrow sub(S,T) \wedge sub(T,S) \tag{20}$$

Definition 2. Given a DLP(S) program P, the DLP^f program $\mathrm{dlp^f}(P)$ consists of the following rules:

- the rules (7)–(12) and (17)–(20);
- for every predicate $\hat{p}$ in P of arity n, the rules

  $$\hat{p}(X_1, \cdots, X_n)[X_i/Y] \leftarrow \hat{p}(X_1, \cdots, X_n) \wedge eq(X_i, Y)$$

  where $[X_i/Y]$ is the substitution replacing X_i by Y;
- for every rule $r \in P$, the rules $\mathrm{dlp^f}(r)$.

The next result is a pre-condition for the practical utility of our translation. It can be shown directly but also follows from Theorem 5 below.

Theorem 2. *All stable models of* $\mathrm{dlp^f}(P)$ *are finite.*

We can convert ground terms s that contain $c_\emptyset$ or $f_\cup$ to canonical set terms $[s] \in \mathbf{set}(P)$ in the obvious way: $[c_\emptyset] = \emptyset$ and $[f_\cup(t, s)]$ is the representation of $\{t\} \cup [s]$ in $\mathbf{set}(P)$. Moreover, let $[t] = t$ for other terms (not representing sets). Then, for every stable model $\mathcal{J}$ of $\mathrm{dlp}^{\mathrm{f}}(P)$, we find an interpretation $[\mathcal{J}] = \{p([t_1], \cdots, [t_n]) \mid \hat{p}(t_1, \cdots, t_n) \in \mathcal{J}\}$.

Theorem 3. *The set of stable models of a DLP(S) program P is exactly the set $\{[\mathcal{J}] \mid \mathcal{J}$ is a stable model of $\mathrm{dlp}^{\mathrm{f}}(P)\}$.*

4 Lazy Grounding

A popular approach towards reasoning in ASP is "ground & solve", where one first computes a set of ground instances of rules that are then turned into a propositional logic problem to which an ASP solver can be applied [Gebser *et al.*, 2018; Faber, 2020]. To ensure that the grounding stage produces enough rule instances, it is common to disregard negative body literals when applying rules during grounding. Optimisations can further reduce the number of rule applications, e.g., using predicate dependencies to compute a *stratification*. A classical algorithm that combines several such ideas was proposed by Calimeri *et al.* [2008], who defined $\mathcal{FG}$ as the class of all DLP^f programs on which their grounding is finite. However, although our programs $\mathrm{dlp}^{\mathrm{f}}(P)$ have finite stable models (Theorem 2), such classical grounding approaches will usually not work for them:

Proposition 4. *If P contains a fact $p(\emptyset)$ and a rule $p(S \cup \{a\}) \leftarrow p(S)$, then $\mathrm{dlp}^{\mathrm{f}}(P) \notin \mathcal{FG}$.*

Indeed, it can be verified in practice that grounders such as *gringo* and *iDLV* do not terminate on $\mathrm{dlp}^{\mathrm{f}}(P)$ for programs P as in Proposition 4. A challenge that grounders are facing in this case is that negation in the rules (7)–(10) of $\mathrm{dlp}^{\mathrm{f}}(P)$ is not stratified. To avoid groundings that are too small, most classical grounders will therefore ignore the negated literal in rule (7), which leads to a program that has no finite model.

Fortunately, this problem can be avoided by *lazy grounding* approaches, which interleave grounding and solving to produce rule instances only when relevant to the search for a stable model. Notable systems of this type include *Alpha* [Weinzierl, 2017; Taupe *et al.*, 2019] and *ASPeRiX* [Lefèvre *et al.*, 2017]. We analyse the former to show that such approaches are applicable to our task:

Theorem 5. *The algorithm of Alpha terminates on every program of the form $\mathrm{dlp}^{\mathrm{f}}(P)$.*

Alpha therefore produces a complete set of (necessarily finite) stable models of $\mathrm{dlp}^{\mathrm{f}}(P)$, which correspond to the stable models of P by Theorem 3. The essence of our proof of this claim is the fact that Alpha eagerly applies a form of *unit propagation* where it exhaustively applies rules (8)–(10) before considering further *choice points* obtained by grounding rule (7). Since the propagation will derive facts for *in*, rule (7) is only potentially applicable to values of X that are not in the set represented by S, and sets can only be enlarged a finite number of times before no such elements are left.

5 Grounding with Existential Rules

Instead of relying on lazy grounding, an alternative approach towards reasoning in DLP(S) is to develop a set-aware grounder whose output can be combined with existing solvers. Since classical grounders are mostly Datalog engines (with some support for stratified negation), we essentially need an engine for Datalog(S), the extension of Datalog with sets. To obtain this, we follow an approach by Carral *et al.* [2019b] who used a reasoning algorithm from databases (the *standard chase*) to simulate Datalog(S) reasoning, and we argue that it can safely be combined with stratified negation, which is not immediate in this context.

We first explain grounding based on Datalog(S) with stratified negation. Given a DLP(S) program P, a *stratification* $s : P \to \mathbb{N}$ maps rules to natural numbers such that, for all pairs of rules $r_1, r_2 \in P$ of form $r_i : H_i \leftarrow B_i^+ \wedge B_i^- \in P$, and all $p \in \mathrm{preds}(H_2)$: (1) if $p \in \mathrm{preds}(B_1^+)$, then $s(r_1) \geq s(r_2)$, and (2) if $p \in \mathrm{preds}(B_1^-)$, then $s(r_1) > s(r_2)$. This induces a partitioning $P_1, \ldots, P_n$ of P such that $P_i = \{r \in P \mid s(r) = i\}$. A program P is *stratified* if it has a stratification.

Now for any program P, let $P^! \subseteq P$ be the largest disjunction-free, stratified subset of P such that predicates in heads of $P \setminus P^!$ do not occur in $P^!$.[1] A predicate is *certain* if it occurs only in $P^!$. The Datalog(S) program $\mathrm{grnd}(P)$ contains all facts in P and, for each non-fact rule $H \leftarrow B^+ \wedge B^- \in P$, the rules

$$rule_r(\boldsymbol{X}) \leftarrow B^+ \wedge B^! \tag{21}$$

$$\textstyle\bigwedge_{\alpha \in H} \alpha \leftarrow rule_r(\boldsymbol{X}) \tag{22}$$

where $rule_r$ is a dedicated fresh predicate for r, $\boldsymbol{X}$ is a list of all variables in r, and $B^! = \bigwedge\{\mathbf{not}\,\alpha \in B^- \mid$ the predicate of α is certain$\}$. For a ground atom of the form $rule_r(\boldsymbol{c})$, the ground rule $r[\boldsymbol{c}]$ is obtained by applying the substitution $\boldsymbol{X} \mapsto \boldsymbol{c}$ to r. We get a simple but correct grounding:

Proposition 6. *For any DLP(S) program P, $\mathrm{grnd}(P)$ is a stratified Datalog(S) program. If $\mathcal{I}$ is the (unique) stable model of $\mathrm{grnd}(P)$, then the stable models of $\{r[\boldsymbol{c}] \mid rule_r(\boldsymbol{c}) \in \mathcal{I}\}$ are exactly the stable models of P.*

To compute the stable model of $\mathrm{grnd}(P)$, we simulate reasoning in stratified Datalog(S) by generalising an approach for *existential rules* (also know as *tuple-generating dependencies*), which we extend by negation (which will be stratified in the same sense as defined above). Such rules have the form $\exists \boldsymbol{Y}.H \leftarrow B^+ \wedge B^-$, where H and B^+ are conjunctions of atoms, B^- is a conjunction of negated atoms, $\boldsymbol{Y}$ is a list of existentially quantified variables, and all other (implicitly universally quantified) variables also occur in B^+ (*safety*). Existential quantifiers may lead to new domain elements, represented by *named nulls*, which play the role of anonymous constants. We can *apply* an existential rule $\exists \boldsymbol{Y}.H \leftarrow B^+ \wedge B^-$ to a set $\mathcal{I}$ of facts (using constants and possibly nulls) if (1) there is a substitution θ with $B^+\theta \subseteq \mathcal{I}$ and $B^- \cap \mathcal{I} = \emptyset$, and (2) the rule is not already satisfied under θ, i.e., $\mathcal{I} \not\models \exists \boldsymbol{Y}.(H\theta)$. If this holds, we apply the rule by extending $\mathcal{I}$ with $H\theta'$, where θ' is an extension of θ that maps each $Y \in \boldsymbol{Y}$ to a fresh null.

For existential rules without negation, a *(standard) chase* is a possibly infinite set $\mathcal{I}$ obtained by exhaustive, fair application of rules. For a set of existential rules with negation

[1]This can be viewed as a kind of split program in the sense of Lifschitz and Turner [1994].

that has a stratification $P_1, \ldots, P_n$, the *stratified chase* $\mathcal{I}$ is $\bigcup_{i=1}^{n} \mathcal{I}_i$, where $\mathcal{I}_0 = \emptyset$ and $\mathcal{I}_i$ ($1 \leq i \leq n$) is the possibly infinite set obtained by exhaustive, fair application of rules P_i to $\mathcal{I}_{i-1}$. Note that this only leads to a practical (and overall fair) procedure if each $\mathcal{I}_i$ is finite, which may depend on the chosen order of rule applications.[2] We follow Carral *et al.* [2019b] and require that, within each stratum P_i, rules without existential variables are applied first (the so-called *Datalog-first chase*). This assumption simplifies presentation and is justified for the system used in our evaluation.

Example 2. Contrary to expectations in normal logic programming, the stratified chase does not yield a unique result, not even up to homomorphic renaming of nulls. Consider the fact $q(a)$ and the rules

$$\begin{aligned}
r_1: &\quad r(X,X) \leftarrow q(X)\\
r_2: &\quad \exists V. r(X,V) \leftarrow q(X)\\
r_3: &\quad e(Y,Y) \leftarrow r(X,Y)\\
r_4: &\quad p() \leftarrow r(X,Y) \wedge r(X,Z) \wedge \textbf{not}\, e(Y,Z)
\end{aligned}$$

A stratification s must satisfy $s(q(a)) \leq s(r_1) \leq s(r_3)$, $s(q(a)) \leq s(r_2) \leq s(r_3)$, and $s(r_3) < s(r_4)$. A valid partitioning would be $\{q(a), r_1\}, \{r_2, r_3\}, \{r_4\}$, which yields a stratified chase $\{q(a), r(a,a), e(a,a)\}$. Note that r_2 is not applicable since $r(a,a)$ is already derived before r_2 is considered. However, if we consider the partition $\{q(a), r_2\}, \{r_1, r_3\}, \{r_4\}$, which is also a valid stratification, then the stratified chase is $\{q(a), r(a,n), r(a,a), e(a,a), e(n,n), p()\}$, where n is a named null introduced when applying r_2 in the first stratum (as it is not satisfied at this point).

Now we can simulate sets in a similar way as in Definition 2, but using existential rules and nulls instead of DLP^f rules and function terms. To this end, we replace rule (7) by

$$\exists Y. su(X,S,Y) \leftarrow get\_su(X,S) \tag{23}$$

while all other auxiliary rules remain as before. The translation of Datalog(S) rules to existential rules likewise remains the same as in Definition 2. This produces existential rules with negation if the input rules contain negation but no disjunction. However, stratification can be lost if auxiliary predicates like get_su are needed in every stratum. To address this, each stratum P_i will use distinct copies of each auxiliary rule, using indexed predicates such as su_i and get_su_i. The resulting set of existential rules with negation is denoted $\mathrm{nex}(P_i)$, and for Datalog(S) program P with negation and stratification $P_1, \ldots, P_n$, we define $\mathrm{nex}(P) = \mathrm{nex}(P_1) \cup \bigcup_{i=2}^{n} \mathrm{nex}(P_i) \cup \{su_i(X,S,U) \leftarrow su_{i-1}(X,S,U)\}$. Then $\mathrm{nex}(P)$ is stratified. Since every null n that is introduced by a chase over $\mathrm{nex}(P)$ first appears in a fact $su(t,s,n)$, we can extend the notation introduced before Theorem 3 and associate n with the set $[n] \coloneqq [f_\cup(t,s)]$.

Theorem 7. *If P is a stratified Datalog(S) program with negation, then every stratified, Datalog-first chase $\mathcal{I}$ of $\mathrm{nex}(P)$ is finite and $[\mathcal{I}] \coloneqq \{p([t_1], \cdots, [t_n]) \mid \hat{p}(t_1, \cdots, t_n) \in \mathcal{I}\}$ is the unique stable model of P.*

[2]This is typical for the standard chase; Krötzsch *et al.* [2019] give an introductory discussion.

$$\begin{aligned}
same(C,C) &\leftarrow class(C)\\
ind(A,C) &\leftarrow sc(A,B) \wedge sc(B,C) \wedge \textbf{not}\, same(A,B)\\
&\quad\ \wedge \textbf{not}\, same(B,C)\\
sc^-(A,C) &\leftarrow sc(A,C) \wedge \textbf{not}\, ind(A,C)\\
&\quad\ \wedge \textbf{not}\, same(A,C)
\end{aligned}$$

Figure 2: Rules for transitive reduction

$$\begin{aligned}
same(C,C) &\leftarrow class(C)\\
in\_chain(C) &\leftarrow class(C) \wedge \textbf{not}\, out\_chain(C)\\
out\_chain(C) &\leftarrow class(C) \wedge \textbf{not}\, in\_chain(C)\\
comp(C) &\leftarrow class(C) \wedge sc(C,D) \wedge in\_chain(D)\\
&\quad\ \wedge \textbf{not}\, same(C,D)\\
comp(D) &\leftarrow class(C) \wedge sc(C,D) \wedge in\_chain(C)\\
&\quad\ \wedge \textbf{not}\, same(C,D)\\
&\leftarrow in\_chain(C) \wedge comp(C)\\
&\leftarrow out\_chain(C) \wedge \textbf{not}\, comp(C)
\end{aligned}$$

Figure 3: Rules for maximal antichains

The previous result is remarkable in the light of Example 2, since it identifies a case where a classical stratification can safely be applied to existential rules with negation.

This completes our approach of using existential rule reasoners for DLP(S)-reasoning: for a DLP(S) program P, we conduct a stratified chase over $\mathrm{nex}(\mathrm{grnd}(P))$ to infer facts of the form $rule_r(\boldsymbol{t})$ and $in(c,n)$, from which we can construct ground instances of DLP(S) rules as in Proposition 6.

6 Evaluation

To evaluate the practical feasibility of our approaches, we developed prototypical implementations of the necessary procedures and combined them with existing reasoning engines to compute stable models. The first approach (LAZY) uses the transformation $\mathrm{dlp^f}(P)$ and the lazy ASP solver *Alpha*[3] [Weinzierl, 2017]. The second approach (EXRULES) implements grounding with existential rules using the *Rulewerk* library with the *VLog* reasoner[4] [Carral *et al.*, 2019a] to produce a grounded file in *aspif* format, to which we apply the ASP solver *clasp* v3.2.1 [Gebser *et al.*, 2007]. Our prototype code will be published as open source as soon as possible (after double-blind review). As a third scenario DLVCOMP, we use the native implementation of sets provided in *DLV-complex* [Calimeri *et al.*, 2009].

As a challenging task for set-based reasoning, we use a rule-based reasoning method for the expressive description logic Horn-$\mathcal{ALC}$ by Carral *et al.* [2019b]. Given an ontology that is syntactically transformed into facts, the rules compute the inferred subclass relationships that follow from the ontology – a problem that is EXPTIME-complete in data complexity. Following Carral *et al.*, we omit auxiliary rules (11)–(20)

[3]https://github.com/alpha-asp/Alpha, *master*, 29 Aug 2021

[4]https://github.com/karmaresearch/vlog, *master*, 30 Aug 2021

ID	Name	#Classes	#scAx	#scInf
00668	vaccine	6,481	6,090	94,605
00368	bp	16,298	25,626	187,379
00371	bp×cell	17,810	27,171	198,683
00541	mf×anatomy	18,955	23,592	168,338
00375	bp×cell. comp.	27,490	44,949	352,738
00395	bp×anatomy	36,752	55,022	437,770
00533	mf×ChEBI	52,726	61,148	911,856
00477	Gazetteer	150,976	10,606	11,031

Table 1: Ontologies used in experiments with their names ("mf": molecular function; "bp": biological processes) and numbers of class names, stated subclass relations, and inferred subclass relations

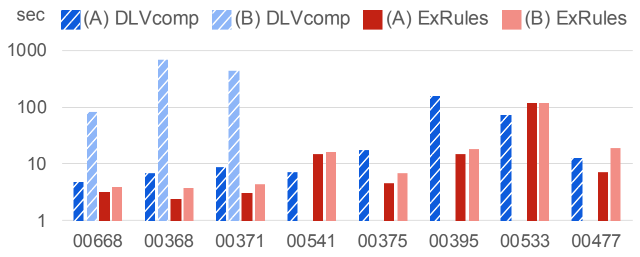

Figure 4: Evaluation results for DLVcomp and ExRules (time in sec, log-scale; empty columns indicate timeouts).

from $\mathrm{dlp}^{\mathrm{f}}(P)$ and the corresponding existential rules version, since they would not contribute new derivations here. The rules define a binary predicate *sc* that represents the inferred subclass hierarchy, and a unary predicate *class* that marks class names in the ontology.

On top of this basic classification task, we specified two tasks that require additional expressive power of DLP(S). First, we compute the *transitive reduction* of the class hierarchy, i.e., a directed graph that contains only direct subclass relations without the transitive bridges (Task A). The additional rules in Figure 2 were used for this task. While Task A is inherently non-monotonic, these rules are still stratified and lead to a unique stable model.

As a second task, we compute maximal antichains (i.e., sets of incomparable classes) in the class hierarchy (Task B). This task admits many solutions, each a stable model, and we asked systems to compute five stable models in this case. The rules we used are shown in Figure 3.

Experiments were executed with eight different ontologies from the Oxford Ontology Repository[5] as shown in Table 1. In each case, we deleted axioms that are not supported in the description logic Horn-$\mathcal{ALC}$ and normalised the remaining axioms as required by the classification rules [Carral *et al.*, 2019b]. Our final evaluation data sets are provided in the auxiliary material. Our evaluation computer is a mid-end server (Debian Linux 9.13; Intel Xeon CPU E5-2637v4@3.50GHz; 384GB RAM DDR4; 960GB SSD), though most experiments did not use more than 8GB of RAM. A timeout of 15min was used in all experiments. We report results of single runs, as we observed almost no time variations across runs.

[5]https://www.cs.ox.ac.uk/isg/ontologies/

The results of scenarios DLVcomp and ExRules on both tasks are shown in Figure 4, with missing values indicating a timeout. Scenario Lazy is omitted from the figure since it did not succeed in any of the tasks. The most difficult problem that we could solve with method Lazy was the basic classification (without any of the additional rules for tasks A or B) of the *vaccine* ontology 00668, which took 248sec (vs. $<$4sec needed in the other approaches).

Overall, we find that ASP with set terms can be used to solve complex problems on large real-world inputs. Both the native implementation in DLV-complex and our new existential-rule grounding performed well across a range of inputs. Contrary to our expectations, the lazy grounding approach was much less effective. Since Alpha solved instances of the base task, which already contains (7) as the only rule that can create infinitely many domain elements, we speculate that the algorithm behaves correctly in principle but is sometimes affected by additional optimisations or heuristics.

DLVcomp and ExRules often showed similar performance on Task A, though we can see some differences, most notably an order of magnitude advantage for VLog+clasp on 00395. Interestingly, ExRules showed almost the same performance on Task B, whereas times for DLVcomp increased such that only smaller problems could be solved. Again, this might be caused by a particular weakness of the implementation that is not conceptual. For ExRules, most of the time was spent in the grounding phase, whereas solving the grounded files was relatively quick. Overall, we see promise in the performance of our existential grounding prototype, but the absolute run times should not be considered the main outcome of our experiments, given the diversity of the underlying systems in terms of maturity and recency.

7 Conclusions

Set terms are a natural and intuitive modelling construct, and clearly a useful addition to practical ASP tools. This seems to be the first work, however, that highlights their expressive advantages and puts them to concrete use on real-world problems that (due to their computational complexity) cannot be solved with plain ASP. Our work indicates that various existing ASP systems can feasibly be applied to these new kinds of tasks. The competitive performance of our prototypical existential-rule grounder encourage further research into combinations of existential rules and ASP.

We were surprised by the weak performance of lazy grounding approaches in our experiments, but we still consider them promising. Our programs can serve as benchmarks for further improving such implementations. An operational alternative to DLV-complex would to implement own set datatypes through extension points of modern ASP solvers, such as clingo's *@-terms* or DLV's *external atoms*. Finally, there is potential for investigating further uses of set terms in ASP for improving performance or readability. Candidate applications can be found, e.g., in the area of abstract argumentation, where ASP has been successfully applied [Gaggl *et al.*, 2015]. Our experiments with description logics (DLs) also suggest the use of sets for a native integration of DLs and ASP, e.g., in the style of *dl-programs* [Eiter *et al.*, 2004].

Acknowledgments

We thank an anonymous reviewer of IJCAI'22 for pointing out the potential utility of clingo *@-terms* and DLV *external atoms* for implementing complex-value extensions of ASP. This work was partly supported by *Deutsche Forschungsgemeinschaft* (DFG) in project 389792660 (TRR 248, Center for Perspicuous Systems), by the *Bundesministerium für Bildung und Forschung* (BMBF) in projects 01IS20056_NAVAS and ScaDS.AI (Center for Scalable Data Analytics and Artificial Intelligence), and by the Center for Advancing Electronics Dresden (cfaed).

References

[Bonatti, 2004] Piero A. Bonatti. Reasoning with infinite stable models. *Artif. Intell.*, 156(1):75–111, 2004.

[Brewka *et al.*, 2011] Gerhard Brewka, Thomas Eiter, and Miroslaw Truszczynski. Answer set programming at a glance. *Commun. ACM*, 54(12):92–103, 2011.

[Calimeri *et al.*, 2008] Francesco Calimeri, Susanna Cozza, Giovambattista Ianni, and Nicola Leone. Computable functions in ASP: theory and implementation. In *Proc. 24th Int. Conf. on Logic Programming (ICLP'08)*, volume 5366 of *LNCS*, pages 407–424. Springer, 2008.

[Calimeri *et al.*, 2009] Francesco Calimeri, Susanna Cozza, Giovambattista Ianni, and Nicola Leone. An ASP system with functions, lists, and sets. In *Proc. 10th Int. Conf. on Logic Programming and Nonmonotonic Reasoning (LPNMR'09)*, volume 5753 of *LNCS*, pages 483–489. Springer, 2009.

[Carral *et al.*, 2019a] David Carral, Irina Dragoste, Larry González, Ceriel Jacobs, Markus Krötzsch, and Jacopo Urbani. VLog: A rule engine for knowledge graphs. In *Proc. 18th Int. Semantic Web Conf. (ISWC'19, Part II)*, volume 11779 of *LNCS*, pages 19–35. Springer, 2019.

[Carral *et al.*, 2019b] David Carral, Irina Dragoste, Markus Krötzsch, and Christian Lewe. Chasing sets: How to use existential rules for expressive reasoning. In *Proc. 28th Int. Joint Conf. on Artificial Intelligence (IJCAI'19)*, pages 1624–1631. ijcai.org, 2019.

[Dantsin *et al.*, 2001] Evgeny Dantsin, Thomas Eiter, Georg Gottlob, and Andrei Voronkov. Complexity and expressive power of logic programming. *ACM Computing Surveys*, 33(3):374–425, 2001.

[Eiter and Simkus, 2010] Thomas Eiter and Mantas Simkus. FDNC: decidable nonmonotonic disjunctive logic programs with function symbols. *ACM Trans. Comput. Log.*, 11(2):14:1–14:50, 2010.

[Eiter *et al.*, 2004] Thomas Eiter, Thomas Lukasiewicz, Roman Schindlauer, and Hans Tompits. Combining answer set programming with description logics for the Semantic Web. In *Proc. 9th Int. Conf. on Principles of Knowledge Representation and Reasoning (KR'04)*, pages 141–151. AAAI Press, 2004.

[Faber, 2020] Wolfgang Faber. An introduction to answer set programming and some of its extensions. In *Tutorial Lectures 16th Int. Summer School on Reasoning Web. Declarative Artificial Intelligence*, volume 12258 of *LNCS*, pages 149–185. Springer, 2020.

[Gaggl *et al.*, 2015] Sarah Alice Gaggl, Norbert Manthey, Alessandro Ronca, Johannes Peter Wallner, and Stefan Woltran. Improved answer-set programming encodings for abstract argumentation. *Theory Pract. Log. Program.*, 15(4-5):434–448, 2015.

[Gebser *et al.*, 2007] Martin Gebser, Benjamin Kaufmann, André Neumann, and Torsten Schaub. clasp: A conflict-driven answer set solver. In *LPNMR*, volume 4483 of *Lecture Notes in Computer Science*, pages 260–265. Springer, 2007.

[Gebser *et al.*, 2018] Martin Gebser, Nicola Leone, Marco Maratea, Simona Perri, Francesco Ricca, and Torsten Schaub. Evaluation techniques and systems for answer set programming: a survey. In *Proc. 27th Int. Conf. on Artificial Intelligence, (IJCAI'18)*, pages 5450–5456. ijcai.org, 2018.

[Krötzsch *et al.*, 2019] Markus Krötzsch, Maximilian Marx, and Sebastian Rudolph. The power of the terminating chase. In *Proc. 22nd Int. Conf. on Database Theory (ICDT'19)*, volume 127 of *LIPIcs*, pages 3:1–3:17. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2019.

[Lefèvre *et al.*, 2017] Claire Lefèvre, Christopher Béatrix, Igor Stéphan, and Laurent Garcia. Asperix, a first-order forward chaining approach for answer set computing. *Theory Pract. Log. Program.*, 17(3):266–310, 2017.

[Lifschitz and Turner, 1994] Vladimir Lifschitz and Hudson Turner. Splitting a logic program. In *Proc. 11th Int. Conf. on Logic Programming (ICLP'94)*, pages 23–37. MIT Press, 1994.

[Lifschitz, 2019] Vladimir Lifschitz. *Answer Set Programming*. Springer, 2019.

[Ortiz *et al.*, 2010] Magdalena Ortiz, Sebastian Rudolph, and Mantas Simkus. Worst-case optimal reasoning for the Horn-DL fragments of OWL 1 and 2. In *Proc. 12th Int. Conf. on Principles of Knowledge Representation and Reasoning (KR'10)*, pages 269–279. AAAI Press, 2010.

[Taupe *et al.*, 2019] Richard Taupe, Antonius Weinzierl, and Gerhard Friedrich. Degrees of laziness in grounding – effects of lazy-grounding strategies on ASP solving. In *Proc. 15th Int. Conf. on Logic Programming and Nonmonotonic Reasoning (LPNMR'19)*, volume 11481 of *LNCS*, pages 298–311. Springer, 2019.

[Weinzierl, 2017] Antonius Weinzierl. Blending lazy-grounding and CDNL search for answer-set solving. In *Proc. 14th Int. Conf. on Logic Programming and Nonmonotonic Reasoning (LPNMR'17)*, volume 10377 of *LNCS*, pages 191–204. Springer, 2017.