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 \(\text{coNP-complete}\), 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 \(\text{ExpTime} = \Pi^2_1\)), 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].

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 \(\text{DLV}-\text{complex}\). 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.

\[ e^+(X, Y) \leftarrow e(X, Y) \]
\[ e^+(X, Y) \leftarrow e^+(X, Z) \land e(Z, Y) \]
\[ c(\{X\}) \leftarrow v(X) \]
\[ c(S \cup \{Y\}) \leftarrow c(S) \land X \in S \land e^+(X, Y) \land e^+(Y, X) \]
\[ \text{subc}(S_1) \leftarrow c(S_1) \land c(S_2) \land S_1 \subseteq S_2 \land \text{not} S_2 \subseteq S_1 \]
\[ \text{scc}(S) \leftarrow c(S) \land \text{not subc}(S) \]

Figure 1: Strongly connected components in ASP with sets.
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.

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 \textbf{obj} and a sort of sets \textbf{set}. A signature of DLP(S) is based on countably infinite sets of object constants \textbf{C}_\textbf{obj} (typically \(a, b, c\)), object variables \textbf{V}_\textbf{obj} (typically \(X, Y, Z\)), set variables \textbf{V}_\textbf{set} (typically \(S, U, V\)), and predicate names \textbf{P} including special predicates \(\{e, \subseteq\} \subseteq \textbf{P}\). The signature of a predicate symbol \(p\) is a tuple \(\text{sig}(p) = (\text{vars}_1, \ldots, \text{vars}_n)\) of sorts, and \(n\) is called its arity. We have \(\text{sig}(\varepsilon) = (\text{obj, obj})\) and \(\text{sig}(\subseteq) = (\text{set, set})\).

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 \((S_1 \cup \ldots \cup S_n)\) where \(S_1, S_2\) are set terms. We use \(\{t_1, \ldots, t_n\}\) as abbreviation for \((\{t_1\} \cup \{t_2\} \cup \ldots \cup \{t_n\})\) and omit parentheses for \(X\) and \(\text{obj}\) if it contains no variables.

An atom is an expression \(p(t_1, \ldots, t_n)\) where \(p \in \textbf{P}\) with \(\text{sig}(p) = (\text{vars}_1, \ldots, \text{vars}_n)\) and \(t_i\) is a term of sort \(\text{vars}_i\). We write \(p \in t\) as \(p \in S\) and \(\subseteq \{s_1, s_2\}\) as \(s_1 \subseteq s_2\). A literal is an atom or a negated atom \(\not p\). We sometimes treat conjunctions or disjunctions of literals as sets of literals. For a formula \(F\), \(\text{preds}(F)\) 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 conjunction 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 \(p \in \textbf{P}\) \(\setminus \{\varepsilon, \subseteq\}\), and
3. the special predicates \(\varepsilon\) 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 \(q(S) \leftarrow p(S \cup U)\) (which violates condition 2) would produce exponentially many derivations in the size of any set \(R\) for which \(p(R)\) holds.

A substitution \(\sigma\) is a sort-preserving partial mapping from variables to terms. \(F\sigma\) denotes the result of applying \(\sigma\) 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 \(\text{obj}(P)\) be the set of all object constants in \(P\) (including those used in set terms), and let \(\text{set}(P)\) be a set of terms of the form \(\{t_1, \ldots, t_n\}\) that bijectively correspond to the powerset of \(\text{obj}(P)\). The ground\(\text{ing}\) \(\text{ground}(P)\) 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 \(\text{obj}(P)\) and \(\text{set}(P)\), respectively; and (2) replacing each of the resulting set terms \(s\) by the corresponding term from \(\text{set}(P)\) that represents the same set under the usual interpretation of \(\emptyset\), \(\{\}\), and \(U\).

An interpretation \(I\) for \(P\) is a set of ground facts that only uses terms from \(\text{obj}(P)\) and \(\text{set}(P)\), 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 \(t\)). \(I\) satisfies (or is a model of) a ground atom \(\alpha\) if \(\alpha \in I\). \(I\) satisfies a ground conjunction \(B\) (disjunction \(H\)) if \(B \subseteq I\) (\(H \cap I \neq \emptyset\)), and a positive ground rule \(H \leftarrow B\) if \(H \in I\) and \(H\) does not satisfy \(B\). The reduction \(P^T\) of \(P\) with respect to \(I\) is obtained from \(\text{ground}(P)\) by (1) deleting every negated atom \(\not p\) with \(\alpha \not \in I\) and (2) deleting every rule with a negated atom \(\not p\) with \(\alpha \neq I\). \(I\) is a stable model of \(P\) if it is a subminimal model of \(P^T\). A fact \(\sigma\) is (cautiously) entailed by \(P\) if every stable model of \(P\) contains the fact \(\sigma\), obtained by replacing set terms in \(\alpha\) with their representative in \(\text{set}(P)\).

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 \(\alpha\) is con\(\text{EXP\-TIME}\) \(\text{NP}\)-complete. If \(P\) does not contain \(\forall\), the problem is con\(\text{EXP\-TIME}\)-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 \(\Pi^P_2\)-complete respectively con\(\text{NP}\)-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 \(\text{suc}\(c\), \(2\), \(\ldots\), suc\(c\), \((\ell - 1)\), \(\ell\)), a chain of length \(2\ell\) is characterised by a predicate \(c\), defined as follows:

\[
\begin{align*}
\text{suc}\(c\), (X, Y) &\leftarrow \text{suc}\(c\), (X, Y) \\
\text{suc}\(c\), (X, Z) &\leftarrow \text{suc}\(c\), (X, Y) \wedge \text{suc}\(c\), (Y, Z) \\
\text{n}(\emptyset, \{\}, 1, \emptyset) &\leftarrow \\
\text{n}(\text{U, \{\}, \text{U\)}, \text{V, \text{V\)}} &\leftarrow \text{n}(\text{U, \text{U\)}, \text{U\)}, \text{n}(\text{U, \text{V\)}, \text{X}) \\
&\wedge \text{suc}\(c\), (X, Y) \\
\text{n}(\text{U, \{\}, \text{Y\}}, \emptyset) &\leftarrow \text{n}(\text{U, \text{U\)}, \text{U\)}, \text{n}(\text{U, \text{Y\}}, \text{X\)} \\
&\wedge \text{suc}\(c\), (X, Y)
\end{align*}
\]
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 \( \{1, \ldots, t\} \) 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 \( c_0 \) (the empty set) and a function \( f_1 \), such that, e.g., \( f_1(a, f_1(b, c_0)) \) represents the set \( \{a, b\} \). However, we cannot generally represent \( \{c, s\} \cup \{t\} \) by \( f_1(c, s, t) \), since this would lead to redundant representations like \( f_1(a, f_1(b, c_0)) \). Instead, we use facts of the form \( su(c, s, t) \) – where \( su \) stands for singleton union – to express \( \{c\} \cup \{s, t\} \). In particular, \( su(c, s, s) \) means \( c = s \), 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:

\[
\begin{align*}
su(X, S, f_1(X, S)) & \leftarrow get_su(X, S) \land \text{not in}(X, S) \quad (7) \\
su(X, U, U) & \leftarrow su(X, U) \quad (8) \\
su(Y, U, U) & \leftarrow su(X, U) \land in(Y, S) \quad (9) \\
in(X, S) & \leftarrow su(X, S) \quad (10)
\end{align*}
\]

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

\[
\begin{align*}
u(S, c_0, S) & \leftarrow get_u(S, c_0) \quad (11) \\
u(S, f_1(X, T), U) & \leftarrow get_u(S, f_1(X, T)) \land su(X, S, \bar{S}) \land u(S, T, U) \quad (12) \\
get_su(X, S) & \leftarrow get_u(S, f_1(X, T)) \quad (13) \\
get_u(S, T) & \leftarrow get_u(S, f_1(X, T)) \land su(X, S, \bar{S}) \quad (14)
\end{align*}
\]

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 \( dlp^f(r) \). For every DLP(S) predicate \( p \) of arity \( n \), let \( \tilde{p} \) be a unique fresh predicate of arity \( n \) and signature \( \tilde{p}(s) = \{\text{obj}_1, \ldots, \text{obj}_j\} \), 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 \( \tilde{p} \), 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 \( s \subseteq u \), ordered such that subterms occur before their superterms. Then \( dlp^f(r) \) consists of the following rules:

\[
\begin{align*}
H' & \leftarrow B' \land \bigwedge_{i=1}^k \beta(s_i) \quad (15) \\
\alpha(s_{j+1}) & \leftarrow B^+ \land \bigwedge_{i=1}^j \beta(s_i) \quad j \in \{0, \ldots, k-1\} \quad (16)
\end{align*}
\]

where we define, for \( v = \{t\} \), \( \alpha(v) = get_{-}su(t, c_0) \) and \( \beta(v) = su(t, c_0, V_s) \); and, for \( v = s \subseteq u \), \( \alpha(v) = get_{-}u(V_s, V_u) \) and \( \beta(v) = u(V_s, V_u) \).

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

\[
\begin{align*}
H' & \leftarrow B' \land su(x, q(V_s, x), V_{su(x)}) \land u(V_s, V_{su(x)}) \land get_{-}su(x, q(V_s, x), V_{su(x)}) \land get_{-}u(V_s, V_{su(x)}) \land B^+ \land \text{not } su(x, q(V_s, x), V_{su(x)})
\end{align*}
\]

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, \( f_1(a, f_1(b, c_0)) \) and \( f_1(b, f_1(a, c_0)) \) both represent \( \{a, b\} \). We therefore explicitly compute equality of sets as follows:

\[
\begin{align*}
sub(c_0, c_0) & \leftarrow \quad (17) \\
sub(c_0, S) & \leftarrow in(X, S) \quad (18) \\
sub(T, U) & \leftarrow su(X, S, T) \land sub(S, U) \land \text{not } in(X, U) \quad (19) \\
eq(X, T) & \leftarrow sub(T, S) \land \text{not } sub(T, S) \quad (20)
\end{align*}
\]

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

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

\[
\tilde{p}(X_1, \ldots, X_n) \land \text{eq}(X_i, Y) \leftarrow \tilde{p}(X_1, \ldots, X_n) \land \text{eq}(X_i, Y)
\]

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

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 \( dlp^f(P) \) are finite.
We can convert ground terms $s$ that contain $c_0$ or $f_i$ to canonical set terms $[s] \in \text{set}(P)$ in the obvious way: $[c_0] = \emptyset$ and $[f_i(t,s)]$ is the representation of $\{t\} \cup [s]$ in $\text{set}(P)$. Moreover, let $[t] = t$ for other terms (not representing sets). Then, for every stable model $\mathcal{J}$ of $\text{dlp}^g(P)$, we find an interpretation $[\mathcal{J}] = \{p([t_1], \cdots, [t_n]) \mid p(t_1, \cdots, t_n) \in \mathcal{J}\}$.

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

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 $FG$ as the class of all DLP$^f$ programs on which their grounding is finite. However, although our programs $\text{dlp}^g(P)$ 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 $\text{dlp}^g(P) \notin FG$.

Indeed, it can be verified in practice that grounders such as gringo and iDLV do not terminate on $\text{dlp}^g(P)$ 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 $\text{dlp}^g(P)$ 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; Taupé et al., 2019] and ASP/RiX [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 $\text{dlp}^g(P)$.

Alpha therefore produces a complete set of (necessarily finite) stable models of $\text{dlp}^g(P)$, 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_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 $s : P \rightarrow \mathbb{N}$ maps rules to natural numbers such that, for all pairs of rules $r_1, r_2 \in P$ of form $r_1 : H_1 \leftarrow B^+_1 \wedge B^-_1 \in P$, and all $p \in \text{preds}(H_2)$: (1) if $p \in \text{preds}(B^+_1)$, then $s(r_1) \geq s(r_2)$, and (2) if $p \in \text{preds}(B^-_1)$, 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 $P^1 \subseteq P$ be the largest disjunction-free, stratified subset of $P$ such that predicates in heads of $P \setminus P^1$ do not occur in $P^1$. A predicate is certain if it occurs only in $P^1$. The Datalog(S) program $\text{grnd}(P)$ contains all facts in $P$ and, for each non-fact rule $H \leftarrow B^+ \wedge B^- \in P$, the rules

\[ \text{rule}_r(X) \leftarrow B^+ \wedge B^- \]  
\[ \wedge_{\alpha \in H} \alpha \leftarrow \text{rule}_r(X) \]

where $\text{rule}_r$ is a dedicated fresh predicate for $r$, $X$ is a list of all variables in $r$, and $B^1 = \{ \text{not } \alpha \in B^- \mid \text{the predicate of } \alpha \text{ is certain} \}$. For a ground atom of the form $r[c]$, the ground rule $r[c]$ is obtained by substituting $X \mapsto c$ to $r$. We get a simple but correct grounding:

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

To compute the stable model of $\text{grnd}(P)$, 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 $\exists Y \cdot H \leftarrow B^+ \wedge B^-$, where $H$ and $B^+$ are conjunctions of atoms, $B^-$ is a conjunction of negated atoms, $Y$ 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 $\exists Y \cdot H \leftarrow B^+ \wedge B^-$ to a set $I$ of facts (using constants and possibly nulls) if (1) there is a substitution $\theta$ with $B^+ \theta \subseteq I$ and $B^- \cap I = \emptyset$, and (2) the rule is not already satisfied under $\theta$, i.e., $I \not\models \exists Y \cdot (H \theta)$. If this holds, we apply the rule by extending $I$ with $H \theta'$, where $\theta'$ is an extension of $\theta$ that maps each $Y \in Y$ to a fresh null.

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

\[1\text{This can be viewed as a kind of split program in the sense of Lifschitz and Turner [1994].}\]
that has a stratification $P_1, \ldots, P_n$, the stratified chase $I$ is $I = \bigcup_{i=1}^{n} I_i$, where $I_0 = \emptyset$ and $I_i$ ($1 \leq i \leq n$) is possibly an infinite set obtained by exhaustive, fair application of rules $P_i$ to $I_{i-1}$. Note that this only leads to a practical (and overall fair) procedure if each $I_i$ is finite, which may depend on the chosen order of rule applications. 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

1. $r_1 : r(X, X) \leftarrow q(X)
2. \exists V. r(X, V) \leftarrow q(X)
3. e(Y, Y) \leftarrow r(X, Y)
4. p() \leftarrow r(X, Y) \land r(X, Z) \land \neg e(Y, Z)

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 rules and function terms. To this end, we replace rule (7) by

$$\exists y. su(X, Y, Y) \leftarrow get_{\text{su}}(X, Y)$$

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 $get_{\text{su}}$ 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 $get_{\text{su}_i}$. The resulting set of existential rules with negation is denoted $nex(P_i)$, and for Datalog(S) program $P$ with negation and stratification $P_1, \ldots, P_n$, we define $nex(P) = \bigcup_{i=1}^{n} nex(P_i) \cup \{su_i(X, U) \leftarrow su_{i-1}(X, U)\}$. Then $nex(P)$ is stratified. Since every null $n$ that is introduced over $nex(P)$ first appears in a fact $su(t, s, n)$, we can extend the notation introduced before Theorem 3 and associate $n$ with the set $[n] = \{j \mid j(t, s)\}$.

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

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 $nex(grnd(P))$ to infer facts of the form $rule_i(t)$ 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 (Lazzy) uses the transformation $dlp^I(P)$ and the lazy ASP solver Alpha³ [Weinzierl, 2017]. The second approach (EXRULES) implements grounding with existential rules using the Rulewerk library with the VLog reasoner⁴ [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-ACCC 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).

³https://github.com/alpha-asp/Alpha, master, 29 Aug 2021
⁴https://github.com/karmaresearch/vlog, master, 30 Aug 2021
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 include 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].

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

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

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

from dlp(\(P\)) 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\(^3\) as shown in Table 1. In each case, we deleted axioms that are not supported in the description logic Horn-\(ALC\) 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.

\(^3\)https://www.cs.ox.ac.uk/isg/ontologies/
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


