Simulating Sets in Answer Set Programming

As presented at the 31st Int. Join Conf. on Artificial Intelligence (IJCAI'22)

Sarah Alice Gaggl, Philipp Hanisch, Markus Krötzsch

Department of Computer Science
TU Dresden

TECHNISCHE UNIVERSITÄT DRESDEN

International Center for Computational Logic

Answer Set Programming with Sets

Approach: Extend ASP with terms that represent finite sets of domain elements.

$\leadsto$ useful in declarative modelling

Answer Set Programming with Sets

Approach: Extend ASP with terms that represent finite sets of domain elements.

$\rightsquigarrow$ useful in declarative modelling

Example: Defining connected components

$$edge^+(X,Y) \leftarrow edge(X,Y)$$
$$edge^+(X,Y) \leftarrow edge^+(X,Z) \wedge edge(Z,Y)$$

$$comp(\{X\}) \leftarrow vertex(X)$$
$$comp(S \cup \{Y\}) \leftarrow comp(S) \wedge X \in S \wedge edge^+(X,Y) \wedge edge^+(Y,X)$$

$$subComp(S_1) \leftarrow comp(S_1) \wedge comp(S_2) \wedge S_1 \subseteq S_2 \wedge \textbf{not}\ S_2 \subseteq S_1$$
$$maxComp(S) \leftarrow comp(S) \wedge \textbf{not}\ subComp(S)$$

Reasoning with DLP(S)

Key observation:
On a given input, a set term can only represent finitely many sets.

Reasoning with DLP(S)

Key observation:
On a given input, a set term can only represent finitely many sets.

$\leadsto$ finite grounding possible
$\leadsto$ grounding might be exponential in the number of constants (data complexity!)

Reasoning with DLP(S)

Key observation:
On a given input, a set term can only represent finitely many sets.

$\leadsto$ finite grounding possible
$\leadsto$ grounding might be exponential in the number of constants (data complexity!)

These bounds are tight:

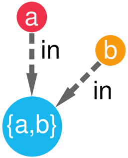
Theorem: Complexity of deciding fact entailment under stable-model semantics:

DLP(S)	coNExpTime$^{\text{NP}}$-complete
DLP(S) without $\vee$	coNExpTime-complete

Hardness holds even if only facts are allowed to vary (data complexity).

Simulating Sets with Function Symbols and Negation

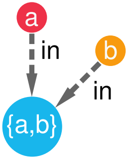
Sets can be encoded in facts:



$in(X, S)$: "$X \in S$"

Simulating Sets with Function Symbols and Negation

Function terms can represent sets:



$f_\cup(\mathrm{b}, f_\cup(\mathrm{a}, \mathrm{c}_\varnothing))$

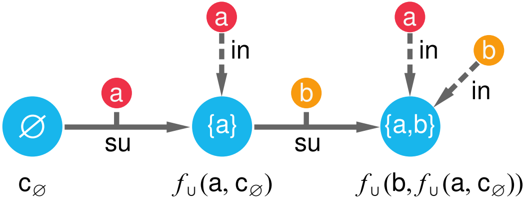
$in(X, S)$: "$X \in S$"

$f_\cup(X, S)$: "$\{X\} \cup S$"

$c_\emptyset$: "$\emptyset$"

Simulating Sets with Function Symbols and Negation

Sets can be constructed element-by-element:



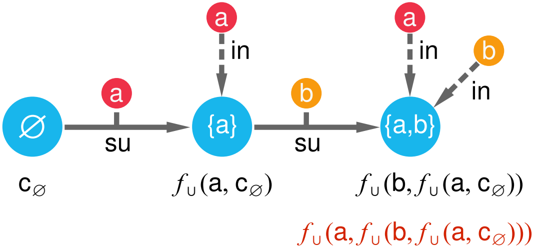
$in(X, S)$: "$X \in S$"

$f_\cup(X, S)$: "$\{X\} \cup S$"

$c_\emptyset$: "$\emptyset$"

$su(X, S, T)$: "$\{X\} \cup S = T$"

Simulating Sets with Function Symbols and Negation



$in(X, S)$: "$X \in S$"

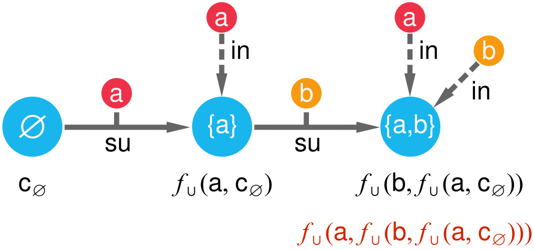
$f_\cup(X, S)$: "$\{X\} \cup S$"

$c_\emptyset$: "$\emptyset$"

$su(X, S, T)$: "$\{X\} \cup S = T$"

Problem: Duplicate elements in terms $\leadsto$ infinitely many terms for finitely many sets

Simulating Sets with Function Symbols and Negation



$in(X, S)$: "$X \in S$"

$f_\cup(X, S)$: "$\{X\} \cup S$"

$c_\emptyset$: "$\emptyset$"

$su(X, S, T)$: "$\{X\} \cup S = T$"

Problem: Duplicate elements in terms $\rightsquigarrow$ infinitely many terms for finitely many sets

Solution: Use negation to prevent duplicates:

$$su(X, S, f_\cup(X, S)) \leftarrow get\_su(X, S) \wedge \textbf{not}\ in(X, S)$$

Finite Stable Models: Theory and Practice

Theorem: The ASP encoding with function symbols has finite stable models, corresponding to the stable models of the encoded DLP(S).

Finite Stable Models: Theory and Practice

Theorem: The ASP encoding with function symbols has finite stable models, corresponding to the stable models of the encoded DLP(S).

Problem: Classical ground-and-solve approaches fail since grounders ignore some negations:

$$su(X, S, f_\cup(X, S)) \leftarrow get\_su(X, S) \wedge \textbf{not}\, in(X, S)$$

Finite Stable Models: Theory and Practice

Theorem: The ASP encoding with function symbols has finite stable models, corresponding to the stable models of the encoded DLP(S).

Problem: Classical ground-and-solve approaches fail since grounders ignore some negations:

$$su(X, S, f_{\cup}(X, S)) \leftarrow get\_su(X, S) \wedge \textbf{not}\; in(X, S)$$

Solution 1: Use lazy grounders

- Lazy ASP systems compute grounding on-demand
  $\leadsto$ non-stratified negations not ignored during grounding
- Systems like **Alpha** [Weinzierl et al.] can produce finite stable models

Finite Stable Models: Theory and Practice

Theorem: The ASP encoding with function symbols has finite stable models, corresponding to the stable models of the encoded DLP(S).

Problem: Classical ground-and-solve approaches fail since grounders ignore some negations:

$$su(X, S, f_{\cup}(X, S)) \leftarrow get\_su(X, S) \wedge \textbf{not}\, in(X, S)$$

Finite Stable Models: Theory and Practice

Theorem: The ASP encoding with function symbols has finite stable models, corresponding to the stable models of the encoded DLP(S).

Problem: Classical ground-and-solve approaches fail since grounders ignore some negations:

$$su(X, S, f_{\cup}(X, S)) \leftarrow get\_su(X, S) \wedge \textbf{not}\, in(X, S)$$

Solution 2: Use existential quantifiers instead of function terms

$$\exists V.su(X, S, V) \leftarrow get\_su(X, S)$$

Finite Stable Models: Theory and Practice

Theorem: The ASP encoding with function symbols has finite stable models, corresponding to the stable models of the encoded DLP(S).

Problem: Classical ground-and-solve approaches fail since grounders ignore some negations:

$$su(X, S, f_{\cup}(X, S)) \leftarrow get\_su(X, S) \wedge \textbf{not}\, in(X, S)$$

Solution 2: Use existential quantifiers instead of function terms

$$\exists V.su(X, S, V) \leftarrow get\_su(X, S)$$

- If $in(X, S)$ holds, then also $su(X, S, S)$, and therefore $\exists V.su(X, S, V)$
  $\rightsquigarrow$ standard redundancy check in existential rule reasoning mimics negation
- Use **existential rule reasoners for grounding**

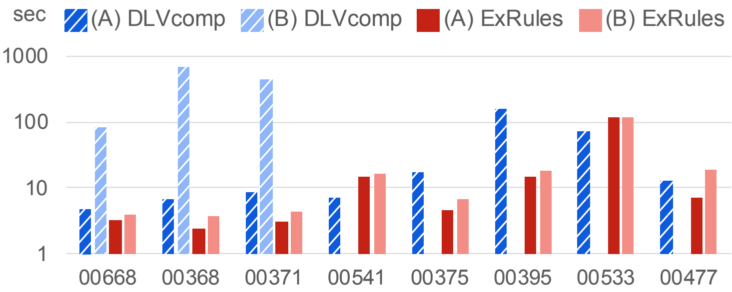
Evaluation: Setup

Use ASP to solve non-monotonic, ExpTime-complete problems for OWL ontologies

- Implement an ontology reasoner in DLP(S)
- Add additional non-monotonic rules to
  (A) compute the reduced class hierarchy
  (B) compute maximal antichains in the class hierarchy
- Evaluated implementations:
  Alpha (lazy grounder + function-term rewriting)
  ExRules (VLog existential rule grounder + clasp solver)
  DLVcomplex (native DLP(S) implementation)
- Test data:
  8 real-world ontologies, entailing 11K–911K subclass relations

Evaluation: Results

- Alpha could not solve task (A) or (B) in any case, but it could compute plain OWL inferences for one ontology
- DLVcomplex could solve (A) in all cases, and (B) for 3 (of 8)
- ExRules could solve (A) and (B) in all cases



⇝ Initial feasibility study – should not exclude any implementation approach yet

Conclusion

We extend disjunctive logic programming with set-terms, study its complexity, and show that it can solve reasoning problems beyond the propositional case.

Main contributions:

- Complexity results for stable-model reasoning with set terms
- Simulation of sets in logic programs with functions **and** finite stable models
- Set-aware grounding using existential rule reasoning
- New grounding implementation and evaluation

Open questions:

- Could lazy grounding approaches perform better here?
- In general: how to make set-related reasoning performance more robust?
- Further use cases to exploit the added expressivity?

Rewatch talk: `https://tinyurl.com/asp-sets-22` IJCAI'22 Poster #174