# $f_{CASP}$: A forgetting technique for XAI based on goal-directed constraint ASP models

Luciana Fidilio-Allende    Joaquín Arias

Grupo de Inteligencia Artificial de la URJC
Center for Intelligent Information Technologies (CETINIA)
Móstoles, Madrid

17 Junio 2024 (PROLE'24)

# Introduction

- Automated Decision Makers, even if they are value-aware, can only be trustworthy if they are capable of explaining their decisions (XAI).
- Logic-based systems, such as s(CASP) [3], a goal-directed execution of Answer Set Programming (ASP) [7], can provide justifications.
  - But the justifications (and the ASP models themselves) may expose sensitive information (violating privacy and/or legacy).

# Introduction

- Automated Decision Makers, even if they are value-aware, can only be trustworthy if they are capable of explaining their decisions (XAI).
- Logic-based systems, such as s(CASP) [3], a goal-directed execution of Answer Set Programming (ASP) [7], can provide justifications.
  - But the justifications (and the ASP models themselves) may expose sensitive information (violating privacy and/or legacy).
- Alternatives: (i) Manipulate the justifications [1], or (ii) apply forgetting, a syntactic transformation that forgets predicates in ASP programs [10].

# Introduction

- Automated Decision Makers, even if they are value-aware, can only be trustworthy if they are capable of explaining their decisions (XAI).
- Logic-based systems, such as s(CASP) [3], a goal-directed execution of Answer Set Programming (ASP) [7], can provide justifications.
  - But the justifications (and the ASP models themselves) may expose sensitive information (violating privacy and/or legacy).
- Alternatives: (i) Manipulate the justifications [1], or (ii) apply forgetting, a syntactic transformation that forgets predicates in ASP programs [10].

Example from [10]

```
person(X) :- ustaff(X).
ustaff(X) :- professor(X).
professor(mary).
```

{ ..., person(mary) }

Result after forgetting `ustaff/1`

```
person(X) :- professor(X).
professor(mary).
```

{ ..., person(mary) }

# Introduction

- Automated Decision Makers, even if they are value-aware, can only be trustworthy if they are capable of explaining their decisions (XAI).
- Logic-based systems, such as s(CASP) [3], a goal-directed execution of Answer Set Programming (ASP) [7], can provide justifications.
  - But the justifications (and the ASP models themselves) may expose sensitive information (violating privacy and/or legislation).
- Alternatives: (i) Manipulate the justifications [1], or (ii) apply forgetting, a syntactic transformation that forgets predicates in ASP programs [10].

## Limitations

(i) sensitive information persists in the model.

(ii) existing proposals of forgetting focus on propositional logic.

# Limitations of state-of-the-art Forgetting techniques

| | (UP) | (SP) | Loops | Commutative | Predicates | Constraints |
|---|---|---|---|---|---|---|
| $f_{SU}$ [8] | Yes | No | Yes | No | No | No |
| $f_{SP}$ [5] | Yes | Limited | No | No | No | No |
| $f^*_{SP}$ [4] | Yes | Limited | Yes | Yes | No | No |
| $f_{AC}$ [6] | Yes | Yes | Yes | Yes | No | No |

- (UP): Uniform Persistence means that the original program and the forgetting result are equivalence even if we add new facts.
- (SP): Strong Persistence is similar to (UP) but adding new rules.
- Loops: Deal with even/odd loops (by adding auxiliary predicates).
- Commutative: Allow iterative application, regardless of the order.

# Limitations of state-of-the-art Forgetting techniques (cont.)

| | (UP) | (SP) | Loops | Commutative | Predicates | Constraints |
|---|---|---|---|---|---|---|
| $f_{SU}$ [8] | Yes | No | Yes | No | No | No |
| $f_{SP}$ [5] | Yes | Limited | No | No | No | No |
| $f_{SP}^{*}$ [4] | Yes | Limited | Yes | Yes | No | No |
| $f_{AC}$ [6] | Yes | Yes | Yes | Yes | No | No |
| $f_{CASP}$ | Yes | Yes | Yes | Yes | WiP | WiP |

**Our proposal $f_{CASP}$** based on $f_{SU}$

- It is simple, iterable, and commutable.
- Uses the compiler of s(CASP) to generate dual rules.
- Supports even loops, denials and odd loops.

... and we believe it would support Predicates and Constraints.

# Background: ASP

- Answer Set Programming (ASP) is based on the stable model semantics [7], supporting non-stratified negation:

```
p :- not q.
q :- not p.
```

Even loop: `{p}, {q}`

```
p :- not q.
q :- p.
```

Odd loop: `no models`

- In this work, we extended ASP with **double default negations** [12]: The clause `p :- not not p.` generates two models: `{p}` and `{}`.
  - Double negations can be modeled as even loops. For example, the predicate `p :- not not p` is transformed into:

```
p :- not neg_p.
neg_p :- not p.
```

# Background: s(CASP)

- It is a top-down, goal-directed interpreter of ASP with Constraints [3].
- Can provide justifications (in natural language) [1].
  - They can be manipulated (to hide sensitive information) using the directive `#show` and the flag `--short`.
- It solves negated atoms (`not p`) against the dual rules of the program (the negation of the rules present in the program) [2]. E.g.:

```
% Original program
p(0).
p(X) :- q(X), not t(X,Y).
```

# Background: s(CASP)

- It is a top-down, goal-directed interpreter of ASP with Constraints [3].
- Can provide justifications (in natural language) [1].
  - They can be manipulated (to hide sensitive information) using the directive `#show` and the flag `--short`.
- It solves negated atoms (`not p`) against the dual rules of the program (the negation of the rules present in the program) [2]. E.g.:

```
% Original program
p(0).
p(X) :- q(X), not t(X,Y).
```

```
% Dual rules
not p(X) :- not p1(X), not p2(X).
not p1(X) :- X\=0.
not p2(X) :- forall(Y, not p2_(X,Y)).
not p2_(X,Y) :- not q(X).
not p2_(X,Y) :- q(X), t(X,Y).
```

# Background: s(CASP)

- It is a top-down, goal-directed interpreter of ASP with Constraints [3].
- Can provide justifications (in natural language) [1].
  - They can be manipulated (to hide sensitive information) using the directive `#show` and the flag `--short`.
- It solves negated atoms (`not p`) against the dual rules of the program (the negation of the rules present in the program) [2]. E.g.:

```
1 % Original program
2 p(0).
3 p(X) :- q(X), not t(X,Y).
```

```
1 % Dual rules
2 not p(X) :- not p1(X), not p2(X).
3 not p1(X) :- X\=0.
4 not p2(X) :- forall(Y, not p2_(X,Y)).
5 not p2_(X,Y) :- not q(X).
6 not p2_(X,Y) :- q(X), t(X,Y).
```

NOTE: The existential quantifier for `Y` (line 3)

... is translated into `forall` (line 4).

# The forgetting technique $f_{CASP}$

# The forgetting technique $f_{CASP}$: (Updated) Algorithm design

$f_{CASP}$ consists on five steps that can be iteratively repeated to forget multiple predicates (consider we want to forget the predicate `p`):

1. Add auxiliary predicates due to even loops, facts and/or missing predicate.

   If `p` is part of an even loop, `not p` is replaced, and `neg_x :- not p` is added.

2. Generate the simplified dual rule(s) using s(CASP).

```
% Clauses of p
p :- t, not u.
p :- not r.
```

```
% s(CASP) dual rules
not p :- not p_1, not p_2.
not p_1 :- not t.
not p_1 :- u.
not p_2 :- r.
```

```
% Simplified dual rule(s)
not p :- not t, r.
not p :- u, r.
```

3. Forget the predicate and its negation.
4. Clean true/false and add double negations to preserve even loops.

   Repeat steps 1 to 4 to forget the next predicate...

5. (Optional) Transform double negations into even loops.

# The forgetting technique $f_{CASP}$: Implementation

```
f_scasp(Flag, [Pred|Preds], P_0, P_Forgetting) :-
    transform_even_loop(Pred, P_0, P_1a, Neg_Pred),        % Step 1
    transform_fact_missing(Pred, P_1a, P_1b),
    transform_auto_calls(Pred, P_1b, P_1c),
    gen_dual(Pred, P_1c, Dual_Rule),                       % Step 2
    forget_pred(Pred, Dual_Rule, P_1c, P_3),               % Step 3
    restore_even_loop(Neg_Pred, P_3, P_4a),                % Step 4
    restore_facts_missing(P_4a, P_4b),
    f_scasp(Flag, Preds, P_4b, P_Forgetting).              % Repeat 1,2,3,4
f_scasp(0, [], P_Forgetting, P_Forgetting).                % Skip Step 5
f_scasp(1, [], P_Forgetting, P_Scasp) :-
    transform_double_negations(P_Forgetting, P_Scasp).     % Step 5
```

- $f_{CASP}$ is implemented under s(CASP) available at: https://gitlab.software.imdea.org/ciao-lang/sCASP.
- Can be invoked using the flag `--forget='LIST'[/F]`.

# The forgetting technique $f_{CASP}$: Running Example

example.pl

```
p :-
  not q.
q :-
  t,
  not u.
q :-
  not r.
r :-
  not s.
s :-
  q,
  not p.
```

scasp example.pl --forget='p,q'/0

```
r :-
  not s.
s :-
  t,
  not u,
  not not neg_1.
s :-
  not r,
  not not neg_1.
neg_1 :-
  t,
  not u.
neg_1 :-
  not r.
```

{ q, s } { r, p }

{ neg_1, s } { r }

# The forgetting technique $f_{CASP}$: Running Example

example.pl

```
p :-
  not q.
q :-
  t,
  not u.
q :-
  not r.
r :-
  not s.
s :-
  q,
  not p.
```

{ q, s } { r, p }

scasp example.pl --forget='p,q'/0

```
r :-
  not s.
s :-
  t,
  not u,
  not not neg_1.
s :-
  not r,
  not not neg_1.
neg_1 :-
  t,
  not u.
neg_1 :-
  not r.
```

{ neg_1, s } { r }

scasp example.pl --forget='p,q'/1

```
r :-
  not s.
s :-
  t,
  not u,
  not neg_2.
s :-
  not r,
  not neg_2.
neg_1 :-
  t,
  not u.
neg_1 :-
  not r.
neg_2 :-
  not neg_1.
```

# Step 1: Add auxiliary predicates due to even loops, facts and/or missing clauses.

## Original program

```
p :- not q.
q :- t, not u.
q :- not r.
r :- not s.
s :- q, not p.   % HERE
```

## Program after step 1

```
p :- not q.
q :- t, not u.
q :- not r.
r :- not s.
s :- q, neg_1.   % HERE
neg_1 :- not p. % HERE
```

# Step 2: Generate the simplified dual rule(s) using s(CASP).

Program after step 1

```
p :- not q.
q :- t, not u.
q :- not r.
r :- not s.
s :- q, neg_1.
neg_1 :- not p.
```

Program after step 2

```
p :- not q.
q :- t, not u.
q :- not r.
r :- not s.
s :- q, neg_1.
neg_1 :- not p.
% Dual:
not p :- q.
```

# Step 3: Forget the predicate and its negation.

Program after step 2

```
p :- not q.
q :- t, not u.
q :- not r.
r :- not s.
s :- q, neg_1.
neg_1 :- not p.   % HERE
% Dual:
not p :- q.
```

Program after step 3

```
                  % HERE
q :- t, not u.
q :- not r.
r :- not s.
s :- q, neg_1.
neg_1 :- q.       % HERE
```

# Step 4: Clean true/false and add double negations to preserve even loops.

Program after step 3

```
q :- t, not u.
q :- not r.
r :- not s.
s :- q, neg_1.  % HERE
neg_1 :- q.
```

Program after step 4

```
q :- t, not u.
q :- not r.
r :- not s.
s :- q, not not neg_1. %HERE
neg_1 :- q.
```

# Repeat Steps 1-4: forget the predicate q

Program after step 4 (for p)

```
q :- t, not u.
q :- not r.
r :- not s.
s :- q, not not neg_1.
neg_1 :- q.
```

Program after step 4 (for p and q)

```
r :- not s.
s :- t, not u, not not neg_1.
s :- not r, not not neg_1.
neg_1 :- t, not u.
neg_1 :- not r.
```

# Step 5 (Optional): Transform double negations into even loops.

Program after step 4 (for p and q)

```
r :- not s.
s :- t, not u, not not neg_1.% HERE
s :- not r, not not neg_1.   % HERE
neg_1 :- t, not u.
neg_1 :- not r.
```

Program after step 5

```
r :- not s.
s :- t, not u, not neg_2. % HERE
s :- not r, not neg_2.    % HERE
neg_1 :- t, not u.
neg_1 :- not r.
neg_2 :- not neg_1.       % HERE
```

- As we mentioned before, this step is optional. By default, it is always performed `F=1`, but it can be disabled by setting `F=0`.
- The resulting program after step 5 can be executed using s(CASP).

# Evaluation

# Evaluation 1: $f_{CASP}$ supports even loops

$P_1$ = Example 3 from [9]

```
a :- p.
b :- q.
p :- not q.
q :- not p.
```

{p, a} {q, b}

$f_{CASP}(P_1, \{p, q\})$

```
a :- not not neg_2.
b :- not not neg_1.
neg_1 :- not not neg_1.
neg_2 :- not neg_1.
```

{a, neg_2} {b, neg_1}

# Evaluation 2: $f_{CASP}$ handle double negations

$P_2$ = Example 4 from [11]

```
p :- not not p.
q :- p.
r :- not p.
```

{p, q} {r}

$f_{CASP}(P_2, \{p\})$

```
q :- not neg_1.
r :- not not neg_1.
neg_1 :- not not neg_1.
```

{q} {r, neg_1}

# Evaluation 3: $f_{CASP}$ is commutative (same result regardless of order)

$P_3$ = Example 1 from [4]

```
a :- p, q.
q :- not p.
p :- not not p.
```

{p} {q}

$f_{CASP}(P_3, \{p, q\})$

```
a :- not neg_1, not not neg_1.
neg_1 :- not not neg_1.
```

$f_{CASP}(P_3, \{q, p\})$

```
a :- not neg_1, not not neg_1.
neg_1 :- not not neg_1.
```

{ } {neg_1}

# Evaluation 4: Comparing $f_{CASP}$ vs. $f_{AC}$

$P_4$ = Example 5 from [6]

```
q :- not not q, b.
a :- q.
c :- not q.
```

{c}

$f_{AC}(P_4, \{q\})$

```
a :- b, δ_q.
c :- not δ_q.
c :- not b.
δ_q :- not not δ_q.
```

{c} {c, $\delta_q$}

$f_{CASP}(P_4, \{q\})$

```
a :- not neg_1, b.
c :- not not neg_1.
neg_1 :- not not neg_1.
neg_1 :- not b.
```

{c, neg_1}

# (Real) use case 1: School place allocation

submitted to ICLP'24

- In the "Comunidad de Madrid", school placements are determined by assigning points based on specific criteria.
  - One criterion is being a victim of gender-based violence.

…legally protected data (Art. 63, Organic Law 1/2004).

```
% Original program
[...]
met_common_requirement :-
  large_family.
met_common_requirement :-
  recipient_social_benefits.
met_common_requirement :-
  disability_status.
met_common_requirement :-
  gender_based_violence_victim.
```

```
% Student 2
gender_based_violence_victim.
sibling_enroll_center.
same_education_district.
come_non_bilingual.
want_bilingual_section.
english_native.
```

# (Real) use case 1: School place allocation

submitted to ICLP'24

- In the "Comunidad de Madrid", school placements are determined by assigning points based on specific criteria.
  - One criterion is being a victim of gender-based violence.

  ...legally protected data (Art. 63, Organic Law 1/2004).

```
% Original program
[...]
met_common_requirement :-
  large_family.
met_common_requirement :-
  recipient_social_benefits.
met_common_requirement :-
  disability_status.
met_common_requirement :-
  gender_based_violence_victim.
```

```
% Student 2
gender_based_violence_victim.
sibling_enroll_center.
same_education_district.
come_non_bilingual.
want_bilingual_section.
english_native.
```

```
% Result after forgetting
% for Student 2
[...]
met_common_requirement :-
  large_family.
met_common_requirement.
```

# (Real) use case: School place allocation (cont.)

- In other scenarios the clauses involve even loops.

```
% Original program
[...]
accredit_english_level :- english_certificate.
accredit_english_level :- english_native.
accredit_english_level :- english_exam_passed.

english_certificate :- external_certificate.
english_certificate :- english_exam_passed.

english_exam_passed :- onsite_exam_passed.
english_exam_passed :- english_native,
                       not last_exam_failed.

last_exam_failed :- not english_certificate.
```

# (Real) use case: School place allocation (cont.)

- In other scenarios the clauses involve even loops.

```
% Original program
[...]
accredit_english_level :- english_certificate.
accredit_english_level :- english_native.
accredit_english_level :- english_exam_passed.

english_certificate :- external_certificate.
english_certificate :- english_exam_passed.

english_exam_passed :- onsite_exam_passed.
english_exam_passed :- english_native,
                       not last_exam_failed.

last_exam_failed :- not english_certificate.
```

```
% Result after forgetting
% for Student 2
accredit_english_level :-
    not last_exam_failed.
accredit_english_level.

last_exam_failed :- not neg_2.
neg_2 :- not neg_1.
neg_1 :- last_exam_failed.
```

# (Real) use case: School place allocation (cont.)

- Let's see the justifications for `?- accredit_english_level`.

# (Real) use case: School place allocation (cont.)

- Let's see the justifications for `?- accredit_english_level`.

```
% Justification Original program
% for Student 2
accredit_english_level :-
  english_certificate :-
    english_exam_passed :-
      english_native,
        not last_exam_failed :-
          chs(english_certificate).
```

# (Real) use case: School place allocation (cont.)

- Let's see the justifications for `?- accredit_english_level`.

```
% Justification Original program
% for Student 2
accredit_english_level :-
  english_certificate :-
    english_exam_passed :-
      english_native,
        not last_exam_failed :-
          chs(english_certificate).
```

# (Real) use case: School place allocation (cont.)

- Let's see the justifications for `?- accredit_english_level`.

```
% Justification Original program
% for Student 2
accredit_english_level :-
  english_certificate :-
    english_exam_passed :-
      english_native,
        not last_exam_failed :-
          chs(english_certificate).
```

```
% Justification after forgetting
% for Student 2
accredit_english_level :-
  not last_exam_failed :-
  neg_2 :-
    not neg_1 :-
      chs(not last_exam_failed).
```

# (Real) use case: School place allocation (cont.)

- Let's see the justifications for `?- accredit_english_level.`

```
% Justification Original program
% for Student 2
accredit_english_level :-
  english_certificate :-
    english_exam_passed :-
      english_native,
        not last_exam_failed :-
          chs(english_certificate).
```

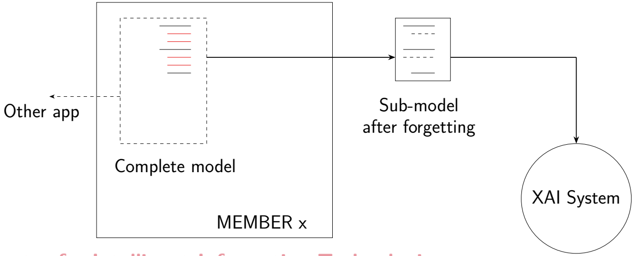
```
% Manipulated justification
% for Student 2
accredit_english_level :-
  not last_exam_failed.
```

```
% Justification after forgetting
% for Student 2
accredit_english_level :-
  not last_exam_failed :-
  neg_2 :-
    not neg_1 :-
      chs(not last_exam_failed).
```

# (Real) use case 2: Energy Assignment

accepted in PAAMS'24

- We propose a value-aware automated decision-making systems for energy assignment in agricultural cooperative:
  - Their decisions must be fair.
  - To trust a decision, a justification is required (XAI).
  - Additionally, the members of the cooperative may want to preserve business secrets (e.g., salary complements).



# (Real) use case 2: Energy Assignment

accepted in PAAMS'24

- We propose a value-aware automated decision-making systems for energy assignment in agricultural cooperative:
  - Their decisions must be fair.
  - To trust a decision, a justification is required (XAI).
  - Additionally, the members of the cooperative may want to preserve business secrets (e.g., salary complements).

```
% Original clauses
salary(eric, Salary):-
  base_salary(eric, S0),
  distance_home_work(eric, S1),
  has_children(eric, S2),
  Salary is S0 + S1 + S2.
```

```
% Result after forgetting
salary(eric, Salary):-
  S0 = 1200,
  S1 = 100,
  S2 = 100,
  Salary is S0 + S1 + S2.
```

# (Real) use case 2: Energy Assignment (cont.)

- Again, let's see how it works in the presence of even loops.

```
% Original clauses
over_40_bea :-
  not neg_over_40_bea.
neg_over_40_bea:-
  not over_40_bea.

generational_renewal(bea, 0):-
  over_40_bea.
generational_renewal(bea, 100):-
  not over_40_bea.

salary(bea, Salary):-
  base_salary(bea, S0),
  generational_renewal(bea, S1),
  holiday_worked(bea, S2),
  Salary is S0 + S1 + S2.
```

# (Real) use case 2: Energy Assignment (cont.)

- Again, let's see how it works in the presence of even loops.

```
% Original clauses
over_40_bea :-
  not neg_over_40_bea.
neg_over_40_bea:-
  not over_40_bea.

generational_renewal(bea, 0):-
  over_40_bea.
generational_renewal(bea, 100):-
  not over_40_bea.

salary(bea, Salary):-
  base_salary(bea, S0),
  generational_renewal(bea, S1),
  holiday_worked(bea, S2),
  Salary is S0 + S1 + S2.
```

# (Real) use case 2: Energy Assignment (cont.)

- Again, let's see how it works in the presence of even loops.

```
% Original clauses
over_40_bea :-
  not neg_over_40_bea.
neg_over_40_bea:-
  not over_40_bea.

generational_renewal(bea, 0):-
  over_40_bea.
generational_renewal(bea, 100):-
  not over_40_bea.

salary(bea, Salary):-
  base_salary(bea, S0),
  generational_renewal(bea, S1),
  holiday_worked(bea, S2),
  Salary is S0 + S1 + S2.
```

```
% Result after forgetting
neg_1 :- not neg_2.
neg_2 :- not neg_1.

salary(bea, Salary):-
  S0 = 900,
  neg_2,
  S1 = 0,
  S2 = 0,
  Salary is S0 + S1 + S2.
salary(bea, Salary):-
  S0 = 900,
  neg_1,
  S1 = 100,
  S2 = 0,
  Salary is S0 + S1 + S2.
```

# Conclusions

- We have presented the design (and implementation) of $f_{CASP}$, an iterative and commutative forgetting technique that:
  - Supports the presence of even and odd loops
    ...we tested its correctness with examples from [5; 8].
  - Could be extended to support predicates and constraints
    ...thanks to the use of dual rules from s(CASP).
- We have applied $f_{CASP}$ to (real) use cases
  ...considering value-aligned by preserving confidentiality and privacy.

# Conclusions

- We have presented the design (and implementation) of $f_{CASP}$, an iterative and commutative forgetting technique that:
  - Supports the presence of even and odd loops
    ...we tested its correctness with examples from [5; 8].
  - Could be extended to support predicates and constraints
    ...thanks to the use of dual rules from s(CASP).
- We have applied $f_{CASP}$ to (real) use cases
  ...considering value-aligned by preserving confidentiality and privacy.

## Future Work

- Provide a formal proof of the $f_{CASP}$ algorithm's correctness.
- Expand $f_{CASP}$ to support generic CASP programs.

## Conclusions

- We have presented the design (and implementation) of $f_{CASP}$, an iterative and commutative forgetting technique that:
  - Supports the presence of even and odd loops
    ...we tested its correctness with examples from [5; 8].
  - Could be extended to support predicates and constraints
    ...thanks to the use of dual rules from s(CASP).
- We have applied $f_{CASP}$ to (real) use cases
  ...considering value-aligned by preserving confidentiality and privacy.

## Future Work

- Provide a formal proof of the $f_{CASP}$ algorithm's correctness.
- Expand $f_{CASP}$ to support generic CASP programs.

THANKS!

# Bibliography I


[1] Arias, Joaquín, Carro, Manuel, Chen, Zhuo, and Gupta, Gopal (2020). **Justifications for Goal-Directed Constraint Answer Set Programming**. In: *Proceedings 36th International Conference on Logic Programming (Technical Communications)*. Vol. 325. EPTCS. Open Publishing Association, pp. 59–72. DOI: 10.4204/EPTCS.325.12.

[2] — (2022). **Modeling and Reasoning in Event Calculus using Goal-Directed Constraint Answer Set Programming**. In: *Theory and Practice of Logic Programming* 22.1, pp. 51–80. DOI: 10.1017/S1471068421000156.

[3] Arias, Joaquín, Carro, Manuel, Salazar, Elmer, Marple, Kyle, and Gupta, Gopal (2018). **Constraint Answer Set Programming without Grounding**. In: *Theory and Practice of Logic Programming* 18.3-4, pp. 337–354. DOI: 10.1017/S1471068418000285.

[4] Berthold, Matti (2022). **On Syntactic Forgetting with Strong Persistence**. In: *Proceedings of the International Conference on Principles of Knowledge Representation and Reasoning*. Vol. 19, pp. 43–52.

# Bibliography II


[5] Berthold, Matti, Gonçalves, Ricardo, Knorr, Matthias, and Leite, Joao (2019a). **A Syntactic Operator for Forgetting that satisfies Strong Persistence**. In: *Theory and Practice of Logic Programming* 19.5-6, pp. 1038–1055.

[6] — (2019b). **Forgetting in Answer Set Programming with Anonymous Cycles**. In: *Progress in Artificial Intelligence: 19th Conference on Artificial Intelligence EPIA*. Springer, pp. 552–565. DOI: 10.1007/978-3-030-30244-3_46.

[7] Gelfond, Michael and Lifschitz, Vladimir (1988). **The Stable Model Semantics for Logic Programming**. In: *5th International Conference on Logic Programming*, pp. 1070–1080. DOI: 10.2307/2275201.

[8] Gonçalves, Ricardo, Janhunen, Tomi, Knorr, Matthias, and Leite, João (2021). **On Syntactic Forgetting under Uniform Equivalence**. In: *European Conference on Logics in Artificial Intelligence*. Springer, pp. 297–312.

# Bibliography III


[9] Gonçalves, Ricardo, Knorr, Matthias, and Leite, Joao (2016). **You can't always forget what you want: on the limits of forgetting in Answer Set Programming**. In: *Proceedings of the Twenty-second European Conference on Artificial Intelligence*, pp. 957–965.

[10] Gonçalves, Ricardo, Knorr, Matthias, and Leite, João (2023). **Forgetting in Answer Set Programming–A Survey**. In: *Theory and Practice of Logic Programming* 23.1, pp. 111–156.

[11] Knorr, Matthias and Alferes, José Júlio (2014). **Preserving Strong Equivalence while Forgetting**. In: *Logics in Artificial Intelligence: 14th European Conference, JELIA 2014*. Springer, pp. 412–425. DOI: 10.1007/978-3-319-11558-0_29.

[12] Lifschitz, Vladimir, Tang, Lappoon R, and Turner, Hudson (1999). **Nested expressions in logic programs**. In: *Annals of Mathematics and Artificial Intelligence* 25, pp. 369–389. DOI: 10.1023/A:1018978005636.