Universidad Rey Juan Carlos

CETINIA
Centre for Intelligent Information Technologies

Value Awareness Engineering

Subproject 3: Value-aware Systems

Forgetting what we want to forget

Luciana Fidilio Allende

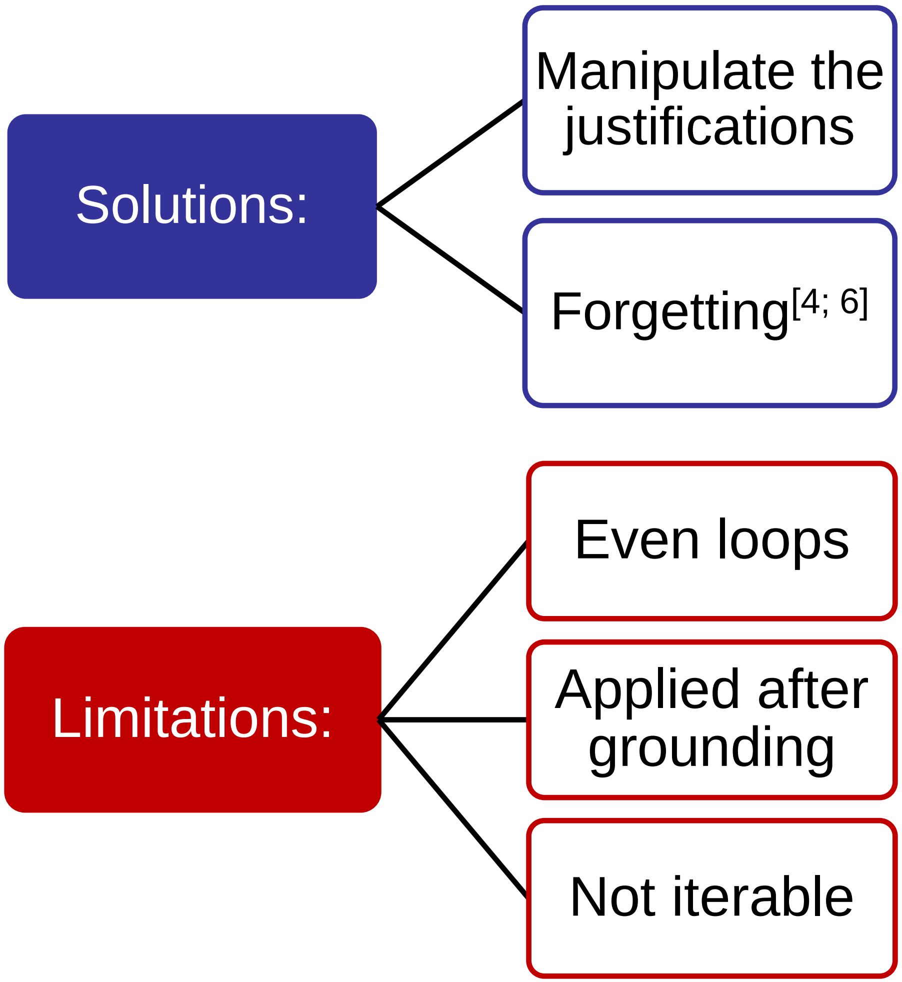
Introduction

- AI systems, even if they are value-aware, can only be trustworthy if they are **capable of explaining** the reasons for their decisions.

  - s(LAW) [3], based on s(CASP) [2] under the ASP semantics [5], **provides justifications** (in NL) [1] and its models can be audited.

- However, the justifications (and the ASP models themself) may **expose sensitive information** (violating privacy and/or legacy)

Introduction



Our proposal:
a forgetting framework based on s(CASP) **dual rules**

It can be used to:

Hide data on victims of
gender violence
in allocation of school places.

Remove predicates in learned
Inductive Logic Programs.

Avoid **reputational damage** due to
security incident reports.

Background: s(CASP)

- **Top-down** execution of Constraint Answer Set Programming.
- Support of even loops due to **non-stratified negation**. E.g.:

```
1. p :- not q.
2. q :- not p.
```

```
% Generates two models
{p, not q} {q, not p}
```

- Compile dual rules and denials:
  - **Dual rules**: the negation of the rules in the original program
  - **Denials**: specify that certain literals cannot hold simultaneously.
    - User-defined: E.g., a man cannot stand and sit at same time.

      ```
      :- stand(Man, Time), sit(Man, Time).
      ```

    - Odd loops: E.g., to avoid inconsistency due to rules of the form `p:- q, not p`, the compiler adds the denial `:- q, not p`.

      ...discarding models where q and not p are true simultaneously.

Background: Forgetting in ASP

- It is a technique that deletes one or more clauses from a program without affecting its semantic.
- Proposals:
  - From Lisbon F_{SU} : [Gonçalves et al. 2021]
    - Simple, iterable, satisfies uniform persistence.
    - Disadvantages: Loss of information, it only works over stratified programs (it does not support even loops).
  - From Leipzig+Lisbon: F^*_{SP} : [Berthold et al. 2019]
    - Iterable.
    - Disadvantages: Satisfies strong persistence only if the original program has certain properties.

Our proposal: Based on F_{SU}

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

Our proposal: Forgetting based on s(CASP) dual rules

- The forgetting implementation is divided in 4 major steps:

1. Transform the clauses of the predicate to forget

a. Double negations

```
1. % Generates two models {q} {}
2. q :- not not q.
```

→

```
1. q :- not neg_a.
2. neg_a :- not q.
```

b. Facts and missing predicates

```
1. p.        % p :- true.
2. q :- t.   % t :- false.
```

2. Generate s(CASP) dual rules of the predicate to forget

```
1. % Clauses of q/0
2. q :- r, not t.
3. q :- not neg_a.
```

```
1. % s(CASP) dual rules
2. not q :- not q_1, not q_2.
3. not q_1 :- not r.
4. not q_1 :- t.
5. not q_2 :- neg_a.
```

→

```
1. % Simplified dual program
2. not q :- not r, neg_a.
3. not q :- t, neg_a.
```

3. Forget a predicate (and its negation) and restore even loops.

4. (Optional) Transform double negations into even loops

Our proposal: Forgetting based on s(CASP) dual rules

The following program:

1. p :- not q.
2. q :- r, not t.
3. q :- not not q.
4. r :- not s.
5. s :- p.

… has two models {p, s}
and {q, r},

Our proposal: Forgetting based on s(CASP) dual rules

The following program:

1. p :- not q.
2. q :- r, not t.
3. q :- not not q.
4. r :- not s.
5. s :- p.

… has two models {p, s} and {q, r},

and after forgetting p and q, the program:

1. neg_a :- not r, not not neg_a.
2. neg_a :- t, not not neg_a.
3. r :- not s.
4. s :- not not neg_a.

… has the models {s} and {r} (without p and q).

Our proposal: Forgetting based on s(CASP) dual rules

Step 1: Transform the clauses of the predicate to forget (p)

Original program:

1. p :- not q.
2. q :- r, not t.
3. q :- not not q.
4. r :- not s.
5. s :- p.

Program after step 1:

1. p :- not q.
2. q :- r, not t.
3. q :- not not q.
4. r :- not s.
5. s :- p.

In this case, no change is needed.

Our proposal: Forgetting based on s(CASP) dual rules

Step 2: Generate s(CASP) dual rules for predicate p.

Program after step 1:

```
1. p :- not q.
2. q :- r, not t.
3. q :- not not q.
4. r :- not s.
5. s :- p.
```

Program after step 2:

```
1. p :- not q.
2. q :- r, not t.
3. q :- not not q.
4. r :- not s.
5. s :- p.
6. not p :- q. % HERE
```

We generate the dual rule for p.

Our proposal: Forgetting based on s(CASP) dual rules

Step 3: Forget the “sensitive” predicates (and their negation) and restore even loops.

Program after step 2:

```
1. p :- not q.
2. q :- r, not t.
3. q :- not not q.
4. r :- not s.
5. s :- p. % HERE
6. not p :- q.
```

Program after step 3:

```
1. p :- not q.
2. q :- r, not t.
3. q :- not not q.
4. r :- not s.
5. s :- not q. % HERE
6. not p :- q.
```

```
1.
2. q :- r, not t.
3. q :- not not q.
4. r :- not s.
5. s :- not q.
6.
```

Our proposal: Forgetting based on s(CASP) dual rules

Step 1: Transform the clauses of the predicate to forget (q)

Program after forgetting p:

```
1. q :- r, not t.
2. q :- not not q. % HERE
3. r :- not s.
4. s :- not q.        % HERE
```

Program after step 1:

```
1. q :- r, not t.
2. q :- not neg_a  % HERE
3. neg_a :- not q. % HERE
3. r :- not s.
4. s :- neg_a.        % HERE
```

The term ‘not not’ in ASP is equivalent to the even loops in s(CASP).

Our proposal: Forgetting based on s(CASP) dual rules

Step 2: Generate s(CASP) dual rules for predicate q.

Program after step 1:

```
1. q :- r, not t.
2. q :- not neg_a
3. neg_a :- not q.
3. r :- not s.
4. s :- neg_a.
```

Program after generating duals:

```
1. q :- r, not t.
2. q :- not neg_a
3. neg_a :- not q.
3. r :- not s.
4. s :- neg_a.
5. not q :- not q_1, not q_2. % HERE
6. not q_1 :- not r, neg_a.     % HERE
7. not q_2 :- t, neg_a.         % HERE
```

Our proposal: Forgetting based on s(CASP) dual rules

Step 2: Generate s(CASP) dual rules for predicate q.

Program after generating duals:

```
1. q :- r, not t.
2. q :- not neg_a
3. neg_a :- not q.
3. r :- not s.
4. s :- neg_a.
5. not q :- not q_1, not q_2. % HERE
6. not q_1 :- not r, neg_a.    % HERE
7. not q_2 :- t, neg_a.        % HERE
```

Program after step 2:

```
1. q :- r, not t.
2. q :- not neg_a
3. neg_a :- not q.
3. r :- not s.
4. s :- neg_a.
5.
6. not q :- not r, neg_a. % HERE
7. not q :- t, neg_a.     % HERE
```

Our proposal: Forgetting based on s(CASP) dual rules

Step 3: Forget the “sensitive” predicates (and their negation) and restore even loops.

Program after step 2:

```
1. q :- r, not t.
2. q :- not neg_a
3. neg_a :- not q.
3. r :- not s.
4. s :- neg_a.
5. not q :- not r, neg_a.
6. not q :- t, neg_a.
```

Program after the transformation:

```
1. q :- r, not t.
2. q :- not neg_a
3. neg_a :- not r, neg_a. % HERE
4. neg_a :- t, neg_a.     % HERE
3. r :- not s.
4. s :- neg_a.
5. not q :- not r, neg_a.
6. not q :- t, neg_a.
```

```
1.
2.
3. neg_a :- not r, neg_a.
4. neg_a :- t, neg_a.
3. r :- not s.
4. s :- neg_a.
5.
6.
```

Our proposal: Forgetting based on s(CASP) dual rules

Step 3: Forget the “sensitive” predicates (and their negation) and restore even loops.

Program after the last transformation:

```
1. neg_a :- not r, neg_a.
2. neg_a :- t, neg_a.
3. r :- not s.
4. s :- neg_a.
```

Program after step 3:

```
1. neg_a :- not r, not not neg_a. % HERE
2. neg_a :- t, not not neg_a.     % HERE
3. r :- not s.
4. s :- not not neg_a.            % HERE
```

Our proposal: Forgetting based on s(CASP) dual rules

Step 4: Transform double negations into even loops

Program after step 3:

```
1. neg_a :- not r, not not neg_a.
2. neg_a :- t, not not neg_a.
3. r :- not s.
4. s :- not not neg_a.
```

Program after forgetting p and q, step 4.

```
1. neg_a :- not r, not neg_b. % HERE
2. neg_b :- not neg_a.
3. neg_a :- t, not neg_b.       % HERE
4. r :- not s.
5. s :- not neg_b.              % HERE
```

This transformation allows us to execute
the resulting program under s(CASP)

Evaluation I: Justification

Justifications for query ? - s.

Program after step 1

Model: {s, p}

```
s :-
  p :-
   not q :-
     not r :-
       chs(s).
     neg_a :-
       chs(not q).
```

Forgetting p and q

Model: {s}

```
s :-
  not neg_b :-
    neg_a :-
      not r :-
        chs(s).
      chs(not neg_b).
```

Evaluation I: Justification

Justifications for query ? - s.

Program after step 1

Model: {s, p}

```
s :-
  p :-
   not q :-
     not r :-
       chs(s).
     neg_a :-
       chs(not q).
```

Forgetting p and q

Model: {s}

```
s :-
  not neg_b :-
     neg_a :-
        not r :-
           chs(s).
        chs(not neg_b).
```

Manipulate using --short

Model: {s}

```
s  :-
  not r :-
    chs(s).
  neg_a.
```

Evaluation II: Justification

Justifications for query ? - r.

Program after step 1

Model: {r, q}

```
r :- % first answer
  not s :-
    not p :-
      q :-
        chs(r),
        not t.
r :- % second answer
  not s :-
     not p :-
       q :-
         not neg_a :-
         chs(q).
```

Forgetting p and q

Model: {r}

```
r :-
  not s :-
    neg_b :-
      not neg_a :-
          chs(r),
          not t.
```

Evaluation II: Justification

Justifications for query ? - r.

Program after step 1

Model: {r, q}

```
r :- % first answer
  not s :-
    not p :-
      q :-
        chs(r),
        not t.
r :- % second answer
  not s :-
    not p :-
      q :-
        not neg_a :-
        chs(q).
```

Forgetting p and q

Model: {r}

```
r :-
  not s :-
    neg_b :-
      not neg_a :-
        chs(r),
        not t.
```

Manipulate using –short

Model: {r}

```
r :- % first answer
  not s :-
    chs(r),
    not t.
r :- % second answer
  not s :-
    not neg_a.
```

Application of Forgetting in ILP: Models

Given a school places allocation database, the algorithm FOLD-R++ learns:

```
1. obtain_p(yes) :- large_f(yes), not ab3, not ab1.
2. ab1 :- come_non_b(yes), want_b_s(yes), not b1_c(yes).
3. ab2 :- same_education_d(yes), not ab1.
4. ab3 :- not sibling_enroll_c(yes), not ab2.
```

After forgetting the predicates ab1, ab2 and ab3, we obtain:

```
1. obtain_p(yes) :- large_f(yes), sibling_enroll_c(yes), not come_non_b(yes).
2. obtain_p(yes) :- large_f(yes), sibling_enroll_c(yes), not want_b_s(yes).
3. obtain_p(yes) :- large_f(yes), sibling_enroll_c(yes), b1_c(yes).
4. obtain_p(yes) :- large_f(yes), same_education_d(yes), not come_non_b(yes).
5. obtain_p(yes) :- large_f(yes), same_education_d(yes), not want_b_s(yes).
6. obtain_p(yes) :- large_f(yes), same_education_d(yes), b1_c(yes).
7. obtain_p(yes) :- large_f(yes), same_education_d(yes), not come_non_b(yes), b1_c(yes).
8. obtain_p(yes) :- large_f(yes), same_education_d(yes), not want_b_s(yes), not come_non_b(yes).
9. obtain_p(yes) :- large_f(yes), same_education_d(yes), b1_c(yes), not want_b_s(yes).
```

But rules in lines 7 and 8 are subsumed by rule 4,

…and rule in line 9 is subsumed by rule 5.

Application of Forgetting in ILP: Models

Given a school places allocation database, the algorithm FOLD-R++ learns:

```
1. obtain_p(yes) :- large_f(yes), not ab3, not ab1.
2. ab1 :- come_non_b(yes), want_b_s(yes), not b1_c(yes).
3. ab2 :- same_education_d(yes), not ab1.
4. ab3 :- not sibling_enroll_c(yes), not ab2.
```

After removing lines 7, 8 and 9:

```
1. obtain_p(yes) :- large_f(yes), sibling_enroll_c(yes), not come_non_b(yes).
2. obtain_p(yes) :- large_f(yes), sibling_enroll_c(yes), not want_b_s(yes).
3. obtain_p(yes) :- large_f(yes), sibling_enroll_c(yes), b1_c(yes).
4. obtain_p(yes) :- large_f(yes), same_education_d(yes), not come_non_b(yes).
5. obtain_p(yes) :- large_f(yes), same_education_d(yes), not want_b_s(yes).
6. obtain_p(yes) :- large_f(yes), same_education_d(yes), b1_c(yes).
7.
8.
9.
```

We obtain a program that is easier to understand.

Application of Forgetting in ILP: Justifications

For student with data: `large_f(yes), same_education_d(yes), come_non_b(no), sibling_enroll_c(no), want_b_s(yes)` and `b1_c(no),` we obtain:

Learned program:

```
obtain_p(yes) :-
   large_f(yes),
   not ab3 :-
      not siblig_enroll_c(yes),
      ab2 :-
         same_education_d(yes),
         not ab1 :-
            not come_non_b(yes).
         proved(not ab1).
```

Forgetting `ab1`, `ab2` and `ab3`.

```
obtain_p(yes) :-
   large_f(yes),
   same_education_d(yes),
   not come_non_b(yes).
```

Application of Forgetting in ILP: Justifications

For student with data: `large_f(yes), same_education_d(yes), come_non_b(no), sibling_enroll_c(no), want_b_s(yes)` and `b1_c(no),` we obtain:

Learned program:

```
obtain_p(yes) :-
   large_f(yes),
   not ab3 :-
      not siblig_enroll_c(yes),
      ab2 :-
         same_education_d(yes),
         not ab1 :-
            not come_non_b(yes).
         proved(not ab1).
```

Forgetting `ab1`, `ab2` and `ab3`.

```
obtain_p(yes) :-
   large_f(yes),
   same_education_d(yes),
   not come_non_b(yes).
```

Manipulate using `–short`

```
obtain_p(yes) :-
   large_f(yes),
   not siblig_enroll_c(yes),
   same_education_d(yes),
   not come_non_b(yes).
```

Conclusions

- We have presented an algorithm of forgetting that:
  - Removes predicates of a program without affecting its semantics.
    … we tested its correctness with several examples (including those in [4; 6]).
  - Generates programs with reduced explanations and…
    … more value-aligned w.r.t. confidentiality and privacy of the sensitive data.

Conclusions

- We have presented an algorithm of forgetting that:
  - Removes predicates of a program without affecting its semantics.
    … we tested its correctness with several examples (including those in [4; 6]).
  - Generates programs with reduced explanations and…
    … more value-aligned w.r.t. confidentiality and privacy of the sensitive data.

Future Work

- Write a formal proof of the algorithm's correctness, i.e., demonstrate that the program obtained after the transformation is equivalent to the original.
- Fully implement the use case for the allocation of school places
  … preserving the privacy of victims of gender-based violence.
- Apply this framework in other fields:
  - **Cybersecurity**: Obfuscate software models and avoid reputational damage In security reports.
  - **Energy Infrastructures**: Limit data exposure of critical infrastructures.
- Investigate counter-offensives for the application of forgetting to hide bias in decision-making algorithms.

Bibliography I

[1] Arias, J. Carro, M. Chen, Z. and Gupta, G. (2020). Justifications for Goal-Directed Constraint Answer Set Programming. *In: Proceedings 36th ICLP (TC). Vol. 325. EPTCS*, pp. 59-72. *DOI: 20.4204/EPTCS.325.12*

[2] Arias, J., Carro, M., Salazar, E., Marple, K., & Gupta, G. (2018). Constraint answer set programming without grounding. *Theory and Practice of Logic Programming*, *18*(3-4), 337-354.

[3] Arias, J., Moreno-Rebato, M., Rodriguez-García, J. A., & Ossowski, S. (2023). Automated legal reasoning with discretion to act using s (LAW). *Artificial Intelligence and Law*, 1-24.

[4] Berthold, M., Gonçalves, R., Knorr, M., & Leite, J. (2019). A syntactic operator for forgetting that satisfies strong persistence. *Theory and Practice of Logic Programming*, *19*(5-6), 1038-1055.

Bibliography II

[5] Gelfond, M., & Lifschitz, V. (1988, August). The stable model semantics for logic programming. In *ICLP/SLP* (Vol. 88, pp. 1070-1080).

[6] Gonçalves, R., Janhunen, T., Knorr, M., & Leite, J. (2021, May). On syntactic forgetting under uniform equivalence. In *European Conference on Logics in Artificial Intelligence* (pp. 297-312). Cham: Springer International Publishing.

[7] Wang, H., & Gupta, G. (2022, May). FOLD-R++: a scalable toolset for automated inductive learning of default theories from mixed data. In *International Symposium on Functional and Logic Programming* (pp. 224-242). Cham: Springer International Publishing.