Universidad Internacional de La Rioja
Escuela Superior de Ingeniería y Tecnología
Universidad de Alcalá
Escuela de Posgrado
Universidad Rey Juan Carlos
Escuela de Másteres oficiales

Máster Universitario en Inteligencia Artificial para el Sector de la Energía y las Infraestructuras

f_{CASP}: a forgetting operator and its application to energy distribution under a goal-directed ASP decision model

Autor/a: **Fidilio Allende, Luciana Camila**

Tipo de trabajo: Desarrollo software

Directores: Ossowski, Sascha
Arias, Joaquín

Fecha September 11, 2024

Abstract

Decision systems based on Artificial Intelligence not only streamline the resolution of complex problems but can also generate more effective responses. As these responses can affect humans, they must be aligned with human values such as fairness. For instance, in a cooperative/competitive context where they allocate crucial resources, they must provide not only effective but also fair decisions. But, to ensure that their decisions are trustworthy and value-aligned, they must be able to explain their decisions. However, such explanations and the model itself may expose sensitive information from the users or third parties, violating their privacy or, in some cases, the legislation. To filter sensitive data, rule-based systems, such as those based on Answer Set Programming (ASP), offer the possibility of manipulating their justifications. Yet, the information can still be leaked from the rules contained in their models. To remove the information directly from these models, forgetting operators can be used. However, these operators have limitations in their practical applications, they can only be applied in propositional programs, or they are not implemented.

In this work, we propose the design, implementation, and application of f_{CASP}, a new forgetting operator implemented over s(CASP), a goal-directed ASP reasoner, and its dual rules. Through several examples extracted from the literature, we show that f_{CASP} not only fulfills the main features of existing forgetting operators but also, being based on s(CASP) dual rules, we believe it could be extended to support generic ASP programs with constraints.

Finally, to validate the effectiveness and practicality of f_{CASP}, we defined two decision models in the context of an agricultural cooperative that generates and distributes local renewable energy: (i) a propositional model where f_{CASP} is used to forget sensitive information in the process of accepting or denying the application of a member to join the energy generation plan, and (ii) a first-order logic model that assigns the distribution of the energy generated based on human values, such as fair income from the Strategic Plan of the Common Agricultural Policy, and where f_{CASP} is used to hide business secrets.

Acknowledgments

This work summarizes the work done in the last year, which would not have been possible without the invaluable help of my directors, Joaquín and Sascha, who not only proposed this very interesting line of research but also have tirelessly helped and motivated me to pursue it.

I would also like to thank my co-workers, who, although working in very different fields, have listened to my ideas, have been interested in them, and have offered me advice and points of view that I could not have come across in any other way.

Last but not least, I would like to thank my family and friends, who have been there to encourage me to continue, showing me their support all this time.

To all of you, thank you very much for everything.

Resumen

Los modelos de decision basados en Inteligencia Artificial permiten no solo agilizar la resolución de problemas complejos, sino que también pueden facilitar respuestas más justas y efectivas. Como estas respuestas pueden afectar a las personas, es importante que estén en consonancia con valores humanos. Por ejemplo, en un contexto cooperativo/competitivo en el que se asignan recursos clave, se debe generar decisiones no sólo eficaces sino también justas. Pero, para garantizar que sus decisiones son dignas de confianza y se ajustan a los valores, los sistemas deben poder ofrecer una explicación de sus decisiones. Sin embargo, estas explicaciones y el propio modelo pueden exponer información sensible de los usuarios o de terceros, violando su privacidad o la legislación vigente. Para esconder dicha información, los sistemas basados en reglas, como los basados en Answer Set Programming (ASP), ofrecen la posibilidad de manipular sus justificaciones. Sin embargo, dicha información puede seguir filtrándose a partir de sus modelos. Para eliminar la información directamente de estos, se pueden utilizar operadores de forgetting. Sin embargo, estos operadores tienen limitaciones prácticas, sólo pueden aplicarse en programas proposicionales, o no están implementados.

En este trabajo proponemos el diseño, implementación y aplicación de f_{CASP}, un nuevo operador de forgetting implementado sobre s(CASP), un razonador ASP goal-directed, y basado en sus reglas duales. A través de varios ejemplos extraídos de la literatura, mostramos que fCASP no sólo cumple las principales características de los operadores de olvido existentes, sino que además, al estar basado en reglas duales de s(CASP), creemos que podría extenderse para soportar programas ASP genéricos con restricciones.

Para validar la efectividad y practicidad de f_{CASP}, definimos dos modelos de decisión en el contexto de una cooperativa agrícola que genera localmente (y distribuye) energía renovable: (i) un modelo proposicional en el que f_{CASP} se utiliza para olvidar información sensible en el proceso de aceptación o denegación de la solicitud de adhesión de un socio al plan de generación de energía, y (ii) un modelo usando lógica de primer orden que asigna la distribución de la energía generada en función de valores humanos, como los ingresos justos del Plan Estratégico de la Política Agrícola Común, y en el que f_{CASP} se utiliza para ocultar secretos de negocio.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Decision systems have rapidly evolved in recent years thanks to the application of various Artificial Intelligence techniques. In particular, there has been an improvement in the effectiveness of the answers given by the models thanks to the advances in the sector.

More recently, research and debate on the ethics applied to Artificial Intelligence have gained momentum, as the decisions generated by the models may adversely affect the fundamental rights and safety of individuals, even if the system was well-intentioned. For this motive, whether through self-regulation and soft law (guidelines, codes of conduct, declarations, ethical charters, etc.) or legal regulation (e.g., the General Data Protection Regulation (GDPR) or the Regulation on AI, both by the EU), promoting a reliable AI, focused on humans, is of foremost importance.

To ensure that the models' responses are not only effective, but that they are also fair, it's necessary that a series of ethical values are taken into account in their development, and that the dataset (if any) used to infer the model is not biased, that is, factors such gender or race (or derived factors) are only relevant to the decision if they have a fair explanation for their impact. For example, gender should not be a relevant factor in the hiring process for warehouse positions, but it can be a crucial factor in pharmaceutical studies as some chemicals may have different effects depending on it.

In order to audit a model and verify that it is aligned to a set of ethical values, verifying both its development and the possible dataset used, the model must be transparent and understandable, being explainability a requirement marked by the Regulation on AI of the European Union. In particular, value-aligned (Montes et al., 2023) systems must be capable of explaining the models and justifying decisions taken in a human-understandable manner, in terms of the values and norms that influenced the reasoning process, among others. These

justifications make it possible for these users can understand the terms and norms used for generating the justification, generating in turn a stronger confidence in the model.

However, the explanations that allow an understanding of the logic behind a model's decision, with the aim of increasing the transparency of the model, can also expose sensitive user information if this data is used to generate a response.

Some proposals based on Answer Set Programming (ASP), thanks to their rule-based nature, are able to provide explanations for their decisions. A notable system with these characteristics is s(CASP) (Arias et al. 2018), a goal-directed Answer Set Programming (ASP) reasoner, which provides justifications for its decisions. Furthermore, these justifications can be presented in natural language (Arias et al. 2020). In addition, it allows the manipulation of justifications to create a semantically equivalent justification with hidden sensitive atoms. Yet, even if the justifications have been manipulated, the model still can expose sensitive information. This fact poses a challenge when the model must be audited. To remove sensitive information effectively, it is necessary to apply forgetting, a technique that removes predicates from a program with minimal changes in its behavior and answers.

Another promising approach to hide sensitive information could be the cryptographic technique of Zero-Knowledge Proofs (Goldreich and Oren, 1994), where the model would explain/prove that the decision is based on certain information without revealing the information or any aspect of it, but this line of research is outside the scope of this work.

Our proposal, based on s(CASP), is a forgetting technique called f_{CASP} that supports the presence of even loops in programs with non-stratified negations and can be applied to Constraint Answer Set Programs. In particular, we plan to manipulate s(CASP) programs to provide models and justifications forgetting sensitive information in several scenarios.

To validate the practical application of f_{CASP}, we propose a (real) use case that models the requirements for qualifying for membership in an energy community within an agricultural cooperative, being the purpose of this community the generation of energy to address the instability of energy supply in rural areas while benefiting the ecological environment. Furthermore, the distribution of energy is also carried by an ASP decision system, which allocates the energy by awarding points considering human values (in this case, fairness). It's important to note that both models will involve private or confidential information, which will be required to be forgotten.

1.1 Objectives

The goal of this work can be divided into two objectives:

The first one is to design, model, and implement a suitable technique that can hide sensitive information from a program and its justifications without affecting its explainability.

The second one is creating decision models based on human values to automate the acceptance of a candidate to a renewal energy community and the distribution of the energy that it generates, digitalizing the agricultural sector through the use of Artificial Intelligence, and promoting the use of rule-based models.

1.1.1 Implementation of a suitable forgetting technique

The answers of rule-based programs can be explained with the help of certain reasoners, such as s(CASP) (Arias et al. 2018), detailed in Chapter 2 along with the background theory related to this work, i.e., Answer Set Programming (Section 2.1), s(CASP) (Section 2.2) and the properties and techniques related with the concept of forgetting (Section 2.3). However, even if these explanations can bring transparency to the program, they may also reveal sensitive information. To affront this challenge, in this work we propose extending the functionality of s(CASP) to preserve the privacy of the programs while maintaining their explainability. With this approach in mind, the first objective is to design and implement a technique capable of deleting sensitive information in a program, while minimizing the impact of this modification on the program, thus conserving its explainability and preserving its original answers.

To achieve this, we studied the techniques related to forgetting, which focused precisely on the preservation of semantic information while removing predicates from a program. Nonetheless, as we comment in Section 2.3 where we discuss the properties of the most relevant forgetting techniques, while researching forgetting operators, we found out that most of them were not implemented and were focused on programs without variables or (arithmetical) constraints. The only implemented operator found (Calimeri et al. 2024) was applied after grounding, so the program needed to be executed to apply the operator. Furthermore, this operator was not aimed to preserve the privacy of original programs but as a complement to simplify the program obtained while over-grounding. Additionally, some of the theoretical operators could not be applied to forget multiple predicates, or were limited in other manner on the grade of equivalence obtained or on the properties a program needed for the operator to be applied on.

Some restrictions could not be resolved, as some level of equivalence cannot be obtained for every program without using additional predicates (Gonçalves, Knorr, and Leite 2016).

However, we wanted to focus not on this theoretical challenge, but on achieving an implemented operator that could generate a program that could output the same answers as the original (omitting the sensitive information) and did not expose private or confidential data on the model or its justifications. In that matter, we focused on the practicality of the algorithm, but not without trying to reduce the number of additional predicates needed.

To satisfy this objective, in this work we have defined the forgetting operator f_{CASP} in Chapter 3., given more details about its design and its implementation in Section 3.1 , and its use as part of s(CASP) in Section 3.2. This forgetting operator can delete sensitive information in both an ASP program itself and its justifications. As for the capacities of this operator, it needs to be capable of forgetting multiple predicates in a commutative way, even if they are present in loops or double negations. The specific objectives of f_{CASP} here defined are evaluated in Section 3.3 by solving flagship examples of the literature.

1.1.2 Modeling value-aware decision systems for locally generated energy management

The second objective of this work is to control the management of energy through the use of two value-aware decision systems, one for the acceptance of a candidate to the energy association, and one for the distribution of the generated energy. The use of these systems paves the way for the digitalizing of the agricultural sector through the use of Artificial Intelligence, and the modeling of fair systems, and promotes the use of rule-based models. The motivation and general reasoning behind the models are contemplated in Chapter 4, while the specific details behind the criteria chosen and their modeling are described in Chapter 5.

Model explainability is essential for building trust in its decisions while filtering sensitive data is crucial to protect the privacy and confidentiality of farmers and their businesses. To effectively protect sensitive data, it is necessary to remove that information from the model itself, that is, to forget that information, as it is detailed in Section 4.1.

The forgetting operator f_{CASP} helps to maintain the balance between the right to privacy and the right to explanation, a conflict more detailed in the first use case, a propositional system to automatically decide if a candidate may or not (or it is uncertain) join the community formed to generate energy locally using renewable sources. This use case is more detailed in Section 4.2, while the details for its implementation are contemplated in Section 5.1.

To apply f_{CASP} to energy distribution, a second use case (with variables) has been defined where the membership to a community of energy generation, and its distribution, are determined by a number of points awarded to farmers. As detailed in Section 4.3, these points are assigned considering the objectives for improved agricultural exploitation defined in the Strategic Plan of the Common Agricultural Policy, which considers values such as equity and environmental consciousness. Section 5.2 details the specific modeling of these criteria. As f_{CASP} does not support programs with variables yet, the application of the operator to this program is theoretical.

Finally, in Chapter 6 it is shown that the results of evaluating the performance of f_{CASP} using the flagship examples of the literature and the two use cases proposed are promising, and satisfy the objectives described as we describe in Section 6.1. In addition, in Section 6.2 we have detailed the future lines of work identified to improve our proposal.

1.2 Thesis contributions and impact

In this Section, the contributions of this work to the industry are detailed, along with the publications that have stemmed from these contributions.

As outlined in Section 1.1, this work describes the design and implementation of a new forgetting operator capable of hiding information in propositional Answer Set Programs. It is anticipated that this operator can be extended to support variables and (algorithmic) constraints as well in the future. The first insights of this operator were presented in a preliminary work (Fidilio-Allende and Arias 2024) published under the PROgramación y LEnguajes (PROLE) conference, hosted by the SISTEDES society.

Then, to check the practical use of this operator, two use cases were defined: the first involved a propositional decision system for the filtering of applicants, submitted to the 40th International Conference on Logic Programming (ICLP). The second use case focused on a decision system fair distribution of energy in agricultural cooperatives. This last use case was accepted in the Workshop on Adaptive Smart areaS and Intelligent Agents (ASSIA) under the Conference of Practical Applications of Agents & Multi-Agent Systems (PAAMS). It incorporates variables and recreates a simulation of how forgetting would be applied in these programs focusing on its practical implications.

1.3 Thesis Organization

Chapter 1 expands on the motivation behind this work and the general and specific objectives marked to value the effectivity of f_{CASP} are specified.

Chapter 2 provides a theoretical background of ASP programs and s(CASP), and includes the current state of the art, including forgetting operators and the more important properties to describe its capacities.

Chapter 3 focuses on the implementation, use, and capacities of f_{CASP}.

Chapter 4 details the motivations for the decision systems and their criteria for accepting the application of a farmer to join the energy community, and for the energy allocation.

Chapter 5 models the criteria chosen, presenting the implementation of the decision systems that constitute the use case. Then, this Chapter evaluates the application of f_{CASP} to check if the sensitive information is effectively removed from the models and their explanations.

Chapter 6 summarizes the results obtained after the experiments, and proposes future lines of research and improvement.

Chapter 2

Background and related work

2.1 Answer Set Programming

ASP (Gelfond and Lifschitz 1988; Brewka, Eiter, and Truszczyński 2011) is a logic programming paradigm suited for knowledge representation and reasoning (Brewka, Eiter, and Truszczyński 2011). ASP also facilitates commonsense reasoning (Gupta 2022). An ASP program P is a finite set of *rules*. Each rule $r \in P$ is of the form:

$$\texttt{a:- b1,...,b}_m\texttt{,not b}m+1\texttt{,...,not b}_n\texttt{.}$$

where a and $b_1, \ldots, b_n$ are atoms and *not* corresponds to *default* negation. An atom is an expression of form $p(t_1, \ldots, t_n)$ where p is a predicate symbol of arity n and t_i, are *terms*. An atom is *ground* if no variables occur in it. The set of all *constants* appearing in P is denoted by C_P. The head of rule r is $h(r) = \{a\}$ and the body consists of positive atoms $b^+(r) = \{b_1, \ldots, b_m\}$ and negative atoms $b^-(r) = \{b_{m+1}, \ldots, b_n\}$. Intuitively, rule r is a justification to *derive* that a is true if all atoms in $b^+(r)$ have a derivation and no atom in $b^-(r)$ has a derivation. An interpretation I is a subset of the program's Herbrand base and it is said to satisfy a rule r if $h(r)$ can be derived from I. A model of a set of rules is an interpretation that satisfies each rule in the set.

Under the stable model semantics logic programs with non-stratified negations may generate multiple models. E.g., the following program has two models:

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

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

In the first one, denoted as `{p}`, the atom `p` is true, and `q` is false, while in the other, denoted as `{q}`, `p` is false and `q` is true. These interleaving calls over negation form an *even loop* because there is an even number of intervening negations. If the number were odd, it would result in an *odd loop*, leading to an inconsistency.

A difference between ASP and Prolog-style (i.e., SLD resolution-based) languages is the treatment of negated literals. Negated literals in a body are treated in ASP using their logical semantics based on computing stable models. The *negation as failure* rule of Prolog (i.e., SLDNF resolution (Clark 1978)) makes a negated call succeed (respectively, fail) iff the non-negated call fails (respectively, succeeds). To ensure soundness, SLDNF has to be restricted to ground calls, as a successful negated goal cannot return bindings. However, SLDNF increases the cases of non-termination w.r.t. SLD.

Extended ASP with double default negation (Lifschitz, Tang, and Turner 1999) In this work we consider extended logic programs with clauses of the form `p:- not not p`. This operator, `not not`, is made to represent the concept of not having proof of the fact of not having proof that a predicate is true. This concept generates two possibilities: in one the predicate holds, as the predicate is true and there is proof of that, and the other possibility is that even if there is no proof of not having proof of the predicate being true, the predicate does not hold.

An interesting fact about these programs under the answer set semantics is that two negations as failure operators in a row do not, generally, cancel each other. I.e., the program `p:- not not p` gives two possible models `{}` and `{p}`. But, if we cancel the double negation, the resulting program `p:- p` has only one model `{}`. On the other hand, if we follow (Lifschitz, Tang, and Turner 1999) and "denote" the subformula `not p` by `q` the program turns into the even loop example we shown above, and after dropping the "auxiliary" symbol `q` from both stable models, we will get the expected models `{}` and `{p}`, and the semantic information is conserved as `q` represents not having proved the predicate `p` holds, and `p` represents not having proof of the predicate `q` holding.

Our proposal, f_{CASP}, uses this transformation strategy when identifying atoms that incur in even loops (in both cases, due to double negations and interleaving calls over negation).

It is worth noting that the double negation operator 'not not' is not supported in s(CASP). A possible substitute that preserves the information expressed is an explicit even loop using an auxiliary predicate. This transformation is explained in more detail in Section 3.1.1.

2.2 s(CASP): Goal-directed ASP reasoner

s(CASP) is a top-down, goal-driven interpreter of Constraint ASP programs written in Prolog (`https://gitlab.software.imdea.org/ciao-lang/sCASP`). The top-down evaluation makes the *grounding* phase unnecessary. The execution of an s(CASP) program starts with a *query*, and each answer is the resulting *mgu* of a

successful derivation, its justification, and a (partial) stable model. This partial stable model is a subset of the ASP stable model (Gelfond and Lifschitz 1988) including only the literals necessary to support the query with its output bindings.

s(CASP) has two main differences w.r.t. Prolog: first, s(ASP) resolves negated atoms *not* l_i against *dual rules* of the program, instead of using negation as failure. This makes it possible for a non-ground negated call `not p(X)` to return the results for which the positive call `p(X)` would fail. Second, and very important, the dual program is **not** interpreted under SLD semantics. Instead, different loops, such as even, odd, and positive loops are handled to construct different models, detect inconsistencies and reduce non-termination cases respectively.

It is also important to mention the difference between denials and constraints in s(CASP). In this context, 'denials' refer to rules without head (classic constraints), while the term 'constraints' relates specifically to arithmetic constraints. In the continuation of this work, this naming convention will be utilized.

2.2.1 Dual rules compilation

The dual of a predicate `p/1` is another predicate that returns the `X` such that `p(X)` is not true. To synthesize the dual of a logic program P, we first obtain the Clark's completion (Clark 1978), which assumes that the rules of the program completely capture all possible ways for atomic formulas to be true. Then, we apply De Morgan's laws. Since the forgetting technique we are presenting in this work is based on s(CASP) dual programs, let us include the description from (Arias et al. 2018) (presented at length elsewhere (Arias et al. 2022)).

1. For each literal p/n that appears in the head of a rule, choose a tuple $\overrightarrow{x}$ of n distinct, new variables $x_1, \ldots, x_n$.
2. For each i^{th} rule of p/n of the form $p_i(\overrightarrow{t_i}) \leftarrow B_i$, with $i = 1, \ldots, k$, make a list $\overrightarrow{y_i}$ of all variables that occur in the body B_i but do not occur in the head $p_i(\overrightarrow{t_i})$, add $\exists \overrightarrow{y_i}$ to the body and rename the variables that appear in the head $\overrightarrow{t_i}$ with the tuple $\overrightarrow{x}$, obtained in the previous step, resulting in a predicate representing $\forall \overrightarrow{x}\ (p_i(\overrightarrow{x}) \leftarrow \exists \overrightarrow{y_i}\ B_i)$. Note that $\overrightarrow{x}$ are local, fresh variables. This step captures the standard semantics of Horn clauses.
3. With all these rules and using Clark's completion, we form the sentences:

$$\forall \overrightarrow{x}\ (\ p(\overrightarrow{x}) \quad \longleftrightarrow \quad p_1(\overrightarrow{x}) \vee \cdots \vee p_k(\overrightarrow{x})\ )$$
$$\forall \overrightarrow{x}\ (\ p_i(\overrightarrow{x}) \quad \longleftrightarrow \quad \exists \overrightarrow{y_i}\ (b_{i.1} \wedge \cdots \wedge b_{i.m} \wedge \neg\ b_{i.m+1} \wedge \cdots \wedge \neg\ b_{i.n})\ )$$

4. Their semantically equivalent duals $\neg p/n$, $\neg p_i/n$ are:

$$\forall \overrightarrow{x}\ (\ \neg p(\overrightarrow{x}) \quad \longleftrightarrow \quad \neg(p_1(\overrightarrow{x}) \vee \cdots \vee p_k(\overrightarrow{x}))\ )$$
$$\forall \overrightarrow{x}\ (\ \neg p_i(\overrightarrow{x}) \quad \longleftrightarrow \quad \neg\ \exists \overrightarrow{y}\, i\ (bi.1 \wedge \cdots \wedge b_{i.m} \wedge \neg\ b_{i.m+1} \wedge \cdots \wedge \neg\ b_{i.n})\ )$$

5. Applying De Morgan's laws we obtain:

$$\forall \overrightarrow{x}(\ \neg p(\overrightarrow{x}) \quad \longleftrightarrow \quad \neg p_1(\overrightarrow{x}) \wedge \cdots \wedge \neg p_k(\overrightarrow{x})\ )$$
$$\forall \overrightarrow{x}\ (\neg p_i(\overrightarrow{x}) \quad \longleftrightarrow \quad \forall \overrightarrow{y} i\ (\neg bi.1 \vee \cdots \vee \neg\ b_{i.m} \vee\ b_{i.m+1} \vee \cdots \vee\ b_{i.n})\ )$$

   which generates a definition for $\neg p(\overrightarrow{x})$ and a separate clause with head $\neg p_i(\overrightarrow{x})$ for each positive or negative literal $b_{i.j}$ in the disjunction. Additionally, a construction to implement the universal quantifier introduced in the body of the dual program is available in s(CASP) reasoner.

Definitions for the initially negated literals $\neg b_{i.m+1} \ldots \neg b_{i.n}$ and for each of the *new* negated literals $\neg b_{i.1} \ldots \neg b_{i.m}$ are similarly synthesized. At the end of the chain, unification has to be negated to obtain disequality, e.g., $x = y$ is transformed into $x \neq y$, also handled by s(CASP) interpreter.

To avoid redundant answers in predicate ASP, every clause for a negated literal $\neg l_i$ includes calls to any positive literal l_j with $j < i$. For example, given the rules `p(0)` and `p(X):- q(X),not t(X,Y)`, the resulting dual rules for `p/1` are:

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

where while the clause in line 5 would only need to be `not p2_(X,Y):- t(X,Y)` the literal `q(X)` is included to avoid exploring solutions already provided by `not p2_(X,Y):- not q(X)` Note that for propositional ASP programs, this optimization can be disabled in s(CASP) by passing the flag `-d, --plaindual`.

2.2.2 Explainability: Justification Trees

As s(CASP) models are rule-based models and partially self-explanatory ASP proof trees, it is possible to obtain justifications for their answers. Given a program (that can contain constraints) and a query, it will search for a number of possible answers. Those answers are partial stable models with only the relevant atoms that satisfy the query given, which means: only the atoms needed to (i) satisfy the non-negated atoms, (ii) make the negated atoms can only be false, and (iii) do not make any contradiction true (by satisfying the constraints). For each answer (stable model), s(CASP) is capable of generating the justification tree and the grounding of the relevant variables (Arias et al. 2022).

A justification tree is a factual explanation in which the top-down reasoning (from a general concept to specific ones) to prove why the conditions given by a query are satisfied. For example, given a program with the predicates `p :- not q`, `q :- not r` and `r` and the query `p`, the reasoning would be that `p` is true because `not q` is true or `q` is not true, and that statement is true because `r` is true. In

addition, no contradiction has been triggered, so the global constraint is also satisfied. That explanation, represented as a justification tree, would be:

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

Other systems based on ASP that follow a top-down execution can also trace which rules have been used to obtain the answers more easily. One such system is ErgoAI (https://coherentknowledge.com), based on XSB (Swift and Warren 2012), which generates justification trees for programs with variables. ErgoAI has been applied to analyze streams of financial regulatory and policy compliance in near real-time providing explanations in English that are fully detailed and interactively navigable. However, default negation in ErgoAI is based on the well-founded semantics (Gelder, Ross, and Schlipf 1991) and therefore ErgoAI is not a framework that allows the representation of ambiguity and/or discretion.

To the best of our knowledge, explainable AI techniques for black-box AI tools, most of them based on machine learning, are not able to explain how variation in the input data changes the resulting decision (DARPA 2017). While counterfactual explanations provide local justifications for a specific decision, they are not able to provide a global justification.

2.3 State-of-the-art Forgetting Techniques

Forgetting is a transformation that removes predicates of an ASP program with minimal impact on its behavior and outputs. For example, if we consider a part of a taxonomy from (Gonçalves, Knorr, and Leite 2023), which includes professors, university staff, and persons with assigned properties, and represented in rules as:

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

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

If we were to forget the predicate `ustaff` exists, it should still hold that if `X` is a professor, then `X` is a person. This relationship would be represented as `person(X):- professor(X)`, being that clause the result of forgetting `ustaff` in the proposed taxonomy.

Forgetting operators are the techniques used to make these transformations in a model while preserving as much as possible the output and structure of the original model.

In recent years, research on forgetting operators has intensified due to the importance of this technique in areas such as grounding, stream reasoning, or Explainable Artificial Intelligence (XAI) systems to resolve conflicts caused by inconsistencies in propositional logic, and to update knowledge databases, among other applications of forgetting such as those described in (Eiter and Kern-Isberner 2019) and (Gonçalves, Knorr, and Leite 2023).

Most of the work in the literature focuses on discussing the properties of different forgetting operators (Gonçalves, Knorr, and Leite 2023). These properties are formulated by comparing the original program with the result after applying forgetting, thereby defining the equivalence relationship between them.

- **Weak Equivalence (WE)** (Wong 2009) is a basic relationship that needs to be satisfied: two programs are weak equivalent if their answer sets are the same.
- **Uniform Equivalence (UE)** (Gonçalves et al. 2021) is a more relaxed equivalence: two programs are uniform equivalent if their answer sets are the same, even when adding (explicitly limited) additional facts, being a fact a rule without body.
- **Strong Equivalence (SE)** (Wong 2009) is more restrictive: two programs are strong equivalent if their answer sets are the same, even when adding additional rules.
- **Relativized Strong Equivalence (RSE)** (Woltran 2004) is a relaxed form of (SE) that permits declaring the set of atoms that cannot appear as part of the additional rules.

Based on these equivalence classes the following properties have been defined over the forgetting operators:

- **Consequence persistence (CP)** (Gonçalves, Knorr, and Leite 2016): an operator is consequence persistent when the result of forgetting satisfies (WE) with the original program, provided that the predicates marked for forgetting are removed from both sets of stable models.
- **Uniform persistence (UP)** (Gonçalves et al. 2021): an operator is uniform persistent when the result of forgetting an atom and the original program share the same answer sets (ignoring the forgotten predicates) even when an arbitrary set of facts are added.
- **Strong persistence (SP)** (Knorr and Alferes 2014): is a key property that requires the original program and the result of generated program share the same answer sets even if new rules (not containing forgotten predicates) are added. That is, an operator is strong persistent when the original program and the result of forgetting satisfies (CE) with the original program, i.e., are strong equivalent (SE), modulo the forgotten atoms.

The (SP) property establishes the basis for defining forgetting on classic logic and ASP. However, (Gonçalves, Knorr, and Leite 2016) have proved that the application of forgetting in programs with even loops (due to interleaving even negations or double negations) is not always possible if we want to preserve (SP). Later, they overcome this limitation by adding an auxiliary predicate (see (Berthold et al. 2019) for details).

In Table 2.1 we compare some properties of our proposal w.r.t. the more relevant operators attempting to comply with uniform/strong persistence:

- $\mathbf{f_{SU}}$ by (Gonçalves et al. 2021), complies with (UP), but it does not satisfy (SP), is restricted to propositional programs without constraints, and while it can act over loops, it is restricted to stratified programs, and is not commutative (i.e., the forgotten program differs according to the order in which the atoms are forgotten).
- $\mathbf{f_{SP}}$ by (Gonçalves et al. 2017), satisfies (UP) and (when no additional predicates are needed) also (SP). However, it cannot act effectively over loops or multiple predicates, and it is also restricted to propositional programs without constraints.
- $\mathbf{f^*_{SP}}$ by (Berthold 2022) has the same limitations on satisfying SP as f_{SP}, but it can act over loops and multiple predicates.
- Finally, $\mathbf{f_{AC}}$ by (Berthold et al. 2019), is able to always satisfy (SP), because it uses additional predicates (as our proposal does), and it can act over loops and multiple predicates. This auxiliary predicate is added when compiling the as-dual, i.e. the set of literals that can be used to represent the negated value of a predicate. Specifically, the additional predicate is used as the as-dual of a literal that does not appear in any head. However, it is restricted to propositional programs without constraints.

	(UP)	(SP)	Loops	Commutative	Predicates	Constraints
f_{SU}	Yes	No	Yes	No	No	No
f_{SP}	Yes	Limited	No	No	No	No
f^*_{SP}	Yes	Limited	Yes	Yes	No	No
f_{AC}	Yes	Yes	Yes	Yes	No	No
f_{CASP}	Yes	Yes	Yes	Yes	WiP	WiP

Table 2.1: Comparison of the more relevant forgetting operators vs. f_{CASP}

Chapter 3

f_{CASP}, a Forgetting Technique based on Goal-directed ASP

In this chapter, we explain the intended design of f_{CASP} as an extension of s(CASP), the algorithm that implements this design, how it can invoked using the goal-directed reasoner, and we evaluate its performance against a set of examples of the literature as a first step to evaluate its capacities.

3.1 A simple and iterative Design

In this work, we planned to design an operator that could support a wide range of programs within the scope of those supported by s(CASP). To achieve an acceptable equivalence relationship between the original program and the generated one, it was necessary to consider including additional atoms, as strong persistence can only be achieved (without adding them) on the programs that do not satisfy Ω, that is, the conditions defined in (Gonçalves, Knorr, and Leite 2016).

The current design covers propositional programs with even/odd loops and double negations, and it is planned to be extended in the future to programs with variables and constraints. It consists of 4+1 steps: four of them are performed iteratively with each predicate that must be forgotten, while the last step is optional, and is performed when there are no more predicates to forget.

For example, if we want to forget `p` and `q` in the following program:

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

it would perform steps one to four for predicate `p`, then repeat those steps for predicate `q`, and, optionally, in the end, the fifth step would be performed. Next, we will go through the steps one by one in more detail, considering we want to forget the predicate `p` from the example displayed above.

3.1.1 Step 1: Add auxiliary predicates due to even loops, facts, and/or missing predicate

In the first step, three types of transformations are performed depending on the conditions of the predicate to forget.

Even loops. The first type of transformation will be performed if the predicate is part of an even loop. In that case, an auxiliary predicate `neg_x` will be added, where `x` is a numeric value. As it is the first auxiliary predicate used, `x` will be `1`. All appearances of the negated predicate, i.e., `not p` if the predicate to forget was `p`, are replaced with the auxiliary predicate. Then, a clause of the form `neg_x :- negated predicate` is formed, in this case, the clause `neg_1 :- not p`.

Original program

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

Transformed program

```
1 p :- not q.
2 q :- neg_1.
3 neg_1 :- not p.
```

Facts. If the predicate was a fact, that is, a predicate with a clause without body, then the predicate `true` would be added as the body of that clause, as this clause would always be true.

Original program

```
1 p.
```

Transformed program

```
1 p :- true.
```

Missing predicates. If the predicate was missing, that is, a predicate that appears on the clause of another predicate but does not appear on the head of any clause, then `predicate :- false` would be added as a clause to add value to that predicate, as it could never be true.

Original program

```
1 q :- p.
```

Transformed program

```
1 q :- p.
2 p :- false.
```

Example program. In our example program, predicate `p` is part of an even loop, so the following transformation would be performed:

Original program

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

Program after step 1

```
1 p :- not q.
2 q :- t, not u.
3 q :- not r.
4 r :- not s.
5 s :- q, neg_1.
6 neg_1 :- not p.
```

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

In the second step, we generate the dual rule of the predicate to forget using s(CASP).

Original clauses of predicate `p`.

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

Dual rules for predicate `p`.

```
1 not p :- not p_1.
2 not p_1 :- q.
```

As we can see, the dual rules in s(CASP) are generated using additional clauses. As these clauses could make the lecture of the result program difficult, we perform a simplification, by replacing the additional predicates with their value:

Dual rules for predicate `p`.

```
1     not p :- not p_1.
2     not p_1 :- q.
```

Simplified dual rules for predicate `p`.

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

Multiple clauses. If the additional predicates had several clauses, then a combination of them would be performed, contemplating all possible combinations that negate the predicate. For example:

Original clauses.

```
1 p :- q, r.
2 p :- s, t.
```

Dual rules for predicate `p`.

```
1 not p :- not p_1, not p_2.
2 not p_1 :- not q.
3 not p_1 :- not r.
4 not p_2 :- not s.
5 not p_2 :- not t.
```

Simplified dual rules for predicate `p`.

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

Optimized dual rules. Note that s(CASP) by default makes a slight modification to the dual rules generated to avoid exploring redundant options. Given the last example, the dual rules generated by s(CASP) by default would be:

Dual rules for predicate `p`.

```
1 not p :- not p_1, not p_2.
2 not p_1 :- not q.
3 not p_1 :- q, not r.
4 not p_2 :- not s.
5 not p_2 :- s, not t.
```

On the second clause of each additional predicate, the negation of the first case is added to avoid exploring that negation if the first one is already satisfied. This optimization can be disabled using the flag `-d`. In our case, we will use the non-optimized negations.

Example program. After performing the second step, we have obtained the simplified negation for the predicate `p`, `not p :- q`, resulting in the following program:

Program after step 1.

```
1 p :- not q.
2 q :- t, not u.
3 q :- not r.
4 r :- not s.
5 s :- q, neg_1.
6 neg_1 :- not p.
```

Program after step 2.

```
1 p :- not q.
2 q :- t, not u.
3 q :- not r.
4 r :- not s.
5 s :- q, neg_1.
6 neg_1 :- not p.
7 % Dual rules:
8 not p :- q.
```

3.1.3 Step 3: Forget the predicate and its negation

In the third step, we replace all appearances of the predicate in the body of the clauses with the body of its own clauses. Then, we repeat the process with its negation, this time replacing it with the body of the clauses of its dual rule. For example, for the following program, we replace the appearance of `p` in line 2 for the body of the clause shown in line 1 and the negation of `p` in line 3 with the body of its dual rule displayed in line 5. After performing the replacement, the clauses of the predicate to forget and its negation can be discarded, as they are not going to be used in later steps.

Original clauses.

```
1 p :- not q.
2 r :- p.
3 s :- not p.
4 % Dual rules:
5 not p :- q.
```

Clauses after replacing `p` and `not p`.

```
1 p :- not q.
2 r :- not q.
3 s :- q.
4 % Dual rules:
5 not p :- q.
```

Replacing multiple clauses. In the case the clauses of the predicate or its negation were more than one, then the number of clauses for a predicate (for example, `a`) that depends on the predicate marked to forget (for example, `p`) would be the number of clauses of `a` that do not contain `p` plus the product of the number of original clauses of `a` that contains `p` and the clauses of `p` (or its negation). For example, consider the following program:

Original clauses.

```
1 p :- q, r.
2 p :- s, t.
3
4 a :- p, c.
5 a :- d, e.
6
7 b :- p, c.
8 b :- p, d.
```

Clauses after replacing `p`.

```
1 a :- q, r, c.
2 a :- s, t, c.
3 a :- d, e.
4
5 b :- q, r, c.
6 b :- s, t, c.
7 b :- q, r, d.
8 b :- s, t, d.
```

Note that in this case, `p` has two clauses (lines 1 and 2 of the original clauses), and `a` has one clause that contain `p` (line 3 of the original clauses). After replacing `p`, plus the clause that was not altered, we have $1+2=3$ clauses for `a`, in lines 1 to 3 of the program on the right.

In the case of predicate `b`, we have no clauses that do not contain `p`, two clauses that contain `p`, and `p` has two clauses. If we compute the resultant clauses, we have $0+(2*2)=4$ clauses for `b`, which are displayed on lines 5 to 8 of the program on the right.

Even if this replacement could provoke a computational explosion, it ensures that all possible combinations that would satisfy the non-forgotten predicates remain unchanged, and preserves as much as possible the semantics of the program.

Example program. If we consider the clauses of the original program where the predicate `p` or its negation appears, the replacement would be the following:

Relevant program clauses
and dual rules after step 2.

```
1 neg_1 :- not p.
2 not p :- q. % dual
```

Program after replacing `not p`
for the clauses of the dual predicate.

```
1 neg_1 :- q.
2 not p :- q. % dual
```

Note that the (not negated) predicate p does not appear in any clauses. As we can see in lines 1, 7, and 8 of the following program, after making the replacement, the clauses of p and its dual are removed as they are no longer necessary:

Program after step 2.

```
1 p :- not q.
2 q :- t, not u.
3 q :- not r.
4 r :- not s.
5 s :- q, neg_1.
6 neg_1 :- not p.
7 % Dual rules
8 not p :- q.
```

Program after step 3.

```
1 q :- t, not u.
2 q :- not r.
3 r :- not s.
4 s :- q, neg_1.
5 neg_1 :- q.
```

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

In the fourth step, we clean the clauses that contain `true` or `false`, and we add double negations to the auxiliary predicates added in the first step.

Cleaning true clauses. If the program has clauses that contain `true`, an atom that always holds, then that atom is removed. As we can see in the following program, if `true` was the only atom in the clauses of a predicate, then that predicate is transformed into a fact, as it will always hold.

Original clauses.

```
1 p :- true.
2 q :- r, true, s.
```

Clauses after cleaning `true`.

```
1 p. % p is now a fact
2 q :- r, s.
```

Cleaning false clauses. If the program has clauses that contain `false`, an atom that never holds, then those clauses are removed as they cannot be satisfied. As we can see in the following program, if `false` was the only atom in the clauses of a predicate, then that predicate is transformed into a missing predicate, as it will never hold.

Original clauses.

```
1 p :- false.
2 q :- r, false, s.
3 q :- t.
```

Clauses after cleaning `false`.

```
1 % p is now a missing predicate
2 q :- t.
```

Adding double negations. If an auxiliary predicate was added in the first step, it is necessary to preserve the original even loop to add the operator `not not` to each appearance of that predicate on a body of a clause. Note that, as we can see on the following program on line 2, if the auxiliary predicate was negated, three consecutive negations are transformed into a single negation.

Original clauses.

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

Clauses after adding `not not`.

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

Example program. In the example program, we have the additional predicate `neg_1` on line 5. We can see that, on the program on the right, after performing the fourth step, line 5 has been modified by adding the necessary double negation.

Program after step 3.

```
1 q :- t, not u.
2 q :- not r.
3 r :- not s.
4 s :- q, neg_1.
5 neg_1 :- q.
```

Program after step 4.

```
1 q :- t, not u.
2 q :- not r.
3 r :- not s.
4 s :- q, not not neg_1.
5 neg_1 :- q.
```

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

After all predicates have been forgotten, the optional final step is performed. In this fifth step, we convert any double negations that may appear in the program into even loops, with the assistance of another auxiliary that follows the transformation performed in step 1. This modification allows the resulting program to run with s(CASP), as this reasoner does not natively support the `not not` operator.

Example program. In our example program, the double negation present in line 4 is replaced with the negated predicate `not neg_2`, and the corresponding clause has been added in line 6.

Program after step 4.

```
1 q :- t, not u.
2 q :- not r.
3 r :- not s.
4 s :- q, not not neg_1.
5 neg_1 :- q.
```

Program after step 5.

```
1 q :- t, not u.
2 q :- not r.
3 r :- not s.
4 s :- q, not neg_2.
5 neg_1 :- q.
6 neg_2 :- not neg_1.
```

3.2 Implementation as s(CASP) extension

The algorithm is implemented over s(CASP) (Arias et al. 2018) in Ciao Prolog., which is available at the following repository: https://gitlab.software.imdea.org/ciao-lang/sCASP.

To invoke the f_{CASP} operator it is necessary to run s(CASP) with the flag `--forget`. This flag follows a specific structure, `--forget=LIST[/F]`, where

- `LIST` is the list of predicates to be forgotten, e.g., `'p,q'`
- `F` is an optional flag to determine if step 5 of the f_{CASP} algorithm is executed (by passing 1, default value) or not (by passing 0).

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

Figure 3.1: Main function of f_{CASP}

In Figure 3.1, we can observe an approximation of the main function of f_{CASP}. The main call of the operator involves a recursive call where the parameters are as follows: the first parameter is the list of predicates pending to be forgotten, the second parameter is the current state of the program (which will undergo changes throughout the iteration), the third parameter is the final program that will be returned, and the fourth parameter indicates whether step 5 will be performed (with a default value of 1) or not (with a value of 0).

During the first iteration, the predicate list takes the value of `LIST`, which is the mandatory parameter of the flag. Subsequently, the recursive function will be invoked in line 9 to forget the remaining predicates. When there are no predicates left to forget, line 10 or lines 11 and 12 are executed based on the value of the `Flag` parameter, which obtains its value from the optional flag `F`.

3.3 Preliminary validation through examples

In this section, we evaluate examples from the literature. We will apply the operator f_{CASP} to these examples and compare the results w.r.t. the proposals described in those articles, to highlight the advantages of using f_{CASP}.

3.3.1 Forgetting predicates in even loops

We have seen that f_{CASP} can forget atoms that incur in even loops by adding additional predicates. This is a form of conserving the symmetry in answer sets (Knorr and Alferes 2014), that is, preserving the predicates not forgotten in the answer sets even when adding additional rules, as strong persistence cannot be achieved (in some cases) without adding additional predicates (Gonçalves, Knorr, and Leite 2016; Berthold et al. 2019).

P_1 = Example 3 from (Gonçalves, Knorr, and Leite 2016)

```
1 a :- p.
2 b :- q.
3 p :- not q.
4 q :- not p.
```

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

```
1 a :- not not neg_2.
2 b :- not not neg_1.
3 neg_1 :- not not neg_1.
4 neg_2 :- not neg_1.
```

The program presented in Example 3 by Gonçalves at el. in 2016 (Gonçalves, Knorr, and Leite 2016) has the models `{a, p}` and `{b, q}`. The result of the generated program using f_{CASP} has the models `{a, neg_2}` and `{b, neg_1}`. I.e., we obtain the same models (ignoring the presence of the auxiliary predicates `neg_1` and `neg_2`).

In the following example, we slightly modify P_1 to check the influence of more complex even loops, due to the presence of `r` in lines 4 and 5.

$P_{1'}$ = Variant of P_1

```
1 a :- p.
2 b :- q.
3 p :- not q.
4 q :- r.
5 r :- not p.
```

$f_{CASP}(P_{1'}, \{p, q\})$

```
1 a :- not not neg_2.
2 b :- r.
3 r :- not not neg_1.
4 neg_1 :- r.
5 neg_2 :- not r.
```

Note that $P_{1'}$ has the models `{r, q, b}` and `{p, a}`, and the result of forgetting program using f_{CASP} has respectively the models `{r, b, neg_1}` and `{a, neg_2}`, i.e., the same models ignoring `p`, `q`, and the auxiliary predicates.

3.3.2 Forgetting predicates present in double negations

P_2 = Example 4 from (Knorr and Alferes 2014)

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

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

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

Consider the program presented in Example 4 by Knorr and Alferes in 2014 (Knorr and Alferes 2014). In the work, the authors mention that it is not possible to satisfy strong persistence when we forget `p`. They explain that the program P_2 has two models `{p, q}` and `{r}` and that adding either `q` or `r` as facts to P_2 simply adds the atom to both answer sets, i.e., $P_2 \cup$ `{q}` has models `{p, q}` and `{q, r}` and $P_2 \cup$ `{r}` has models `{p, q, r}` and `{r}`. They require that $P_2' = f_{xx}(P_2, p)$, where f_{xx} represents whatever forgetting operator is, has two models `{q}` and `{r}`, and that $P_2' \cup$ `{q}` and $P_2' \cup$ `{r}` also both have two models, namely `{q}` and `{q, r}`, and `{r}` and `{q, r}` respectively. They consider that such a program P_2' does not exist over `{q, r}`, because (a) it is required to be symmetric in `q` and `r`, (b) it is needed that precisely only one of `q` and `r` is true in each answer set of P_2', but (c) adding either of the two explicitly, must not avoid the existence of an answer set that contains the other and in which both atoms are true —note that if we restrict P_2' to have no auxiliary predicates, then P_2' does not exist. Since it is impossible to satisfy these conditions without an additional predicate, we propose the use of auxiliary predicates, and as we show in this example with them it is possible to achieve the desired result. I.e., if we add `{r}` to the result of forgetting program $f_{CASP}(P_2, \{p\})$, we get the models `{r, q}` and `{r, neg_1}`, while adding `{q}`, the stable models are `{q}` and `{q, r, neg_1}`. In both cases, ignoring the additional predicate leads to the desired solution that is sought when `p` is forgotten, and which could not be obtained without the use of the auxiliary predicate.

This transformation applies in the same way to other examples such as Example 4 from (Gonçalves, Knorr, and Leite 2016).

3.3.3 Forgetting multiple predicates regardless of the order

P_3 = Example 1 from (Berthold 2022)

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

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

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

Example 1 by Berthold in 2022 (Berthold 2022) is a perfect program to test the behavior of an operator against an even loop while forgetting multiple predicates, iteratively in different orders. The models of this program are `{q}` and `{p}`, and therefore after forgetting the atoms `p` and `q` in any order the expected models are the empty models, i.e., `{}` and `{}`.

The operator f_{SP}, when applied once for forgetting `q` and then again for `p` does not report the same result as when forgetting `p` and then `q`. The result of the generated program, $f_{SP}(P_3, \{q,p\})$, is empty, i.e., it has a unique model, `{}` (consistent with the expected results). But, the result of the generated program, $f_{SP}(P_3, \{p,q\})$, is `a:- not not a`, which have `{a}`, and `{}` as stable models (with model `{a}` a being a non-expected model).

Applying f_{CASP} in both cases, by forgetting first `p` and then `q` or forgetting first `q` and then `p`, we obtain the same result of generated program, shown above, and it has the stable models `{}` and `{neg_1}`, which are the desirable models.

3.3.4 Comparing the required auxiliary predicates

P_4 = Example 5 from (Berthold et al. 2019)

```
1 q :- not not q, b.
2 a :- q.
3 c :- not q.
```

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

```
1 a :- b, δ_q.
2 c :- not δ_q.
3 c :- not b.
4 δ_q :- not not δ_q.
```

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

```
1 a :- not neg_1, b.
2 c :- not not neg_1.
3 neg_1 :- not not neg_1.
4 neg_1 :- not b.
```

In the result of the generated program described in Example 5 by Berthold et. al in 2019 (Berthold et al. 2019), which is located at the center of the figure above, the operator f_{AC} introduces an auxiliary predicate to represent the as-dual of `q`, the predicate to forget.

The stable models of this program are `{c}` and `{c, δ_q}`, while for The result of the generated program using f_{CASP} the stable model is unique, `{c, neg_1}`. While in both cases the stable models are equivalent (ignoring the auxiliary predicate) to the stable model of the original program, `{c}`, f_{AC} returns two models instead.

Chapter 4

(Legal and Ethical) Motivation

In this chapter, we provide further details about the motivation of this work, focusing on the legal and ethical aspects of the goal of f_{CASP} and the use cases defined.

First, we discuss the conflict between the users' right to receive an explanation for high-risk decisions made by AI decision-making systems and the right to maintain the privacy of sensitive information. Specifically, we examine a particularly sensitive criterion employed by the system that determines whether a candidate is accepted into the renewable energy community: being a victim of gender-based violence. Then, we describe the reasoning behind the energy assignment system, how it can be aligned with values, and how we consider fairness in our energy distribution process.

4.1 Right to Explanation vs. Right to Privacy

As we mentioned before, we will apply the forgetting operator to a real use case, the admission of new members to a local renewable energy generation scheme within an agricultural cooperative, and the distribution of the energy generated.

First, to streamline the admission of a potential new member of the local renewable energy community and the assignation of the resource, it is proposed an automatic decision model, s(LAW) by (Arias et al. 2023), which can not only determine whether an applicant will be admitted or not (or it is uncertain) and the energy coefficient assigned based on their circumstances, but also provide justifications for its decision since it is executed using s(CASP). The explainability of the model is fundamental, as this process needs to be committed to the transparency and fairness of its methods for it to be trustworthy. In addition, justifications allow a better understanding of the logic behind a decision, helping

the human controller that needs to review automatically processed decisions that have a significant effect on the subject of the decision, as required by Article 22 of the General Data Protection Regulation (GDPR).

However, if the places that can be assigned are limited, it could be that a member is not accepted due to the demand for places being bigger than the offer, converting this process into a competitive procedure. In that case, part of the justification is showing that other candidates have scored higher than them, being necessary to publicize the list of candidates accepted, although this publicity must include the final result of the ranking, so that it does not imply indiscriminate access to information, or including partial results that may respond to sensitive data or confidential information. Taking that scenario into account, we can assume that the explanations must be accessible to all stakeholders (and the controller), which may lead to the disclosure of sensitive information in the public admission lists or the information transmitted to individuals appealing the list. E.g., one type of sensitive information that is collected is whether the applicant has been affected by gender-based violence, given that this is one of the parameters in the filtering process. This information should not be disclosed in accordance with Organic Law 1/2004, which mandates comprehensive protection measures against gender violence. In particular, Article 63 establishes that:

> **Art. 63 Data protection and limitations on publicity:** In proceedings and procedures related to gender violence, the privacy of the victims shall be protected; in particular, their personal data, those of their descendants, and those of any other person in their care or custody.

For this reason, a candidate may object to the publication of his/her admission for security reasons, as a victim of gender-based violence, in application of Art. 18.1.d of the General Data Protection Regulation.[1]

To solve the challenge of satisfying both explainability and privacy, it is needed that the justifications do not reveal sensitive information, which can be achieved using forgetting, a technique that can "hide" the sensitive information in the publication of the list of those admitted to the plan and, in any case, on the information to be sent to whoever, if any, files an appeal against the list. The subsequent sections elaborate on the criteria upon which the programs (subject to forgetting) are founded, providing insights into their implementation and assessing the outcomes of applying the operator in Chapter 5.

[1] REGULATION (EU) 2016/679 OF THE EUROPEAN PARLIAMENT AND OF THE COUNCIL of April 27, 2016 relating to the protection of natural persons with regard to the processing of personal data and the free circulation of these data and repealing the Directive 95/46/EC (General Data Protection Regulation) https://www.boe.es/doue/2016/119/L00001-00088.pdf.

4.2 Requirements for the admission of a farmer

A community involved in local renewable energy generation can be likened to a renewable energy community. Following this idea, we can center the criteria for admission on the requirements, beliefs, and standards of these communities. In Spain, the requirements for membership in a renewable energy community (REC) are regulated by Royal Decree 23/2020, which defines RECs as:

> **Article 4. Amendment of Law 24/2013, of December 26, 2013, on the Electricity Sector:** Renewable energy communities, which are legal entities based on open and voluntary participation, autonomous and effectively controlled by partners or members that are located in the vicinity of renewable energy projects owned and developed by such legal entities, whose partners or members are natural persons, SMEs or local authorities, including municipalities, and whose primary purpose is to provide environmental, economic or social benefits to their partners or members or to the local areas where they operate, rather than financial gain.

From this definition, we can deduce the following requirements: a member must be located near a power generation point, must have volunteered, and must support positive development for the overall community. The community may add additional requirements to specify compliance with the latter point.

The first point is satisfied as one requisite to be admitted is to generate a minimum amount of renewable energy. All members have volunteered to form part of the community, which covers the second point. Among the criteria established by the community to comply with the positive (environmental or social) development of the local area are: (i) maintaining low carbon emissions resulting from the farmer's agricultural activities, (ii) providing a minimum standard of work quality for workers, (iii) engaging in activities that protect local wildlife (for example, dedicating a portion of land to flower cultivation for bee preservation), and/or (iv) fulfilling a specific condition (such as being a victim of gender-based violence or having a disability) is a criterion for which the community offers additional support.

If processes to determine admission to the community were automated, through the use of Artificial Intelligence systems, according to the proposed AI Regulation by the EU, these systems would be classified as high risk, unless the information output from the system is merely incidental to the final action or decision to be taken (Article 6 and Annex III of the proposed AI Regulation). Therefore, it would be subject to strict compliance with the requirements for such systems, including impartiality, transparency, and explainability.[2]

[2]In particular, human agency and oversight, technical robustness and safety, privacy and data

4.3 Energy Assignment in Cooperatives

This section explains the importance of making decisions taking into account their alignment with values. Specifically, we present as a use case the generation and distribution of energy in agricultural cooperatives.

4.3.1 Ethical Values in Energy Management

Agricultural cooperatives, in which farmers share their assets, allow better access to equipment, services, and supplies, reducing costs that would be difficult to obtain individually. These cooperatives are already well established in the sector, being reported as a beneficial factor to the technological and economic growth of farms, as well as to social and environmental improvement, being these changes are more pronounced in small cooperatives (Kustepeli et al. 2023; Lin et al. 2022).

One of the shared resources is energy, which, when generated locally, represents a good solution to address the issue of inadequate energy connectivity frequently experienced in rural areas. In addition, local generation reduces the energy demand on the general energy grid by decentralizing energy production, a very important point given the growing energy demand of recent decades.

In Spain, the creation of a renewable energy community is regulated by Royal Decree-Law 23/2020, but the criteria for the distribution of energy are at the discretion of the community itself. Deciding how to generate this local energy can be aligned with values that promote positive change on a common ground. For example, we could use as a guide the values of environmental stewardship and climate change mitigation, defined as two of the ten objectives stated in the European Common Agricultural Policy (CAP) for the years 2023 to 2027. Consideration of these values would imply an increase in the amount of energy generated from green and renewable sources.

The energy production is focused on renewable sources. Because of this characteristic, the process itself also helps the alignment with other values, for example, competitiveness, thanks to the reduced cost of green energy (Osman et al. 2023) and the compatibility with land exploitation, thereby improving the overall benefits obtained as discussed in (Jain et al. 2021).

governance, transparency, diversity, non-discrimination and fairness, societal and environmental wellbeing and accountability, vid. Proposal for a Regulation of the European Parliament and of the Council laying down harmonized rules on artificial intelligence (artificial intelligence Act), 2021.

4.3.2 Energy Distribution considering Fairness

Once we have defined the sources of power generation we have to determine the criteria for distribution within the cooperative. In this case, we can consider other objectives defined on the European Common Agricultural Policy, e.g., it could be taken into account generational renewal, rewarding the farmers which creates opportunities and formation in the sector for new generations, or biodiversity care, rewarding farmers which takes into account the preservation of the local wildlife. In general, the distribution of the generated energy is expressed in energy distribution coefficients, which are agreed upon through democratic community discussion within the cooperative. Usually, these values, or, in similar contexts, the cost of the energy consumed, are assigned based on consumption levels.

On the other hand, for making a fair energy distribution in a residential complex, different tariff designs have been proposed, contemplating the costs, consumption, and use of the energy of the consumers (Roberts, Sharma, and MacGill 2022). Some adaptations made to make the system fairer are proposing discounts to promote sustainable consumption while reducing the burden of the cost of the system implementation on the initial members.

To promote a better alignment with positive values that can improve the common well-being of the community, we propose an alternative approach: using the farmers' adherence to the ethical values of the cooperative to distribute the energy. This approach would reward the farmers who benefit the overall community and/or encourage the rest of the members to emulate their actions. For example, rewarding a farmer who uses environmentally friendly fertilizers with improved energy supply conditions may prompt the community to adopt similar practices, leading to a vision more aligned with the care of the environment within the cooperative. Another case could be punishing a farmer who misuses the resources provided, fomenting a resource-saving, sustainable behavior among the members (but it is not always possible to include penalties when distributing resources in cooperatives).

In this context, it is interesting to consider intelligent systems that make it possible to automate distribution decisions to avoid partial or unfair answers. The use of machine learning has been explored to predict and optimize various aspects of renewable energy communities, including storage, management, and energy stability of the energy (Hernandez-Matheus et al. 2022). The automatization of the allocation decision can streamline the whole process, especially when complex variables are involved, and can ensure a correct and fair distribution, as these systems, if well-designed and not ill-intended, do make impartial judgments. However, most black-box models cannot provide justifications for their decisions. This lack of transparency can affect the trust of the users in the system, hampering its implementation. Moreover, if the system does not explain the reason behind the solutions it proposes, it can not be assured that this solution is not biased.

In addition, the new Regulation on AI, by the European Union, sets a minimum requirement on explainability for models that are considered high-risk.

However, as we have already mentioned, these explanations may result in a leakage of private, confidential, or legal information belonging to users. In this case, there could be not only a potential leakage of sensitive information, which violates the General Data Protection Regulation (GDPR), but also business secrets such as the salary complements of the workers, can be revealed, which may affect competitiveness. Therefore, we propose the use of f_{CASP} to forget sensitive information and preserve the privacy and confidentiality of the users, while preserving the explainability of the model.

Chapter 5

Transparent and fair energy assignment using f_{CASP}

This chapter will describe the implementation of both decision systems, i.e., the admission system and the energy assignment system. The first system is a use case in propositional logic, while the second one is implemented in First Order Logic.

In the implementation of both systems, for the sake of simplicity, we assume an agricultural cooperative with four farmers/members: Bob, Adam, Alex, and Lucy. Only the last three members satisfy the requirements for being admitted to the energy plan.

5.1 Use case in Propositional Logic

In the following section, the design and implementation of the model that will manage the admission of candidates to the energy generation plan will be detailed. This model will use the criteria established in Section 4.3.1.

In this first system, to know if a candidate will be accepted, we use a common program that models the rules (membership_acceptance.pl), a program containing the query (query.pl), and the candidate's specific program which contains a list of facts describing the candidate's characteristics. For example, to find out whether Adam will be admitted, the following command is invoked:

```
scasp --tree -n0 membership_acceptance.pl adam.pl query.pl
```

where `--tree` indicates that we want to obtain a justification in tree format for the decision, and `-n0` indicates that we want to obtain all possible answers.

5.1.1 Modeling a fair candidate selection system

This section details the implementation of the decision system for the acceptance of potential candidates for the power generation plan following the criteria described in Section 4.3.1. The common rules that model the criteria are established in the file membership_acceptance.pl.

For an applicant to be admitted, they must meet three conditions: a minimum resource contribution, comply with specific requirements that show a commitment to the community or a special situation, and maintain a low energy consumption (or green energy use) profile.

First, meeting the established minimum resource contribution to the cooperative is represented by a fact, i.e. it is represented as an inherent characteristic of a candidate that will be reflected in their specific module. This characteristic is called `met_minimum_resource_contribution` in the implemented programs.

Second, the requirements include having low carbon emissions, maintaining fair working conditions, having measures for the protection of local flora and fauna, or complying with special conditions established to favor disadvantaged minority groups.

To prove that the agricultural system implemented by the farmer is low in carbon emissions, the applicant can either present a valid external certificate to confirm this (`low_carbon_certification`) or submit (and pass) his system to a test conducted by the cooperative itself (`cooperative_test_passed`), which analyzes the energy consumption of the system, the energy sources used and the emissions produced. If the system passes the test, it will be registered in the database and a certificate of energy efficiency will be issued.

Similar to the minimum contribution, the demonstration of a quality work environment (`met_minimal_work_quality`) above a certain threshold or the adoption of measures to protect autonomous wildlife (`wildlife_protection`) is given by an inherent characteristic of the candidate, modeled in the form of a fact. These facts are then checked against the cooperative's database. On the other hand, there are two special social conditions for belonging to a disadvantaged minority group: one is to be a victim of gender-based violence, and the other is to have a degree of disability higher than 15%.

Finally, to be eligible, applicants must also demonstrate that they have an agricultural system that requires little energy from the grid, either by opting for low-consumption technologies or by generating (green) energy for their own consumption.

```
low_consumption:-
  cooperative_test_passed.
low_consumption:-
  energy_efficiency_certificate.
low_consumption:-
  green_energy_generation.

energy_efficiency_certificate:-
  external_certificate.
energy_efficiency_certificate:-
  cooperative_test_passed.

cooperative_test_passed:-
onsite_exam_passed.
cooperative_test_passed:-
green_energy_generation,
not last_exam_failed.

last_exam_failed:-
  not energy_efficiency_certificate.
```

Figure 5.1: Clauses that verify the low grid energy demand of the candidate.

In the following lines of membership_acceptance.pl, it is modeled that in order to prove that the technologies used are energy efficient, the system can be tested by the cooperative (`cooperative_test_passed`) or an energy efficiency certificate (`energy_efficiency_certificate`) can be presented. This certificate can be either an external certificate or the certificate issued by the cooperative when the test is passed. If an applicant demonstrates that it generates energy from green sources, i.e. the second case in the demonstration of low consumption, it will also be considered to have passed the cooperative test unless the (same) system has failed a prior test.

Since submitting these certificates or taking the exam is optional, some uncertainty is modeled using an even loop to account for the case where a candidate meets the above requirements but has not submitted the appropriate documentation. In such a case, the candidate will be asked for the documentation if it is required to be admitted to the plan.

5.1.2 Evaluation of the propositional use case

In this section, we apply f_{CASP} to the defined use case by invoking s(CASP) with the flag `--forget` (as detailed in Section 3.2).

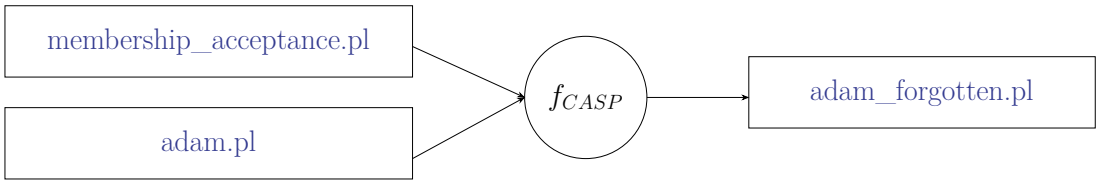


Figure 5.2: Input and output files for the application of f_{CASP} to forget sensitive data of Adam

First, we analyze the result of the generated programs considering different scenarios. Next, we compare the justifications for the original program against a shorter justification and the justifications for the result of the generated program. Figure 5.2 shows input and output[1] files for the application of f_{CASP} to forget sensitive data of Adam:

All files used/generated in this evaluation are available at https://github.com/Lu-all/energy-distribution-with-scasp-and-fcasp, and we used s(CASP) version 0.24.05.29.

Forgetting sensitive data

First, let us focus on the predicate `met_requirement`, and consider that the following member's characteristics `low_carbon_certification`, `met_minimal_work_quality`, `wildlife_protection`, `special_condition`, `gender_based_violence_victim`, and `disability_condition` are sensitive and must be forgotten, while having passed the cooperative test (`cooperative_test_passed`) is not considered sensitive information.

Adam: The remaining `met_common_requirement`'s clauses after forgetting are:

```
1 met_requirement :-
2   low_carbon_emissions.
3 low_carbon_emissions :-
4   cooperative_test_passed.
```

where the clause in line 1, and the fact that Adam has proved that the system has low carbon emissions through the cooperative exam remain because `low_carbon_emissions` and `cooperative_test_passed` are not sensitive data.

Alex: S/he is a victim of gender-based violence and has a disability condition. Therefore, after forgetting the clauses that remain are:

```
1 met_requirement :-
2   low_carbon_emissions.
3 met_requirement.
4 low_carbon_emissions.
5 low_carbon_emissions :-
6   cooperative_test_passed.
7 cooperative_test_passed.
```

where the requirements are met because of meeting a low level of carbon emissions and/or other motives (being a victim of gender-based violence and having a disability condition), but even if these characteristics are taken into account, they do not appear in the program, as we can see on the clauses.

[1] Program obtained by invoking `scasp membership_acceptance.pl adam.pl --forget='low_carbon_certification,met_minimal_work_quality,wildlife_protection,special_condition,gender_based_violence_victim,disability_condition,energy_efficiency_certificate,green_energy_generation,external_certificate,onsite_exam_passed'/1`

Bob: In this case, the clauses that remain are:

```
1 met_requirement :-
2   low_carbon_emissions.
3 low_carbon_emissions :-
4   cooperative_test_passed.
```

Since Bob does not satisfy the `cooperative_test_passed` (thereby, Bob does not satisfy `met_requirement`), we would not obtain a model for `?-accept_membership`, however, one additional advantage of s(CASP) is its ability to provide justifications for the absence of models, by invoking the negated query, `?-not accept_membership`, as we discuss in Section 5.1.2.

Lucy: The left clauses for Lucy are the same as Bob's, but s/he produces green energy and has not failed the last cooperative exam. Therefore, based on these achievements, it is assumed that they have the potential to pass the cooperative exam and its validation would be needed for acceptance into the plan.

Forgetting data involving even loops

The criteria for accrediting a low level of grid energy consumption, modeled by the predicate `low_consumption` in Fig. 5.1, includes an even loop, and therefore, forgetting sensitive data around these criteria allows us to highlight the potential of f_{CASP}. Note that auxiliary predicates are sometimes necessary to preserve the even loop. The sensitive data that we consider in this scenario are `energy_efficiency_certificate`, `green_energy_generation`, `external_certificate`, and `onsite_exam_passed`.A comparison of the justifications for these farmers is discussed in the following section.

Adam: The only `low_consumption` clauses remaining after forgetting are

```
1 low_consumption :-
2   cooperative_test_passed.
3 cooperative_test_passed.
4 last_exam_failed :-
5   not neg_2.
6 neg_2 :-
7   not neg_1.
8 neg_1 :-
9   not cooperative_test_passed.
```

In this case, even if the even loop does not disappear, and the predicate always succeeds (lines 1 to 3), because the s/he passed an on-site exam, information that is hidden in the model and the justifications. The justifications of the original program and the version obtained after forgetting are compared in the following section.

Alex: In this case, the clauses that remain are:

```
1 low_consumption.
2 low_consumption :-
3   cooperative_test_passed.
4
5 last_exam_failed :-
6   not neg_2.
7 neg_2 :-
8   not neg_1.
```

Note that in this case the even loop has been resolved through having an external certificate, which is considered sensitive information, so the even loop has disappeared, and `low_consumption` has been converted into a fact (line 1) that always holds as the characteristic that proved this point has been hidden. Lines 5 to 8 model the conditions for having failed the last exam, but as `neg_1` (line 8) does not appear in the head of any clause, it is always false.

Bob: In this case, the clauses related to `low_consumption` are the same as those for Alex, but without the first line as Bob did not meet any clause that demonstrates this requirement.

Lucy: In this case, the clauses related to `low_consumption` are the same as those for Alex for the same motive, but in this case, the sensitive characteristic is generating green energy.

Comparing the justifications generated using s(CASP)

In this section, we compare the different justifications generated by s(CASP) for the original program and the forgotten versions of the four members.

Original: By invoking `scasp -d -n0 --tree membership_acceptance.pl adam.pl query.pl` it generates all the answers (flag `-n0`) with their justification (flag `--tree`) of the original program, in membership_acceptance.pl, and Adam, in adam.pl, for the query `?-accept_membership`, in query.pl (the justifications are available in adam_justification.txt).

Short: To manipulate the justifications, and only show specific predicates, s(CASP) provides the flag `--short` and the directive `#show`, which can be used to select the predicates to be shown —query.pl selects those predicates that are not sensitive— and by invoking '`scasp -d -n0 --short --tree membership_acceptance.pl adam.pl query.pl`', s(CASP) generates the answers and justification without the sensitive data for Adam (available in adam_short_justification.txt).

Forgotten: Finally, by invoking `scasp -d -n0 --tree adam_forgotten.pl query.pl` we obtain the answers and justifications of the result of the generated program (available in adam_forgotten_justification.txt).

```
1 low_consumption :-
2   energy_efficiency_certificate :-
3       proved(cooperative_test_passed).
```

(a) Original program.

```
1 low_consumption :-
2   proved(cooperative_test_passed).
```

(b) With flag `--short` or after applying f_{CASP}.

Figure 5.3: Justifications of answer 1 for `?-low_consumption` and Adam.

Adam: For the original program, the query `?-low_consumption` generates two answers because there are two possible ways to accredit this requirement. In both cases, the justifications are based on sensitive data, so the short justifications are identical. Figure 5.3 displays a comparison of the original and short first justification of the original program and the justification of the program generated by f_{CASP}. In addition, line 3 of the original justification (line 2 of the short one) shows that the low level of carbon emissions was demonstrated by the cooperative test, which was passed. The result of the generated program only outputs one answer, which is identical to the short justifications.

Alex: In this case, the low consumption levels are always proven through an external certificate. As this case is similar to the first one, a profound analysis is not needed. However, the corresponding files containing the justifications (alex_justification.txt, alex_short_justification.txt, and alex_forgotten_justification.txt are available in the repository).

Bob: As we mentioned before for the query `?-accept_membership` there is no model, meaning that s/he is not accepted in the plan. So to understand why s/he is not accepted, we can query `?-not accept_membership` and check the different answers/justifications, as before. Additionally, we use the flag `-d`, which disables certain optimizations regarding the generation of dual rules. For the sake of brevity let us only compare the justification for the second answer. Fig. 5.4a shows the first justification for the original program where we observe that all the predicates required to meet the common requirements are listed, and (ii) Fig. 5.4b shows the short justification which in this case is equal to the justification obtained from the result of generated program.

Lucy: Since the model to accredit the low demand of grid energy (Fig. 5.1) contains even loops, in this scenario the first answer for the query `?-low_consumption` succeeds under some assumptions. Fig. 5.5 compares the three possible justifications for this answer: (i) Fig. 5.5a shows the justification for the first answer of the original program, where the low demand has been accredited assuming (and later proving) that the member has an energy efficiency certificate (label `proved/1` in line 2). (ii) Fig. 5.5b shows the short justification for the first answer

```
not accept_membership :-
not met_requirement :-
    not low_carbon_emissions :-
        not low_carbon_certification,
        not cooperative_test_passed :-
            not onsite_exam_passed,
            not green_energy_generation.
    not met_minimal_work_quality,
    not wildlife_protection,
    not special_condition :-
        not gender_based_violence_victim,
        not disability_condition.
```

(a) Original program.

```
not accept_membership :-
  not met_requirement :-
      not low_carbon_emissions :-
          not cooperative_test_passed.
```

(b) With flag `--short` or after applying f_{CASP}.

Figure 5.4: Justifications of answer 2 for `?-not accept_membership` and Bob.

that not only hides the sensitive data, but also that at some point in the execution some assumptions were necessary to satisfy the query (that is, the lines with the label `proven`), and (iii) Fig. 5.5c shows the justification for the first answer of the program after applying f_{CASP}, where we observe an assumption in line 2, but without exposing sensitive data. Note that the assumption has been made over a non-sensitive predicate, as this version of the program does not contain the predicate `energy_efficiency_certificate`.

5.2 Use case in First Order Logic

The criteria established in Section 4.3.2 will be used to calculate the energy distribution, using a model whose design and implementation will be defined below.

```
low_consumption :-
  proved(energy_efficiency_certificate).
```

(a) Original program.

```
low_consumption.
```

(b) With flag `--short`.

```
low_consumption :-
  proved(cooperative_test_passed).
```

(c) After applying f_{CASP}.

Figure 5.5: Justifications of answer 1 for `?-low_consumption` and Lucy.

5.2.1 Modeling fair energy assignment

For the sake of simplicity, in the practical example of this work, we model a fair energy assignment decision maker, which will reward farmers according to how fair the income of their employed workers is (we have chosen this criterion as it is another objective defined in the European Common Agricultural Policy). Moreover, this value is key to supporting the socio-economic sustainability of the agricultural sector, as workers in this field have reported remarkably low incomes compared to other jobs. Building a support network and ensuring a competitive income is crucial for the maintenance of the farm and, as a consequence, the improvement of the quality of life of its workers and the food sustainability of society. However, this use case could be expanded to contemplate other values or objectives in a similar way.

We assume that in this cooperative, each member (on the recommendation of the association) uses a rule-based model to manage their business. In order to calculate the coefficients for the distribution of energy, in this example, the cooperative asks for information on the salary and productivity of their workers. For this purpose, each farmer will create a submodule and, in order to preserve the confidentiality of their data, they will apply forgetting to the generated submodule before sending it to the cooperative for its use on the XAI system that calculates the energy percentages. From this point on, we will refer to these submodules as modules.

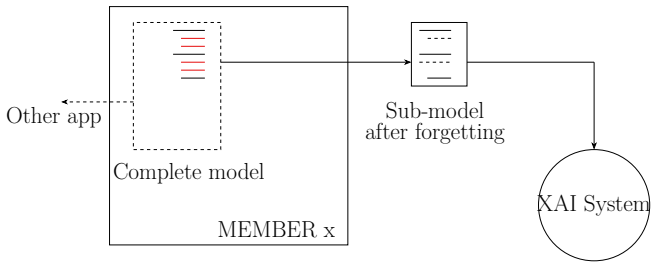
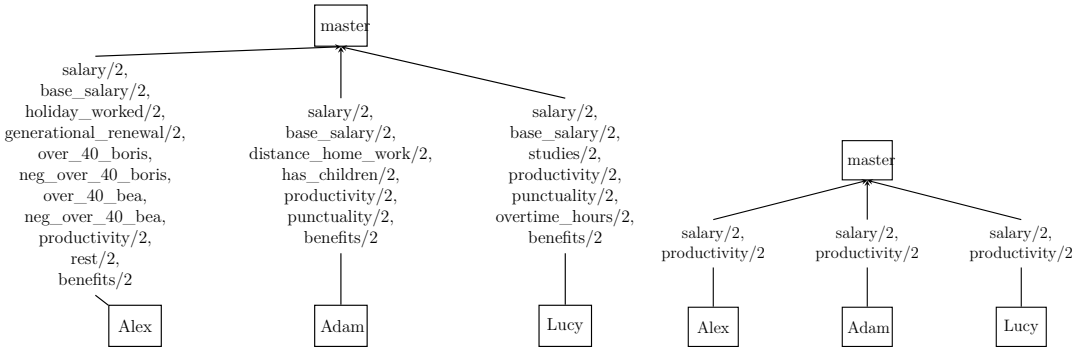


Figure 5.6: Diagram of farmer's modules.

Three members/farmers were admitted into the energy generation plan: Adam, Lucy, and Alex, having each of them has some workers: (i) Adam has two workers, Eric and Mary; (ii) Lucy has three, Julie, Jean, and Jane; and (iii) Alex has two, Boris and Bea. The information regarding each farmer and its workers is modeled in separate modules: module_adam.pl, module_lucy.pl, module_alex.pl, respectively. Fig. 5.7 shows the modules and predicates defined.

Each module calculates the salary and productivity of each worker, depending on different parameters considered by each cooperative member. The fair income points are assigned to each member in the master module based on the mean of the relation between salary and productivity of his/her workers. Note that to calculate the salary each farmer considers different parameters:



(a) Original modules (b) Forgotten modules

Figure 5.7: Diagram of predicates on each module.

- **Adam** adds to the base salary `base_salary/2` the following concepts:

  `distance_home_work/2`: to assist with transportation expenses, Adam offers additional compensation based on the distance an employee resides from the workplace.

  `has_children/2`: to assist with family costs, extra pay is given to those workers with families.

- **Lucy** obtains the final salary adding to the base salary:

  `studies/2`: this additional amount depends on the employee's educational level

  `punctuality/2` and `overtime_hours/2`: to gauge the level of productivity of her workers, she adds to the benefits obtained per employee bonus points based on punctuality and overtime hours logged.

- Finally, **Alex** calculates the salary of his/her employees adding to the base salary the following extras:

  `holiday_worked/2`: a fixed amount of points is assigned based on the number of holidays worked in the last year, encouraging workers by providing a bonus for extra work.

  `generational_renewal/2`: to encourage the inclusion of young workers in the sector, Alex has decided to provide extra points to employees under 40 years old. However, as it is not always possible to obtain the year of birth of an employee, for the missing values, such as Bea's birthdate, an even loop has been created.:

```
1 generational_renewal(bea, 0):- over_40_bea.
2 generational_renewal(bea, 100):- not over_40_bea.
3
4 over_40_bea:- not neg_over_40_bea.
5 neg_over_40_bea :- not over_40_bea.
```

Then, we implement a master module, called master.pl, which represents the cooperative administrator, and computes the energy distribution based on the data contained/computed in the previous modules.

To assign the available energy to each member, the master module calculates the corresponding percentage based on the fair income points that each member obtains depending on the salary and the productivity of their workers. Fair income points are determined by calculating the difference between an employee's productivity, measured in euros, and their salary. As we already mentioned, the salary is calculated by adding their base salary to any additional payments defined by the employer. To assess the fairness of employees' incomes for a given farmer, each worker is associated with a `fair_income_x` predicate (where `x` is the farmer ID). To obtain the resulting assignment invoke

```
scasp --tree -r=2 -n1 master.pl module_adam.pl module_lucy.pl module_alex.pl
```

where the flag `--tree` generates the justification (available at justification_percentage_original.txt), `-r=2` restrict the number of decimals to 2, and `-n1` returns only the first answer. Additionally, by adding the flag `--html`, it is possible to obtain the justification as an HTML navigated file (available at justification_percentage_original.html).

However, these explanations contain business secrets, i.e., the criteria of how the farmers compute extra salary. While it is possible to manipulate the justifications generated by s(CASP) (see (Arias et al. 2020) for details) in this case it is necessary to modify the private module of each farmer because none of them want others to know how the salary values are calculated. Fig. 5.7 shows the minimal predicates that each farmer should share with the master, i.e., `salary/2` and `productivity/2`. Therefore, after forgetting the rest of the predicates the resulting modules are: module_adam_forgotten.pl, module_lucy_forgotten.pl, and module_alex_forgotten.pl, respectively.

Currently, the operator is limited to execution within propositional programs, so, in the model proposed, the operator cannot be applied due to the difference in the treatment of predicates in comparison with propositional rules.

The dual rules of s(CASP) allow us to derive the negation of predicates while preserving the implications of the variables involved. This process involves creating a set of rules for each clause in the original rule. However, challenges arise when variables in the body of a clause are not present in the head, requiring the use of the `forall` predicate. These predicates are not part of the standard ASP semantics but are specific to the s(CASP) metainterpreter. Based on these dual rules, extending the algorithm to support its application with generic ASP predicates is possible, and it has been identified as a relevant future line of work. The use case modules have been forgotten following the desired result of the future implementation while making a grounding optimization to report a clearer and simpler result.

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

(a) $Eric_0$ = Original program

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

(b) $f_{CASP}(Eric_0, set_{Adam})$

```
salary(eric,1400) :-
   base_salary(eric,1200),
   distance_home_work(eric,100),
   has_children(eric,100),
   1400 is 1200+100+100.
```

(c) Justification of original program.

```
salary(eric,1400) :-
   1400 is 1200+100+100.
```

(d) Justification after forgetting.

Figure 5.8: Eric's salary programs and justifications (original vs. forgetting).

5.2.2 Evaluation of the First Order Logic use case

In this section, we present the results of applying forgetting to the owner modules. However, a naive application of f_{CASP} for predicate ASP programs can lead to an exponential number of irrelevant clauses. To address this, we introduce a theoretical optimization of generated programs. It is worth noting that these optimizations are already integrated into most grounders used by bottom-up implementations of ASP, such as the *magic set* technique (Alviano et al. 2012) for large datasets and *external sources* (Calimeri, Cozza, and Ianni 2007) for interpreted function symbols.

The examples were executed using s(CASP) ver.0.24.05.29 (available at `https://gitlab.software.imdea.org/ciao-lang/sCASP`). As mentioned earlier, all the files used in the evaluation can be accessed at `https://github.com/Lu-all/energy-distribution-with-scasp-and-fcasp`.

Adam's module: forgetting Eric's salary

First, Fig. 5.8a shows the predicate for Eric's salary in the module associated with Adam, and Fig. 5.8b shows the result of forgetting set_{Adam} w.r.t. the predicate `salary/2`, where, as we mentioned in Section 5.2.1, the set of predicates involved in the calculation of Eric's salary, set_{Adam} = {`base_salary/2`, `distance_from_home/2`, `has_children/2`}, are omitted to protect confidential information. Then, Fig. 5.8c shows the justification obtained for the query `?-salary(eric,Salary)`, and Fig. 5.8d shows the justification obtained after forgetting the predicates in set_{eric}. The latter justification is obtained by invoking:

```
scasp --tree -r=2 -n1 master.pl module_adam_forgotten.pl
```

module_lucy_forgotten.pl module_alex_forgotten.pl

which can be seen as a subset of the justification for the query `?-percentages(Percentages)` available at justification_percentage_forgotten.txt (the HTML navigated version is available at justification_percentage_forgotten.html).

Alex's module: forgetting Bea's salary

Let's consider Bea's predicate for `salary`, in Alex's module. First, Fig. 5.9a shows the predicate to calculate Bea's salary. Note that, as we mentioned before, the uncertainty in the age of Bea is modeled by using an even loop. As a consequence, for the query `?-salary(bea,Salary)` we would obtain two models:

(i) `{salary(bea,1000),base_salary(bea,900),generational_renewal(bea,100), not over_40_bea,neg_over_40_bea,holiday_worked(bea,0)}`.

(ii) `{salary(bea,900),base_salary(bea,900),generational_renewal(bea,0), over_40_bea,not neg_over_40_bea,holiday_worked(bea,0)}`.

Second, Fig. 5.9b shows the result of forgetting the private predicates in Alex's module $set_{Alex} = \{$ `base_salary/2`, `generational_renewal/2`, `holiday_worked/2`, `over_40_bea/2`, `neg_over_40_bea/2` $\}$.

Note that now auxiliary predicates are required in order to forget those predicates due to the presence of the even loop (see (Fidilio-Allende and Arias 2024) for details). The models obtained from the new program are:

(i) `{salary(bea,1000),neg_1,not neg_2}`

(ii) `{salary(bea,900),neg_2,not neg_1}`

Note that the auxiliary predicates are necessary to preserve the possibility that Bea may be either younger or older than 40 years old, allowing different answers depending on the assumption of the reasoner.

Finally, Fig. 5.9c and 5.9d compare the justification of the original program vs. the justification of the program result of applying forgetting. In these justifications, we observe that both models are displayed: in one model it is considered that `over_40_bea` is false, i.e., there is no evidence that `over_40_bea` is true, and in the second the other way around. Moreover, in Fig. 5.9d, we observe that the justifications after using f_{CASP} conserve the even loop structure.

```
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.
```

(a) Bea_0 = Original program

```
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.
```

(b) $f_{CASP}(Bea_0, set_{Alex})$

```
% Answer 1
salary(bea,1000) :-
    base_salary(bea,900),
    generational_renewal(bea,100) :-
        not over_40_bea :-
            neg_over_40_bea :-
              chs(not over_40_bea).
    holiday_worked(bea,0),
    1000 is 900+100+0.

% Answer 2
salary(bea,900) :-
    base_salary(bea,900),
    generational_renewal(bea,0) :-
        over_40_bea :-
            not neg_over_40_bea :-
              chs(over_40_bea).
    holiday_worked(bea,0),
    900 is 900+0+0.
```

(c) Justification of original program.

```
% Answer 1
salary(bea,1000) :-
  neg_1 :-
      not neg_2 :-
          chs(neg_1).
  1000 is 900+100+0.



% Answer 2
salary(bea,900) :-
  neg_2 :-
      not neg_1 :-
          chs(neg_2).
  900 is 900+0+0.
```

(d) Justification after forgetting.

Figure 5.9: Bea's salary programs and justifications (original vs. forgetting).

Chapter 6

Conclusions and future work

6.1 Conclusions

We have presented a new forgetting operator, f_{CASP}, designed to work with goal-directed Answer Set Programs. Thanks to the use of the dual rules compiled by s(CASP), our proposal f_{CASP}: (i) is able to forget predicates in programs with even loops (also due to double negations), (ii) can be applied in an iterative and commutative way, i.e., regardless of the order, and (iii) its application results in programs that generate the same models as the originals (ignoring the forgotten and auxiliary predicates) while remove sensitive information from both the model and its justifications.

Additionally, in this work we have applied f_{CASP} to a real case, based on the admission of farmers to an energy community, and on the allocation of the generated energy using values as criteria. Since, automated systems for this use case are considered high-risk systems by the AI Act, among other obligations, such systems must provide clear and adequate explanations to users but at the same time they must comply with other regulations, such as Art. 63 of the Organic Law 1/2004 which states that in procedures related to gender violence, the privacy of the victims shall be protected.

6.2 Future work

These are preliminary results, and we have identified the following promising future lines of work for the future.

First, *supporting predicates and constraints:* Obtaining sound negated rules poses a challenge to forgetting in programs with predicates (and constraints) because

of the many complications that arise due to the use of variables. For example, when variables in the body of the clause do not appear in the head, the s(CASP) compiler introduces the `forall/2` predicate in the dual rules. Which could be interpreted by s(CASP), because it is part of the meta-interpreter. Similarly, we claim that f_{CASP} will be able to deal with them.

Second, *dealing with recursive predicates:* In propositional logic, the number of times an atom is called in a recursive function does not affect the result. In contrast, for programs that include recursion such as graph traversal (i.e., the well-known predicate `traversal/2`) it would be necessary to either ground the variables involved or follow a strategy similar to unfolding.

Third, *determining and proving formally which properties of the forgetting operators f_{CASP} can satisfy.* In particular, as can be seen from the examples, we believe that it could satisfy strong persistence (SP), the more relevant property since it preserves the semantics of the original program.

Finally, *splitting CASP programs based on its predicate stratification:* In general, a logic program can contain independent business rules, e.g., in the case of energy distribution, we may generate the requested submodule of the farmer's program automatically. In this case, to generate the strictly necessary model, we could apply splitting. However, in the case of ASP programs, it is necessary to also check the global constraint (both user-defined and required by the presence of odd loops).

Bibliography

Alviano, M., Faber, W., Greco, G., and Leone, N. (2012). **Magic Sets for Disjunctive Datalog Programs**. In: *Artificial Intelligence* 187, pp. 156–192. DOI: 10.1016/j.artint.2012.04.008. URL: https://doi.org/10.1016/j.artint.2012.04.008.

Arias, J., Carro, M., Chen, Z., and Gupta, G. (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.

Arias, J., Carro, M., Chen, Z., and Gupta, G. (2022a). **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.

Arias, J., Carro, M., Chen, Z., and Gupta, G. (2022b). **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.

Arias, J., Carro, M., Salazar, E., Marple, K., and Gupta, G. (2018). **Constraint Answer Set Programming without Grounding**. In: *Theory and Practice of Logic Programming* 18(3-4), pp. 337–354. DOI: 10.1017/S1471068418000285.

Arias, J., Moreno-Rebato, M., Rodriguez-García, J. A., and Ossowski, S. (2023). **Automated legal reasoning with discretion to act using s(LAW)**. In: *Artificial Intelligence and Law* 23, pp. 1–24. DOI: 10.1007/s10506-023-09376-5.

Berthold, M. (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.

Berthold, M., Gonçalves, R., Knorr, M., and Leite, J. (2019). **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.

Brewka, G., Eiter, T., and Truszczyński, M. (2011). **Answer Set Programming at a Glance**. In: *Communications of the ACM* 54(12), pp. 92–103.

Calimeri, F., Cozza, S., and Ianni, G. (2007). **External Sources of Knowledge and Value Invention in Logic Programming**. In: *Annals of Mathematics and Artificial Intelligence* 50(3-4), pp. 333–361. DOI: 10.1007/s10472-007-9076-z. URL: https://doi.org/10.1007/s10472-007-9076-z.

Calimeri, F., Ianni, G., Pacenza, F., Perri, S., and Zangari, J. (2024). **Forget and Regeneration Techniques for Optimizing ASP-Based Stream Reasoning**. In: *International Symposium on Practical Aspects of Declarative Languages*. Springer, pp. 1–17.

Clark, K. L. (1978). **Negation as Failure**. In: *Logic and Data Bases*. Ed. by H. Gallaire and J. Minker. Springer, pp. 293–322. DOI: 10.1007/978-1-4684-3384-5_11.

DARPA (2017). **Explainable Artificial Intelligence (XAI)**. Defense Advanced Research Projects Agency. https://www.darpa.mil/program/explainable-artificial-intelligence.

Eiter, T. and Kern-Isberner, G. (2019). **A Brief Survey on Forgetting from a Knowledge Representation and Reasoning Perspective**. In: *KI-Künstliche Intelligenz* 33, pp. 9–33.

Fidilio-Allende, L. and Arias, J. (2024). **f_{CASP}: A forgetting technique for XAI based on goal-directed constraint ASP models**. In: *XXIII Jornadas sobre Programación y Lenguajes (PROLE)*. URL: http://platon.etsii.urjc.es/~jarias/papers/forgotten-paams24/fidilio-forgetting-prole-24.pdf.

Gelder, A. V., Ross, K., and Schlipf, J. (1991). **The Well-Founded Semantics for General Logic Programs**. In: *Journal of the ACM* 38, pp. 620–650. DOI: 10.1145/116825.116838.

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

Goldreich, O. and Oren, Y. (1994). **Definitions and properties of zero-knowledge proof systems**. In: *Journal of Cryptology* 7(1), pp. 1–32.

Gonçalves, R., Janhunen, T., Knorr, M., and Leite, J. (2021). **On Syntactic Forgetting under Uniform Equivalence**. In: *European Conference on Logics in Artificial Intelligence*. Springer, pp. 297–312.

Gonçalves, R., Knorr, M., and Leite, J. (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.

Gonçalves, R., Knorr, M., and Leite, J. (2023). **Forgetting in Answer Set Programming–A Survey**. In: *Theory and Practice of Logic Programming* 23(1), pp. 111–156.

Gonçalves, R., Knorr, M., Leite, J., and Woltran, S. (2017). **When you must forget: Beyond Strong Persistence when Forgetting in Answer Set Programming**. In: *Theory and Practice of Logic Programming* 17(5-6), pp. 837–854.

Gupta, G. (2022). **Automating Common Sense Reasoning with ASP and s(CASP)**. Technical Report, https://utdallas.edu/~gupta/csr-scasp.pdf.

Hernandez-Matheus, A., Löschenbrand, M., Berg, K., Fuchs, I., Aragüés-Peñalba, M., Bullich-Massagué, E., and Sumper, A. (2022). **A systematic review of machine learning techniques related to local energy communities**. In: *Renewable and Sustainable Energy Reviews* 170, p. 112651.

Jain, P., Raina, G., Sinha, S., Malik, P., and Mathur, S. (2021). **Agrovoltaics: Step towards sustainable energy-food combination**. In: *Bioresource Technology Reports* 15, p. 100766.

Knorr, M. and Alferes, J. J. (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.

Kustepeli, Y., Gulcan, Y., Yercan, M., and Yıldırım, B. (2023). **The role of agricultural development cooperatives in establishing social capital**. In: *The Annals of Regional Science* 70(3), pp. 681–704.

Lifschitz, V., Tang, L. R., and Turner, H. (1999). **Nested expressions in logic programs**. In: *Annals of Mathematics and Artificial Intelligence* 25, pp. 369–389. DOI: 10.1023/A:1018978005636.

Lin, B., Wang, X., Jin, S., Yang, W., and Li, H. (2022). **Impacts of cooperative membership on rice productivity: Evidence from China**. In: *World Development* 150, p. 105669.

Montes, N., Osman, N., Sierra, C., and Slavkovik, M. (2023). **Value Engineering for Autonomous Agents**. In: *CoRR* abs/2302.08759. DOI: 10.48550/arXiv.2302.08759.

Osman, A. I., Chen, L., Yang, M., Msigwa, G., Farghali, M., Fawzy, S., Rooney, D. W., and Yap, P.-S. (2023). **Cost, environmental impact, and resilience of renewable energy under a changing climate: a review**. In: *Environmental Chemistry Letters* 21(2), pp. 741–764.

Roberts, M. B., Sharma, A., and MacGill, I. (2022). **Efficient, effective and fair allocation of costs and benefits in residential energy communities deploying shared photovoltaics**. In: *Applied Energy* 305, p. 117935.

Swift, T. and Warren, D. S. (2012). **XSB: Extending Prolog with Tabled Logic Programming**. In: *Theory and Practice of Logic Programming* 12(1-2), pp. 157–187. DOI: 10.1017/S1471068411000500.

Woltran, S. (2004). **Characterizations for Relativized Notions of Equivalence in Answer Set Programming**. In: *European Workshop on Logics in Artificial Intelligence*. Springer, pp. 161–173.

Wong, K.-S. (2009). **Forgetting in logic programs**. PhD thesis. UNSW Sydney.