Justifications for Goal-Directed Constraint Answer Set Programming[*]

Joaquín Arias[1] Manuel Carro[1,2] Zhuo Chen[3] Gopal Gupta[3]

[1]IMDEA Software Institute
[2]Universidad Politécnica de Madrid
joaquin.arias@imdea.org
manuel.carro@{imdea.org,upm.es}

[3]University of Texas at Dallas
{zhuo.chen,gupta}@utdallas.edu

Ethical and legal concerns make it necessary for programs that may directly influence the life of people (via, e.g., legal or health counseling) to *justify* in human-understandable terms the advice given. Answer Set Programming has a rich semantics that makes it possible to very concisely express complex knowledge. However, justifying why an answer is a consequence from an ASP program may be non-trivial — even more so when the user is an expert in a given domain, but not necessarily knowledgeable in ASP. Most ASP systems generate answers using SAT-solving procedures on ground rules that do not match how humans perceive reasoning. We propose using s(CASP), a query-driven, top-down execution model for predicate ASP with constraints to generate justification trees of (constrained) answer sets. The operational semantics of s(CASP) relies on backward chaining, which is intuitive to follow and lends itself to generating explanations that are easier to translate into natural language. We show how s(CASP) provides minimal justifications for, among others, relevant examples proposed in the literature, both as search trees but, more importantly, as explanations in natural language. We validate our design with real ASP applications and evaluate the cost of generating s(CASP) justification trees.

1 Introduction and Motivation

Since the European Union approved the General Data Protection Regulation (GDPR) [7], every person affected by a decision made by an automated process has the right to obtain an explanation of the decision reached and to challenge the decision (Art. 71). Therefore, the uptake of current Artificial Intelligence (AI) systems is restricted by their limitation to explain their decisions in a way amenable to human understanding. For example, AI systems based on machine learning may give accurate outcomes, but they work as a black box without providing an intuitive high-level description of how they reach a decision. It is therefore hard for users and/or programmers to understand or verify the principles that govern them. In the context of Explainable Artificial Intelligence [6] (XAI), the USA Department of Defense (DoD) is facing challenges that demand XAI to understand, appropriately trust, and effectively manage an emerging generation of artificially intelligent machine partners.

Answer Set Programming (ASP) is a successful paradigm for developing intelligent applications and has attracted much attention due to its expressiveness, ability to represent knowledge, incorporate non-monotonicity, and model combinatorial problems. ASP uses the stable model semantics [12] for programs with negation. It is a declarative paradigm where the programmer

[*]Work partially supported by MINECO project TIN2015-67522-C3-1-R (TRACES), MICINN project PID2019-108528RB-C21 (ProCode), Comunidad de Madrid project S2018/TCS-4339 BLOQUES-CM co-funded by EIE Funds of the European Union, NSF awards IIS 1718945, IIS 1910131, IIP 1916206, and a DARPA award.

Submitted to:
ICLP 2020

specifies rules that describe the problem to be solved and the ASP system computes the solution. The solution of the program is an answer set that satisfies the rules of the program under the stable model semantics. Most ASP systems follow bottom-up executions that require a grounding phase where the variables of the program are replaced with their possible values. During the grounding phase, links between variables are lost and therefore an explanation framework for these systems must face many challenges to provide a concise justification of why a specific answer set satisfies the rules (and which rules). Some of the most relevant approaches to justification are:

- Off-line and on-line justifications [16] provide a graph-based explanation of the truth value (i.e., true, false, or assume) of a literal. The explanation *assume* is used for literals whose truth value is not being requested. This proposal provides a basic theory to support debugging of ASP programs.
- Causal Graph Justification [2] explains why a literal is contained in an answer set, but not why a negated literal is not contained. It can be used to formalize and reason with causal knowledge (i.e., it relies on a causal interpretation of rules and the idea of causal chain).
- Labeled ABA-Based Answer Set Justification, LABAS [17] explains the truth value of an extended literal with respect to a given answer set. A literal is in the answer set if a derivation of this literal is supported and is not in the answer set if all derivations of this literal are "attacked". I.e., a literal is explained in terms of the (negated) literals necessary for its derivations, rather than in terms of the whole answer set.

However, these approaches are applied to grounded versions of the programs and they may produce unwieldy justifications. The explanation of answers from ASP programs with constraint (both under discrete or dense domains) is even more challenging since the grounding phase removes the relationships among constrained variables, in the case of dense domains, because of the difficulty of representing the ranges of the variables.

On the other hand, systems that follow a top-down execution can trace which rules have been used to obtain the answers more easily. One such system is ErgoAI (`https://coherentknowledge.com`), based on XSB [19], that generates justification trees for programs with variables. ErgoAI has been applied to analyze streams of financial transactions 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 [11] and therefore ErgoAI is not a framework that can explain the answers from ASP programs.

In this work we propose the use of s(CASP) [1], a goal-directed interpreter for ASP with constraints (CASP), to explain the results of CASP programs. The top-down evaluation does not require a grounded program. Moreover, the execution of an s(CASP) program returns partial stable models that are the relevant subsets of the ASP stable models which include only the (negated) literals necessary to support the initial query. To the best of our knowledge, s(CASP) is the only system that exhibits the property of relevance [15].

This paper shows how s(CASP) can use top-down evaluation trees to generate minimal justifications where it is possible to control which literals should appear. We also provide a mechanism to present the justifications with natural language using a generic translation and the possibility of customizing it with directives that provide translation patterns. Both plain text and user-friendly, expandable HTML can be generated. Additionally, the translation into natural language can be used with the program text itself, thereby making it easier for experts without a programming background to understand both the program and the results of its execution.

We validate the design and the expressiveness of the s(CASP) justification trees with examples from the literature and with complex applications such as a Physician Advisory System for Chronic Heart Failure Management. Finally, we evaluate the cost of generating the s(CASP) justification trees. All the program and files used or mentioned in this paper are available at `http://www.cliplab.org/papers/sCASP-ICLP2020/`.

2 Background: s(CASP)

Answer Set Programming is a logic programming and modelling language that evaluates normal logic programs under the stable model semantics [12]. s(CASP) [1], based on s(ASP) [14], is a top-down, goal-driven ASP system, that can evaluate Constraint ASP programs with function symbols (*functors*) and constraints **without** grounding them either before or during execution. Grounding is a procedure that substitutes program variables with the possible values from their domain. For most classical ASP solvers, grounding is a necessary pre-processing phase. Grounding, however, requires program variables to be restricted to take values in a finite domain. As a result, modeling continuous change is challenging for ASP solvers, while it is easier for s(CASP). Constraints improve both expressiveness and efficiency in logic programming. As a result, s(CASP) is more expressive and faster than s(ASP).

An s(CASP) program is a set of clauses of the form:

$$\texttt{h :- } \texttt{c}_1\texttt{, ..., c}_l\texttt{, b}_1\texttt{, ..., b}_m\texttt{, not b}_{m+1}\texttt{, ..., not b}_n\texttt{.}$$

where `h` and $\texttt{b}_1\texttt{, ..., b}_n$ are atoms, `not` corresponds to *default* negation, and $\texttt{c}_i\texttt{, ..., c}_l$ are constraints in some constraint system.

In s(CASP), and unlike Prolog's negation as failure, `not p(X)` is a constructive negation that can return bindings for `X` on success. Unlike in ASP, the variable `X` does not need to appear in any positive literal. Both are possible because s(CASP) resolves negated atoms $\texttt{not b}_i$ against *dual rules* of the program [1, 14]. Non-ground calls to `not p(X)` return the bindings for which the call `p(X)` would have failed. We summarize here the synthesis of the dual of a logic program P: Clark's completion [5] is first performed and then De Morgan's laws are applied.

1. The i-th rule of the predicate defining p/n is $p_i(\vec{x}) \leftarrow B_i$ $(i=1,\ldots,k)$. B_i is a conjunction of positive $(b_{i.j})$ or negative $(\neg b_{i.j})$ literals. Let $\vec{y_i}$ be the list of variables that occur in the body B_i but do not occur in the head. The first-order formula corresponding to the rule is $\forall \vec{x}\ (p_i(\vec{x}) \leftarrow \exists \vec{y_i}\ B_i)$.
2. With the rules for a predicate we construct its Clark's completion:
$$\begin{array}{lll} \forall \vec{x}\ (\ p(\vec{x}) & \longleftrightarrow & p_1(\vec{x}) \vee \cdots \vee p_k(\vec{x})\ ) \\ \forall \vec{x}\ (\ p_i(\vec{x}) & \longleftrightarrow & \exists \vec{y_i}\ (b_{i.1} \wedge \cdots \wedge b_{i.m} \wedge \neg\ b_{i.m+1} \wedge \cdots \wedge \neg\ b_{i.n})\ ) \end{array}$$
3. Their semantically equivalent duals $\neg p/n$, $\neg p_i/n$ are:
$$\begin{array}{lll} \forall \vec{x}\ (\ \neg p(\vec{x}) & \longleftrightarrow & \neg(p_1(\vec{x}) \vee \cdots \vee p_k(\vec{x}))\ ) \\ \forall \vec{x}\ (\ \neg p_i(\vec{x}) & \longleftrightarrow & \neg\ \exists \vec{y_i}\ (b_{i.1} \wedge \cdots \wedge b_{i.m} \wedge \neg\ b_{i.m+1} \wedge \cdots \wedge \neg\ b_{i.n})\ ) \end{array}$$
4. Applying De Morgan's laws we obtain:
$$\begin{array}{lll} \forall \vec{x}(\ \neg p(\vec{x}) & \longleftrightarrow & \neg p_1(\vec{x}) \wedge \cdots \wedge \neg p_k(\vec{x})\ ) \\ \forall \vec{x}\ (\neg p_i(\vec{x}) & \longleftrightarrow & \forall \vec{y_i}\ (\neg b_{i.1} \vee \cdots \vee \neg\ b_{i.m} \vee\ b_{i.m+1} \vee \cdots \vee\ b_{i.n})\ ) \end{array}$$

This provides a definition for $\neg p(\vec{x})$ via a clause with head $\neg p_i(\vec{x})$ for each original clause with head $p_i(\vec{x})$. A construction (`C-forall`) to implement the universal quantifier introduced in the body of the dual program [1, Section 3.4] is provided by the metainterpreter.

Definitions for negated literals in P and for each of the newly negated literals are thus synthesized. At the end of the chain, constraints are negated and handled by the constraint solver. After removing explicit quantifiers (as in Horn clauses), the dual program is then executed in a top-down fashion by a meta-interpreter [1, 14] which does not implement SLD semantics. Instead, its main distinguishing points are:

Global constraints: The s(CASP) compiler automatically generates a `global_constraint` predicate that captures all the global constraints written by the programmer (e.g., constructions of the form `:- p, q` to express that `p` and `q` cannot hold simultaneously). This goal is always executed by the metainterpreter to ensure that models are consistent.

Loop handling: Two different cases are distinguished.

- When a call eventually invokes itself and there is an odd number of intervening negations (as in, e.g., `p:- q. q:- not r. r:- p.`), the evaluation fails (and backtracks) to avoid contradictions of the form $p \wedge \neg p$.
- When there is an even number of intervening negations, as in `p:- not q. q:- r. r:- not p.` the metainterpreter generates two models, `{p, not q, not r}` and `{q, r, not p}`.

Additionally, the s(CASP) compiler detects statically rules of the form `r :- q, not r.` and introduces global constraints to ensure that the models satisfy $\neg q \vee r$, even if the literals `r` or `q` are not needed to solve the query. Therefore, s(CASP) will state that the program

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

has no stable models, regardless of the initial query.

In addition to default negation, s(CASP) supports classical negation to capture the explicit evidence that a literal is false. Classical negation is defined with the prefix `'-'`, e.g., `-p(a)` would mean that it is not the case that `p(a)` is true. When a literal `p(X)` and its explicit negation `-p(X)` appear in a program, s(CASP) automatically adds the constraint `:- -p(X),p(X)` to ensure that they are not simultaneously true.

The execution of an s(CASP) program starts with a *query* of the form `?-` $\texttt{c}_q$, $\texttt{l}_1$, …, $\texttt{l}_m$, where $\texttt{l}_i$ are (negated) literals and $\texttt{c}_q$ is a conjunction of constraint(s). This is followed by the evaluation of the global constraints written by the user or introduced by the compiler. The answers to the query are partial stable models where each one is a subset of a consistent stable model [12] including only the literals necessary to support the query (see [1] for details). Additionally, for each partial stable model s(CASP) returns on backtracking the justification tree and the bindings for the free variables of the query that correspond to the most general unifier (*mgu*) of a successful top-down derivation consistent with this stable model.

s(CASP), available at `https://gitlab.software.imdea.org/ciao-lang/scasp`, is implemented in Ciao Prolog [13].

3 Justification Trees of Goal-Directed Evaluations

We will show how s(CASP) execution trees can provide concise justifications for the models supporting a query and how they address several challenges [8] arising in the context of XAI.

```
opera(D) :- not home(D).
home(D) :- not opera(D).
home(monday).

% QUERY:
?- opera(A).

% BINDINGS:
A \= monday
% MODEL:
{ opera(A|{A \= monday}), not home(A|{A \= monday}) }
```

```
% EVALUATION TREE:
opera(A|{A \= monday}) :-
    not home(A|{A \= monday}) :-
        not o_home_1(A|{A \= monday}) :-
            chs(opera(A|{A \= monday})).
        not o_home_2(A|{A \= monday}) :-
            A \= monday.
global_constraint.
```

Figure 1: Code, query, bindings, model, and evaluation of `opera.pl`.

Definition 1 (Minimal s(CASP) Justification Tree) *Let Q be a query to a CASP program P. The s(CASP)* Justification Tree *of a (partial) model of P is the ordered list of literals in the path of the successful goal-driven proof of Q, conjoined with that of the proof of the global constraint under the operational semantics of s(CASP).*

Note that for every model returned by s(CASP), its *Justification Tree* is *minimal* because it only contains the literals needed to support Q and the consistency of the global constraints in P, i.e., the subset of literals contained in a stable model that are relevant to Q.
Let us describe how an s(CASP) justification tree is to be understood. We will follow the evaluation of an example query under s(CASP).

Example 1 *The program in Fig. 1, adapted from [9], represents Bob's plans. Bob either goes to the opera or stays home. Bob will stay at home on Monday and therefore the query `?- opera(A)` returns a partial stable model with the constraint `A \=monday`, meaning that `opera(A)` is valid for any `A` except `monday`.*
Constrained/unbound variables can appear in the top-down derivation steps, shown on the right. The notation `opera(A|{A \=monday})` expresses that `opera(A)` is valid when $A \neq$ `monday`. Following the evaluation trace, the query `?- opera(A)` holds if `not home(A)` does. Being a negative literal, it needs to be resolved against the dual[1] of `home/1`:

```
not home(A) :- not o_home_1(A),
               not o_home_2(A).
```

```
not o_home_1(A) :- opera(A).
not o_home_2(A) :- A \= monday.
```

`not home(A)` succeeds because the two goals in its body succeed: (i) `not o_home_1(A)` holds because s(CASP) detects that `opera(A)` was called through an even number of intervening negations. This is marked by wrapping it with `chs/1`. (ii) `not o_home_2(A)` succeeds because `A` is a free variable and applying the constraint `A \=monday` succeeds. This constraint is automatically propagated to the initial variable. Finally, since there are no global constraints to be checked, a partial model is generated, including the constraint `{A \=monday}`.

Dual predicates, such as `not o_home_1(A)`, are synthesized by the compiler and therefore not immediately known by the user. While it is possible to control which predicates appear in the representation of an evaluation tree (Section 3.4), they are necessary to fully understand all the details of a justification. The s(CASP) compiler can output the original program together with its dual to help in this aspect.

[1]The dual of `opera.pl` is available at `opera_dual.pl` for the reader's convenience.

3.1 Partial Models, Constraints, and Justifications

Conciseness usually makes understanding easier; the goal-directed strategy of s(CASP) contributes to concise justifications:

- s(CASP) computes partial stable models that contain only the literals relevant to a query. Accordingly, s(CASP) justification trees only need to address the literals that appear in the partial model.
- When there is more than one derivation tree that supports the query, justifications are given separately. If only one is necessary, we do not have to compute all of them.
- Additionally, constraints can concisely and finitely represent infinitely many models.

Example 2 (Cont. Example 1) *We adapted the program in Fig. 1 to evaluate Example 1 under* clingo *[10]. A call to a domain predicate for the seven days of the week was added to the clauses in lines 1 and 2 to ensure that they are safe. Thanks to the use of constraints, s(CASP) returns a single model; however, clingo returns 64 models. Even with the more restrictive query* `?- opera(tuesday)`*, clingo still generates 32 models. As a consequence, an explanation framework for clingo such as* `xclingo` *[3], based on Causal Graph Justifications [2], takes 10 times longer than s(CASP) to justify the results of* `opera.pl` *because it needs to process every single model.*

3.2 Justifications of Negated Literals

Real-life domains often use rules such as "*…when it is not the case that…*". In these cases, justifying why a literal is not included in a model is necessary, especially in a non-monotonic reasoning framework such as ASP. In s(CASP), classically-negated literals (those with rules specifying when they do not hold) are evaluated and justified similarly to positive literals, with the proviso that global constraints are added to ensure the consistency of the model. For default-negated literals, s(CASP) uses the dual program to constructively determine when the default negation of a literal succeeds using the top-down evaluation mentioned in Section 2.

For simplicity, we will use a propositional program to illustrate the justifications of negated predicates. Note that thanks to the top-down evaluation and the usage of dual programs, s(CASP) can combine justifications for constrained, non-ground evaluations (as in Fig. 1) with justifications for negated literals.

Example 3 *Consider an ophthalmologist diagnosis system [17] (Fig. 2) that encodes knowledge about optical treatments. For a given patient, Peter,* `intraocularLens` *are recommended. Fig. 2, bottom, shows the partial model where the negated literals correspond to disrecommended treatments; these contribute to making the model more understandable and also increase trust on the system's advice, because a doctor can check which treatments have been discarded. The s(CASP) justification tree (Fig. 2, right, and at* `peter.html`*) justifies this recommendation. To help understand it, we include here the dual of* `laserSurgery` *as generated by s(CASP):*[2]

```
not o_laserSurgery_1 :- not shortSighted.
not o_laserSurgery_1 :- shortSighted, tightOnMoney.
not o_laserSurgery_1 :- shortSighted, not tightOnMoney, correctiveLens.
```

[2]The dual of the ophthalmologist system is available at `peter_dual.pl`.

```
% Treatments
intraocularLens :-
    correctiveLens,
    not glasses,
    not contactLens.
laserSurgery :-
    shortSighted,
    not tightOnMoney,
    not correctiveLens.
glasses :-
    correctiveLens,
    not caresPracticality,
    not contactLens.
contactLens :-
    correctiveLens,
    not afraidToTouchEyes,
    not longSighted,
    not glasses.
```

```
% Auxiliar
correctiveLens :-
    shortSighted,
    not laserSurgery.
tightOnMoney :-
    student,
    not richParents.
caresPracticality :-
    likesSports.

% Peter, a patient
shortSighted.
student.
likesSports.
afraidToTouchEyes.

% QUERY:
?- intraocularLens.
```

```
% JUSTIFICATION_TREE:
intraocularLens :-
   correctiveLens :-
      shortSighted,
      not laserSurgery :-
        not o_laserSurgery_1 :-
           proved(shortSighted),
           tightOnMoney :-
              student,
              not richParents.
   not glasses :-
      not o_glasses_1 :-
        proved(correctiveLens),
        caresPracticality :-
           likesSports.
   not contactLens :-
      not o_contactLens_1 :-
        proved(correctiveLens),
        afraidToTouchEyes.
```

```
% MODEL:
{ intraocularLens,  correctiveLens,  shortSighted,  not laserSurgery,  tightOnMoney,
student, not glasses, caresPracticality, likesSports, not contactLens, afraidToTouchEyes }
```

Figure 2: Code, query, model, and justification of an ophthalmologist system.

Strictly speaking, clause 2 would only need to be `not o_laserSurgery_1 :- tightOnMoney`*. However, including* `shortSighted`*, which appears negated in the previous clause, avoids unnecessary search when generating models, and therefore it also helps remove redundant justifications.*[3] *Clause 3 belongs to a similar case.*

Note that although there are two ways of justifying `not laserSurgery` *(i.e., using* `tightOnMoney` *or* `correctiveLens`*), only one is necessary and generated. As mentioned before, the wrapper* `proved/1` *marks literals already proven whose justification sub-tree can be omitted.*

Some proposals such as LABAS [17] justify all the negated literals, which may provide too much detail. Others, such as Causal Graph Justifications [2], treat negated literals as assumptions and give no justification for them. s(CASP) provides a balance as it only justifies the negated literals needed to support the answer to the query and avoids generating redundant justifications thanks to the way dual rules are generated. Note that reducing the number of unnecessary justifications is relevant in practice: justifying the literals that do not support the query may increase the number of justifications (and the execution time) exponentially w.r.t. size of the program [8].

Additionally, the s(CASP) implementation includes ways to control which literals should appear in the justification tree (Section 3.4). This makes it possible to tailor the contents and size of the justification tree to better adapt it to specific cases.

[3]Generating justification trees for all the possible derivations boils down to not producing the optimized code shown above. Since this requires additional search and execution time, it ought to be a command-line option.

```
% JUSTIFICATION_TREE:
opera(A|{A \= monday,A \= tuesday}) :-
  not home(A|{A \= monday,A \= tuesday}) :-
    not o_home_1(A|{A \= monday,A \= tuesday}) :-
      chs(opera(A|{A \= monday,A \= tuesday})).
    not o_home_2(A|{A \= monday,A \= tuesday}) :-
      A \= monday.
global_constraint :-
  not o_chk :-
    not o_chk_1 :-
```

```
forall(C,not o_chk_1(C)) :-
  not o_chk_1(B|{B \= tuesday}) :-
    not baby(B|{B \= tuesday}) :-
      not o_baby_1(B|{B \= tuesday}) :-
        B \= tuesday.
  not o_chk_1(tuesday) :-
    baby(tuesday),
    not opera(tuesday) :-
      not o_opera_1(tuesday) :-
        home(tuesday) :-
          chs(not opera(tuesday)).
```

Figure 3: s(CASP) justification of `opera.pl` with a global constraint.

3.3 Justifications of Global Constraints

Global constraints are introduced by the s(CASP) compiler (Section 2) to ensure model consistency in the presence of classically-negated literals and of rules such as `p :- q, not p`. Consistency rules written by the user are also added to the global constraints. Therefore, it is necessary to generate justifications to explain how models are compliant with global constraints.

Example 4 (Cont. Example 1) *Let us modify the code in Fig. 1 by adding:*

```
:- baby(D), opera(D).   % When Bob's best friend comes with her baby, it is
                        % not a good idea to take the baby to the opera.
 baby(tuesday).         % They come on Tuesday.
```

The answer for `?- opera(A)` is now `A \= monday, A \= tuesday` because Bob always stays home on Monday and Bob's best friend comes with her baby on Tuesday. Fig. 3 shows the s(CASP) justification tree including the global constraint checks. The first part of the justification tree is similar to that in Fig. 1, left, with the addition that variable `A` is constrained not to be equal to `tuesday`. To follow the justification of the user constraints, let us remember that `:- baby(D), opera(D).` is $\forall x \cdot \neg(baby(x) \land opera(x))$. This is compiled into

```
not o_chk_1 :-
   forall(D, not o_chk_1(D)).
```

```
not o_chk_1(D) :- not baby(D).
not o_chk_1(D) :- baby(D), not opera(D).
```

where the compiler introduced the predicate `forall/2` [1, Section 3.4] that succeeds if for all possible values of `D` present in the program, the predicate `not o_chk_1(D)` holds. The justification tree shows that:

- *`not o_chk_1(B)` succeeds for `B \= tuesday` because `not baby` succeeds for `B \= tuesday`.*
- *`not o_chk_1(tuesday)` holds because `baby(tuesday)` is true and `not opera(tuesday)` succeeds (the variable `A` of `opera/1` can be restricted to be different from `tuesday`). The resulting constraint is `{A \= monday, A \= tuesday}`.*

3.4 Controlling the Literals that Appear in an s(CASP) Justification Tree

ASP systems (and also s(CASP)) provide mechanisms to control which literals should be shown in a model. Controlling which literals should appear in a justification tree is more challenging. A decision based on hierarchy and predicate dependencies is not satisfactory, as we observed that in some cases we may want to hide some literals but not their children in order to inspect the support for some conclusions. This means taking special care for some situations (e.g., having to remove repeated siblings) while providing an interface to facilitate a flexible control.

```
'opera' holds (for A), with A not equal monday, because
    there is no evidence that 'home' holds (for A), with A not equal monday, because
        'rule 1' holds (for A), because
            it is assumed that 'opera' holds (for A), with A not equal monday.
        'rule 2' holds (for A), because
            A is not equal monday.
```

(a) Justification in English using predefined patterns.

```
Bob goes to the opera on a day A not equal monday, because
    Bob does not stay at home on A not equal monday, because
        'rule 1' holds (for A), because
            it is assumed that Bob goes to the opera on a day A not equal monday.
        'rule 2' holds (for A), because
            A is not equal monday.
```

(b) Justification in English using user messages.

Figure 4: s(CASP) justification tree for `opera.pl` in natural language.

The current implementation supports three levels of detail (`short`, `mid`, `long`) plus an additional option (`neg`) that makes the interpreter include the default-negated literals.

`short` shows the (negated) literals selected with **`#show`** directives.

`mid` adds the rest of user-defined predicates (positive and/or classically negated).

`neg` (only with `short` or `mid`) shows also the default-negated versions of the displayed predicates.

`long` generates the complete s(CASP) justification tree, including auxiliary predicates, `forall`, and built-ins.

s(CASP) can generate justification trees as plain text or as expandable **`html`**, accessible through links in this paper. In each of these cases, the three options mentioned above are available.

4 Presenting s(CASP) Justification Trees using Natural Language

Justification trees lend themselves to the generation of natural language explanations quite directly. This is useful for domain experts that may not be familiar with logic rules. Predefined generic patterns are used to generate simple translations. Additionally, more involved explanations, tailored to each particular case, can be produced by adding natural-language explanations to the program. These offer a human-readable meaning of the program at the individual rule level and can control which literals have to be explained. They can be applied both to plain text and HTML generation.

4.1 Predefined Natural Language Patterns

Fig. 4a shows the s(CASP) pseudo-natural language justification for the program and query in Fig. 1 using only predefined patterns. Each line corresponds to a literal in the justification tree shown before and is generated as follows:

`name(Arg1, ..., Argn)`: $\boxed{\textit{'name' holds (for } arg_1, \ldots, \textit{ and } arg_n)}$ where arg_i are either the run-time value for argument i^{th} or a variable name. For constrained variables, the constraints are also translated and shown.

`not name/n`: $\boxed{\textit{there is no evidence that}}$, followed by the pattern for `name/n`.

Table 1: Translation results with and without user annotations.

Var. State	Mark	Translation	Mark	Translation
Free	`@(D)`	*D*	`@(D:day)`	*D, a day*
Constrained	`@(D)`	*D not equal monday*	`@(D:day)`	*a day D not equal monday*
Ground	`@(D)`	*monday*	`@(D:day)`	*the day monday*

`-name/n`: *it is not the case that*, followed by the pattern for `name/n`.

The neck `':-'` is translated into *'because'* and the comma `','` into *'and'*. The mark `chs(name/n)` (a loop with an even number of negations has succeeded) is translated into *'it is assumed that'* followed by the pattern for `name/n`, and the mark `proved(name/n)` (the literal has already been proved) is translated by adding *'already justified'* to the pattern for `name/n`. For brevity, we will skip the description of the translation patterns for `forall`, the built-ins, and the auxiliary predicates of the dual rules.

It is interesting to remark that, since the evaluation tree chains the application of program rules, these predefined patterns can also be applied to translate the program code into natural language — the only difference is that the neck `':-'` is translated into *'if'*. The translation of the dual program of `opera.pl` from Example 1 (including the dual rules and the global constraints) is available at `opera_dual_NL.pl` for the reader's convenience.

4.2 User-Defined Natural Language Patterns

Customizing the predefined patterns makes it possible to better describe the meaning of the predicates. Our framework provides this possibility by allowing structured comments to be added to the literals (either positive or negative) of the programs.[4] Let us consider the two following examples:

```
#pred opera(D) :: 'Bob goes to the opera on @(D:day)'.
#pred not home(D) :: 'Bob does not stay at home on @(D)'
```

Each explanation of a literal is introduced with the directive `#pred` followed by the (negated) literal and a pattern indicating how to translate it. The marks `@(D)` indicates where the values of the arguments of the literals should appear. A qualification such as `@(D:day)` gives additional information on the meaning of the variable that is used to generate a more informative message depending on the instantiation state of the variable. Table 1 shows an intuitive explanation.

Messages can be defined for different instantiation patterns of the same literal, as in

```
#pred is(pregnancy) :: 'the patient is pregnant or planning to get pregnant'.
#pred is(stage(S)) :: 'the patient is in ACCF stage @(S)'.
#pred is(E) :: 'the patient is @(E)'.
```

Literals are scanned from top to bottom and the message associated with the first matching head is used. That makes it possible, for example, to write different explanations for every clause of a predicate (from more particular to more general), as it is common (see the case above) that every clause treats a different case. Translations for default-negated and classically-negated literals can also be provided.

[4]There other proposals, similar in spirit, that allow adding information to programs without changing their meaning, e.g. those of Ciao Prolog [13] or Clingo [10].

```
the treatment ace_inhibitors has been chosen, because
    it is a recommendation to use ace_inhibitors, because
        the patient is in ACCF stage C, and
        the patient is diagnosed with heart failure with reduced ejection fraction, because
            there is a measurement of lvef of 0.35.
        there is no evidence that there is a danger in taking ace_inhibitors, because
            there is no evidence that the patient has a history of angioedema, and
            there is no evidence that the patient is pregnant or planning to get pregnant.
    there is no evidence that the treatment ace_inhibitors is excluded, because
        the treatment diuretics is concomitant if ace_inhibitors is chosen, and
        it is a recommendation to use diuretics, because
            the patient is in ACCF stage C, justified above, and
            the patient is diagnosed with heart failure with reduced ejection fraction, justified above, and
            there is no evidence that there is a danger in taking diuretics.
        the treatment diuretics has been chosen, because
            it is a recommendation to use diuretics, justified above, and
            there is no evidence that the treatment diuretics is excluded, and
            there is no evidence that diuretics is discarded.
        the treatment aldosterone_antagonist is incompatible with ace_inhibitors, and
        aldosterone_antagonist is discarded, and
        the treatment arbs is incompatible with ace_inhibitors, and
        arbs is discarded.
    there is no evidence that ace_inhibitors is discarded.
The global constraints hold, because
    the global constraint number 1 holds.
```

Figure 5: s(CASP) justification tree of the Physician Advisor System.

Fig. 4b shows the s(CASP) justification tree for `opera.pl` (Fig. 1) using user-defined explanations. The complete s(CASP) justification tree and the dual program, translated following the user-defined patterns, are available at `opera_just_userNL.pl` and `opera_dual_userNL.pl`.

s(CASP) generates natural language explanations with the command-line option `--human`. The `short` version also shows the (negated) literals with user-defined predicates, even if they are not selected in the `#show` directives.

5 A Real Application: the Physician Advisor System

We have evaluated the s(CASP) justification generation with the Physician Advisor System [4] (PAS), an ASP program that recommends treatment choices for chronic heart failure (CHF). The system encodes near 80 pages of rules into an ASP program. Medical advisory systems are relevant applications in the context of XAI, and the management of chronic diseases, such as CHF and others, is a major problem in health care. The PAS has a modular implementation:

- The main module of the PAS (available at `PAS_rules.pl`) implements the knowledge patterns described in [4].
- The guide module (available at `PAS_guide.pl`) implements the CHF Guide.
- The patient module (available at `PAS_profile.pl`) contains a patient profile: demographic data, assessment evidences, particular contraindications, previous diagnosis, illness, medication history, and different measurements.

To provide readable recommendations, we extended PAS with explanations for the relevant predicates (available at PAS_rules.pred.pl, PAS_guide.pred.pl, and PAS_patient.pred.pl). With them, s(CASP) can generate concise justification trees in natural language.

5.1 Expressiveness of Justifications

Let us study one example: for the query `?- chose(ace_inhibitor)` there is a unique minimal stable model. Fig. 5 shows the natural language version of the s(CASP) justification tree for that query.[5] The following points are worth mentioning:

- The justification tree supports the choice of `ace_inhibitor` based on evidence (positive literals), e.g., "the patient is in ACCF stage C", and on the absence of counter-evidence (negated literals), e.g., "there is no evidence that there is a danger in taking ace_inhibitors". As mentioned before, providing justifications of negated literals is important because a doctor may want to double-check them.
- The justification includes only the recommendations and choices of treatments that are required to support the choice of `ace_inhibitor`, e.g., the treatment with `diuretics` is chosen because it is concomitant when `ace_inhibitor` is chosen.
- The justification also explains that some treatments have been discarded due to incompatibilities. This information is essential since it would avoid future errors by preventing the use of `aldosterone_antagonist` or `arbs`.
- s(CASP) supports dense domains which can directly represent physical quantities, such as the measurements of *lvef* (`0.35` in this example). This makes it possible to directly encode existing regulations and use these measurements in the justifications.

We compared s(CASP) with **`xclingo`** [3]. **`xclingo`** does not generate justifications for negated literals. The justification of the literal `chose(ace_inhibitors)` (generated using the `%!show_trace` directive for `chose(T)` and `discarded(T)`) is shown in Fig. 6.

```
>> chose(ace_inhibitors)
   |__recommendation(ace_inhibitors)
      |__reason(ace_inhibitors)
         |__evidence(accf_stage_c)
         |__diagnosis(hf_with_reduced_ef)
            |__measurement(lvef,35)
```

Figure 6: Fragment of **`xclingo`** justification.

It can be noted that the information related to negated literals, such as `not contraindication/1` and `not exclude/1`, is absent. As a consequence, the justification generated by **`xclingo`** does not fully explain why `diuretics` is chosen or why `aldosterone_antagonist` and `arbs` are discarded in every model.[6] Since clingo does not support decimals, the values of *lvef* have been normalized.

5.2 Performance Evaluation

We have evaluated the additional cost of generating justifications in s(CASP) and also compared the performance of our approach with that of other similar systems using, again, the PAS system. We used a Debian 4.19 machine with a Xeon E5640 at 2.6GHz.

[5] Other relevant queries to the PAS are listed in PAS_query.pl and at ace_inhibitors.html, beta_blockers.html, and anticoagulation.html.

[6] The **`xclingo`** justification for one of the models for the query `?- chose(ace_inhibitors)`, including the justification of every literal `chose/1` and `discarded/1` in the answer set, is available at PAS_xclingo_just.pl.

Table 2: Performance comparison of w.r.t. evaluation w.o. justification tree.

			`--html`	
	`--plain`	`--human`	`--plain`	`--human`
`--tree --short`	1.06	1.06	1.12	1.12
`--tree --short --neg`	1.06	1.07	1.12	1.13
`--tree --mid`	1.06	1.06	1.12	1.12
`--tree --mid --neg`	1.06	1.07	1.12	1.13
`--tree --long`	1.07	1.08	1.14	1.15

We used the query `?- chose(ace_inhibitors)` to test the performance of the generation of justifications. Generating justifications in s(CASP) has a very small impact on execution time which we show in Table 2 as speed-down (the larger the number, the slower the execution) and which is, at most, a 15% slower, which can be considered perfectly acceptable. This is also true for absolute speed, since the execution without justifications takes 340 ms.

For the same query, the program has 1024 models, because treatments unrelated with the query may be selected or not. This makes `xclingo` need 1'53" to justify selected and discarded treatments, while s(CASP) uses only 0.35 seconds, as only one partial model has to be generated.

6 Conclusion

We showed how s(CASP) can generate justifications for Constraint Answer Set programs, preventing generating excessively many justifications thanks to its ground-free, top-down evaluation strategy and the use of constraints. It can also justify negated literals and global constraints.

To the best of the authors' knowledge, this approach is the only one that can provide full justifications in natural language for ASP programs, including constraints and negated literals. Our approach also makes it possible to control which literals should appear in the justification tree, improving the readability of the justifications.

As part of our future work, we want to explore how s(CASP) justifications can be used to bring XAI principles to a class of knowledge representation and reasoning systems, namely those based on non-monotonic logic with a stable model semantics. These can range from rule-based systems capturing expert knowledge [4] to ILP systems that generate ASP programs [18] or concurrent imperative programs based on behavioral, observable specifications [20].

References

[1] Joaquín Arias, Manuel Carro, Elmer Salazar, Kyle Marple & Gopal Gupta (2018): *Constraint Answer Set Programming without Grounding. Theory and Practice of Logic Programming* 18(3-4), pp. 337–354, doi:10.1017/S1471068418000285. Available at `https://arxiv.org/abs/1804.11162`.

[2] Pedro Cabalar, Jorge Fandinno & Michael Fink (2014): *Causal Graph Justifications of Logic Programs. Theory and Practice of Logic Programming* 14(4-5), pp. 603–618, doi:10.1017/S1471068414000234.

[3] Pedro Cabalar, Jorge Fandinno & Brais Muñiz (2020): *A System for Explainable Answer Set Programming.* In Pedro Cabalar, Andreas Herzig, David Pearce, Torsten Schaub & Stefan Woltran, editors: *Declarative Problem Solving (ECAI 2020 Workshop).*

[4] Zhuo Chen, Kyle Marple, Elmer Salazar, Gopal Gupta & Lakshman Tamil (2016): *A Physician Advisory System for Chronic Heart Failure Management Based on Knowledge Patterns. Theory and Practice of Logic Programming* 16(5-6), pp. 604–618, doi:10.1017/S1471068416000429.

[5] Keith L. Clark (1978): *Negation as Failure.* In H. Gallaire & J. Minker, editors: *Logic and Data Bases*, Springer, pp. 293–322, doi:10.1007/978-1-4684-3384-5_11.

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

[7] European Union (2016): *General Data Protection Regulation (GDPR).* Regulation (EU) 2016/679 of the European Parliament and of the Council. https://eur-lex.europa.eu/eli/reg/2016/679/oj.

[8] Jorge Fandinno & Claudia Schulz (2019): *Answering the "Why" in Answer Set Programming - A Survey of Explanation Approaches. Theory and Practice of Logic Programming* 19(2), pp. 114–203, doi:10.1017/S1471068418000534.

[9] Alejandro Javier García, Carlos Iván Chesñevar, Nicolás D. Rotstein & Guillermo Ricardo Simari (2013): *Formalizing Dialectical Explanation Support for Argument-Based Reasoning in Knowledge-Based Systems. Expert Systems and Applications* 40(8), pp. 3233–3247, doi:10.1016/j.eswa.2012.12.036.

[10] Martin Gebser, Roland Kaminski, Benjamin Kaufmann & Torsten Schaub (2014): *Clingo = ASP + Control: Preliminary Report. arXiv* 1405.3694. Available at https://arxiv.org/abs/1405.3694.

[11] A. Van Gelder, K. Ross & J. Schlipf (1991): *The Well-Founded Semantics for General Logic Programs. Journal of the ACM* 38, pp. 620–650, doi:10.1145/116825.116838.

[12] Michael Gelfond & Vladimir Lifschitz (1988): *The Stable Model Semantics for Logic Programming.* In: *5th International Conference on Logic Programming*, pp. 1070–1080. Available at http://www.cse.unsw.edu.au/~cs4415/2010/resources/stable.pdf.

[13] M. V. Hermenegildo, F. Bueno, M. Carro, P. Lopez-Garcia, E. Mera, J.F. Morales & G. Puebla (2012): *An Overview of Ciao and its Design Philosophy. Theory and Practice of Logic Programming* 12(1–2), pp. 219–252, doi:10.1017/S1471068411000457. Available at http://arxiv.org/abs/1102.5497.

[14] Kyle Marple, Elmer Salazar & Gopal Gupta (2017): *Computing Stable Models of Normal Logic Programs Without Grounding. arXiv* 1709.00501. Available at http://arxiv.org/abs/1709.00501.

[15] Luís Moniz Pereira & Joaquim Nunes Aparício (1989): *Relevant Counterfactuals.* In: *EPIA 89, 4th Portuguese Conference on Artificial Intelligence, Lisbon, Portugal, September 26-29, 1989, Proceedings*, pp. 107–118, doi:10.1007/3-540-51665-4_78. Available at https://doi.org/10.1007/3-540-51665-4_78.

[16] Enrico Pontelli, Tran Cao Son & Omar El-Khatib (2009): *Justifications for Logic Programs under Answer Set Semantics. Theory and Practice of Logic Programming* 9(1), pp. 1–56, doi:10.1017/S1471068408003633.

[17] Claudia Schulz & Francesca Toni (2016): *Justifying Answer Sets Using Argumentation. Theory and Practice of Logic Programming* 16(1), pp. 59–110, doi:10.1017/S1471068414000702.

[18] Farhad Shakerin & Gopal Gupta (2019): *Induction of Non-Monotonic Logic Programs to Explain Boosted Tree Models Using LIME.* In: *AAAI 2019*, pp. 3052–3059, doi:10.1609/aaai.v33i01.33013052.

[19] Terrance Swift & David Scott Warren (2012): *XSB: Extending Prolog with Tabled Logic Programming. Theory and Practice of Logic Programming* 12(1-2), pp. 157–187, doi:10.1017/S1471068411000500.

[20] Sarat Chandra Varanasi, Elmer Salazar, Neeraj Mittal & Gopal Gupta (2019): *Synthesizing Imperative Code from Answer Set Programming Specifications.* In: *LOPSTR*, *Lecture Notes in Computer Science* 12042, Springer, pp. 75–89, doi:10.1007/978-3-030-45260-5_5.