Embedding s(CASP) in Prolog

Jan Wielemaker[1,2], Mikko Tiihonen[3]

[1]*SWI-Prolog Solutions b.v., Amsterdam, The Netherlands*
[2]*Vrije Universiteit Amsterdam, De Boelelaan 1105, 1081 HV Amsterdam, The Netherlands*
[3]*Forsante Oy, Helsinki, Finland*

Abstract
The original s(CASP) implementation is a stand-alone program implemented in Ciao Prolog. It reads the s(CASP) source using a dedicated parser, resolves the query embedded in the source and emits the results in a format dictated by commandline options. Typical applications require composing a s(CASP) program, solving a query and reasoning about the *bindings*, *model* and/or *justification*. This is often done in some external language, e.g., Python. In this paper we propose a closer integration with Prolog. The s(CASP) program is simply a Prolog program that has to respect certain constraints. The scasp library can be used to solve a query using s(CASP) semantics, making the bindings available in the normal Prolog way and providing access to the model and justification as Prolog terms. This way, we can exploits Prolog's power for manipulating (Prolog) terms for construction the s(CASP) program and interpreting the results.

Keywords
s(CASP), answer set programming, multi-paradigm, Prolog,

1. Introduction

ASP, s(ASP) and s(CASP) [1, 2, 3] are languages that deal with knowledge representation and reasoning. Lacking (easily) predictable ordering of resolution steps and facilities for input and output, these languages are not general purpose programming languages. This implies that for deployment in real-world applications we need a program written in a general purpose language to prepare the input for the s(CASP) solver and interpret the output. Traditionally the solver is written as a stand alone application or as a library that is connected to a scripting language such as Lua or Python. While these scripting languages have excellent infrastructure for interfacing with typical ICT infrastructure, they are poorly equipped for manipulating Prolog terms, in particular when these contain *logical variables*, optionally with *constraints*.

In contrast, Prolog is perfectly equipped for manipulating s(CASP) programs and the result of solving an s(CASP) query. This is already sufficient reason to use Prolog as s(CASP) scripting language for mostly self-contained tasks such as performing ILP (*Inductive Logic Programming*) or similar machine learning tasks using s(CASP). In addition, SWI-Prolog [4] provides a rich set of libraries to interface with ICT infrastructure. Examples are its HTTP server framework, STOMP and Redis interfaces and its I/O libraries for XML, HTML, RDF, JSON, YAML and more.

GDE'22
jan@swi-prolog.com (J. Wielemaker); mikko.tiihonen@forsante.com (M. Tiihonen)
0000-0001-5574-5673 (J. Wielemaker); 0000-0001-8773-5672 (M. Tiihonen)

Together with SWI-Prolog's scalable multi-threading this platform can provide scalable s(CASP) based services with pre- and postprocessing in systems based on a *micro service* architecture. A SWI-Prolog based s(CASP) reasoner can also be embedded in C++, Java, Python, Rust and several other languages.

This paper describes the result of embedding s(CASP) in SWI-Prolog as part of a commercial project applying s(CASP) at Forsante Oy for reasoning about medical guidelines. This paper first discusses the requirements for embedding s(CASP) in Prolog. In section 3 we discuss two designs, our initial design, its shortcoming and our final design. Section 4 summarizes the provided Prolog libraries. The final sections discuss how s(CASP) is made available online embedded in SWISH, provide an evaluation and conclusions.

2. A closer look at the requirements

We already mentioned Prolog's advantages for assembling s(CASP) programs and post-processing the output of the s(CASP) solver. In this section we take a closer look at the requirements.

Dynamic assembly of a s(CASP) program Dynamically assembling a s(CASP) program is a requirement for applications that wish to add *case data* to the program, i.e., the state of the (relevant) world. This typically concerns *facts*. In the case of incomplete knowledge we may explore the consequences of defining some facts as *abductive*. In addition, dynamic manipulation of *rules* is required to explore the search space for machine learning tasks such as ILP (*Inductive Logic Programming*).

Tools and program analysis Prolog has a much longer history than s(CASP) when it comes to tools for syntax checking, dependency analysis, etc. Our integration should preserve these features as much as possible.

Modularity of s(CASP) An application may involve multiple s(CASP) programs that may use each other. This implies we must be able to represent s(CASP) programs as units and allow a unit to use other units.

Modular access s(CASP) applications may want to have access at different levels to the results. Some programs want to reason about the model and justification and need a clean representation thereof as Prolog terms. Interactive programs aiming at experts need an integration into the Prolog toplevel that represent the model and justification in a concise and easy to read format. Finally, end users typically want a natural language description of the findings and justification.

3. Choosing a suitable embedding

3.1. Our initial solution: blocks

In our invited talk at GDE-2021 [5] we proposed a solution that stayed as close as possible to a s(CASP) program as represented in a file. A minimal example is below.

```
:- use_module(library(scasp/embed)).
:- begin_scasp(qp, [p/1]).
p(X) :- not q(X).
p(1).
q(X) :- not p(X).
:- end_scasp.
```

Using **begin_scasp/2** ... **end_scasp/0** we established the two steps of the embedding process. Firstly the opening directive allows adjusting the Prolog syntax by means of (re)defining operators and other flags. Below are some examples of statements that are legal in the original s(CASP) implementation and illegal or ambiguous in Prolog.[1]

- `#abducible penguin(X).`
- `_p :- q.`
- `:- p,q.`
- `?- q(X).`

While reading the s(CASP) block a **term_expansion/2** rule is active that validates that all input is valid s(CASP) input. The input is collected. The expansion of **end_scasp/0** calls the s(CASP) compiler to generate the program.

3.1.1. Issues using s(CASP) blocks

While the above solution stays as close as possible to using s(CASP) as loaded from a separate file and the compilation of the s(CASP) program is done at Prolog compile time, this approach has several disadvantages.

- The strongly altered Prolog syntax confuses the Prolog support of many editors.
- The syntactical changes make it harder to manipulate s(CASP) programs. We ended up with a predicate that could add a complete block dynamically. To assemble the content of this block we need to enable the syntax modifications.
- We need specialised code to import other s(CASP) units.
- We cannot use any normal Prolog convention to analyse, inspect or modify the program.

[1] `_p :- q.` can be made legal in SWI-Prolog using the Prolog flag `allow_variable_name_as_functor`.

3.2. Our final solution: dynamic reinterpretation of Prolog

The above limitations made us reconsider our initial choice to embed s(CASP) in isolated blocks. In our current approach we allow s(CASP) reasoning over a *(restricted) Prolog program.* The advantages of this approach are obvious: we can use all normal tools and Prolog interfaces to manage s(CASP) programs. Think about, assert/retract, modules, cross-referencing, etc.

This approach has some consequences. Firstly, we need to define replacements for some of the syntactical problems. Second, given a query we need to collect the relevant s(CASP) program, compile the program, solve the query using s(CASP) semantics and dispose of the program. Below we have a closer look at some of the consequences.

Syntax We took the following decisions about the syntax:

- Predicates starting with an underscore (`_p(X)`) are not supported without quotes (i.e., `'_p'(X)`) is allowed.
- s(CASP) *directives* are written as normal Prolog directives, e.g., `:- abducible p(_X).` **abducible/1** is defined as a predicate that may also be called at runtime. It asserts an annotation that is picked up by the s(CASP) compiler.
- *Global constraints* are written as `false :- Body.` to avoid conflicting with Prolog directives.
- Classic negation is still written as `-p :- Body.` The - prefix for a clause head or goal is handled by **term_expansion/2** and **goal_expansion/2** to include the `-` into the name.

Modifying the s(CASP) program Modification of the s(CASP) program is close to the modification of a normal Prolog program, unless classical negation or global constraints are involved. For this reason we added **scasp_assertz/1** and similar predicates for all database manipulation predicates. For example, `scasp_assertz(-p(1))` is equivalent to `assertz('-p'(1))` Similarly some Prolog directives such as **discontiguous/1** have as scasp_ counterpart to act on classical negation.

Collecting the s(CASP) program Fortunately s(CASP) program terms are a simple subset of Prolog. Due to the lack of higher order constructs such as **call/1**, analysis of the call graph is straightforward. The analysis does however require introspection capabilities to all relevant clauses. SWI-Prolog allows for **clause/2** to operate on *static* predicates which allows for inspecting static Prolog programs. Some other implementations (e.g., Quintus) allow loading programs as *all dynamic*, providing the same features. In other systems one may use **term_expansion/2** to save a copy that can be inspected.

Given **clause/2** (or some alternative) the analysis starts with the query, fetches all clauses relevant to the query and then recursively all clauses relevant to the body of the collected clauses. This process continues until no new clauses can be found. While doing so each clause is inspected not to have any forbidden constructs such as *foreign predicates*, control structures except for conjunction or meta-calling. The translation aborts with an error if such a clause is found.

Global constraints are a separate problem. As said, they are rules for **false/0**, which is translated during program loading to rules for **-/0** to avoid a conflict with the ISO builtin **false/0**. s(CASP) semantics define that all global constraints of the program shall be included in the translation and evaluation. This is impractical as we no longer have blocks and thus do not know which global constraints belong to which s(CASP) program. Therefore we extend the *goal directed* properties of s(CASP). We include a global constraint and its call tree if (and only if) the call tree of the body of the global constraint contains a literal that overlaps with the call tree of the current s(CASP) program. We compare two literals for overlapping after normalization by removing **not/1** and classical negation. This process is applied until there remain no further global constraints whose call tree overlaps with the current program.

Next, we collect all annotations created by s(CASP) directives.

As the original s(CASP) compiler does not support modules we flatten the module qualified program by including the module name into the name of the predicates, e.g. `'rules:p'(X) :- 'facts:f'(X), 'facts:g'(X).` As a result, s(CASP) programs are not allowed to have the `:` character in their name.

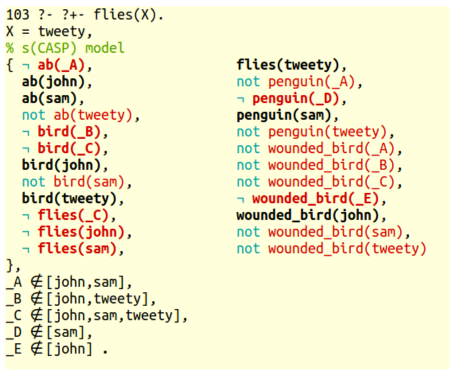
The result is passed to the barely modified original s(CASP) compiler to create the *dual*, *OLON* and *nmr* rules. These are stored in a SWI-Prolog *temporary module*. The module is created and destroyed using **setup_call_cleanup/3**, where the *setup* handler creates the module and performs the s(CASP) compilation as described above, the *call* handler runs **scasp/2** and the *cleanup* handler destroys the module and its content.

Executing a s(CASP) query The principal predicate for evaluating a s(CASP) query is **scasp/2**, which takes a query and an option list. The option list represents many the s(CASP) commandline options (such as the *forall* algorithm, enabling warnings, disabling OLON rules, etc. In addition it has an option model(Model) and tree(Tree) that may be used to get access to the complete model and justification tree.

The *Model* is a list of Prolog terms, optionally embedded in **not/1** or -/1, e.g. `not(-(p))` means the classical negation of **p/0** could not be proven. The model is ordered by the literal that is involved, i.e. all model terms that belong to a specific literal are adjacent. Variables in the model may have constraints that are accessible using the Prolog built-in **copy_term/3**.

The justification *Tree* is a Prolog term of the shape *Node - Children*. Here, *Node* has the same representation as a model term and *Children* is a list of nodes, possibly embedded in **chs/1** (assumed in *even* loop) or **proved/1** (already proved earlier in the justification).

For *toplevel* (REPL) interaction we built upon ?/1 that was provided by the original implementation. We introduced nine predicates with the names **?/1**, **??/1**, **?–/1**, **?+-/1**, **?-+/1**, **?++/1**, **??+-/1**, **??-+/1** and **??++/1**. Predicates with a single ? display their results in formal notation using Unicode characters for logical connectors. Predicates with a double ? (??) display their results as natural language. The first $\pm$ specifies whether the model is displayed and the second $\pm$ wheter the justification is displayed. **?/1** is an alias for **?+-/1** and **??/1** is an alias for **??++/1** for compatibility with the original s(CASP) implementation. If supported by the terminal, both use colour output using green for positive literals, orange for NAF and red for classical negation. See figure 1



```
103 ?- ?+- flies(X).
X = tweety,
% s(CASP) model
{ ¬ ab(_A),                     flies(tweety),
  ab(john),                     not penguin(_A),
  ab(sam),                      ¬ penguin(_D),
  not ab(tweety),               penguin(sam),
  ¬ bird(_B),                   not penguin(tweety),
  ¬ bird(_C),                   not wounded_bird(_A),
  bird(john),                   not wounded_bird(_B),
  not bird(sam),                not wounded_bird(_C),
  bird(tweety),                 ¬ wounded_bird(_E),
  ¬ flies(_C),                  wounded_bird(john),
  ¬ flies(john),                not wounded_bird(sam),
  ¬ flies(sam),                 not wounded_bird(tweety)
},
_A ∉[john,sam],
_B ∉[john,tweety],
_C ∉[john,sam,tweety],
_D ∉[sam],
_E ∉[john] .
```

Figure 1: REPL output of query showing tabular output, Unicode logical connectors, colours to indicate connectors and the truth of literals. Remaining constraints are displayed below the model.

4. s(CASP) as a modular Prolog library

The SWI-Prolog s(CASP) system has several libraries for public usage.

library(scasp) This library is the core for embedded usage. It exports the s(CASP) operators, expansion rules, s(CASP) directives, predicates to inspect and modify the s(CASP) program and predicates to run s(CASP) queries.

library(scasp/html) This library exports predicates to translate a justification tree to an HTML string. The output contains both the formal and natural language output. The system provides JavaScript and CSS to switch between the formal and natural language output and to fold and unfold the justification tree. The library provides partial support for multiple languages. The translation is based on the work on the original s(CASP) implementation [6].

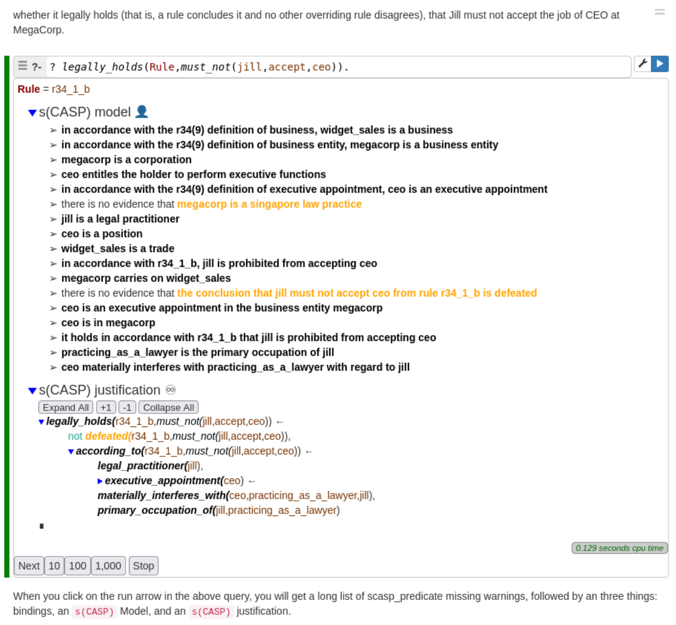
library(scasp/human) This library exports predicates to translate a justification tree into a natural language string. This library uses the rules from library(scasp/html) such that we can use one set of translation rules for both HTML and plain text output.

library(scasp/main) This library provides the code for creating the stand-alone executable.

5. s(CASP) online using SWISH

SWISH [7] is an online version of SWI-Prolog where multiple users can connect to a SWI-Prolog server process to edit, store and execute Prolog programs. The programs are executed on the server. When a user executes a query on a program in the browser, the program is transferred to the server, loaded into a temporary module (see page 5), and the query is executed. The public SWISH instance is popular for education. It currently (May 2022) stores 151,164 programs and is on average used by a couple of hundreds of (mostly) students.[2]

[2]Usage follows a daily, weekly and (academic) year cycle, ranging between about 50 and 1,200 concurrent users).



whether it legally holds (that is, a rule concludes it and no other overriding rule disagrees), that Jill must not accept the job of CEO at MegaCorp.

```
?- ? legally_holds(Rule,must_not(jill,accept,ceo)).
```

```
Rule = r34_1_b
▼s(CASP) model
    ➢ in accordance with the r34(9) definition of business, widget_sales is a business
    ➢ in accordance with the r34(9) definition of business entity, megacorp is a business entity
    ➢ megacorp is a corporation
    ➢ ceo entitles the holder to perform executive functions
    ➢ in accordance with the r34(9) definition of executive appointment, ceo is an executive appointment
    ➢ there is no evidence that megacorp is a singapore law practice
    ➢ jill is a legal practitioner
    ➢ ceo is a position
    ➢ widget_sales is a trade
    ➢ in accordance with r34_1_b, jill is prohibited from accepting ceo
    ➢ megacorp carries on widget_sales
    ➢ there is no evidence that the conclusion that jill must not accept ceo from rule r34_1_b is defeated
    ➢ ceo is an executive appointment in the business entity megacorp
    ➢ ceo is in megacorp
    ➢ it holds in accordance with r34_1_b that jill is prohibited from accepting ceo
    ➢ practicing_as_a_lawyer is the primary occupation of jill
    ➢ ceo materially interferes with practicing_as_a_lawyer with regard to jill
▼s(CASP) justification
  ▼legally_holds(r34_1_b,must_not(jill,accept,ceo)) ←
      not defeated(r34_1_b,must_not(jill,accept,ceo)),
      ▼according_to(r34_1_b,must_not(jill,accept,ceo)) ←
          legal_practitioner(jill),
          ▸executive_appointment(ceo) ←
          materially_interferes_with(ceo,practicing_as_a_lawyer,jill),
          primary_occupation_of(jill,practicing_as_a_lawyer)
  ∎
                                                            0.129 seconds cpu time
```

When you click on the run arrow in the above query, you will get a long list of scasp_predicate missing warnings, followed by an three things: bindings, an `s(CASP)` Model, and an `s(CASP)` justification.

Figure 2: s(CASP) embedded in a SWISH *notebook*. The output field contains the binding, the model in natural language notation and the justification in formal language. This example is created by Jason Morris and available in the public SWISH instance [8]

Given the embedding in Prolog, we can load library(scasp) in SWISH and run s(CASP) queries online. A brief tutorial is provided. Figure 2 illustrates the interface.

6. Evaluation

We used the above integration to realise the scasp embedding into SWISH. We use s(CASP) at Forsante Oy where it will operate as a micro service using the STOMP protocol for communication. Finally we implemented a simple HTTP service for s(CASP) that is available as `examples/dyncall/http.pl` from the SWI-Prolog s(CASP) distribution. Currently (May 2022) SWISH stores 64 programs that match on a search for `library(scasp)`.[3] The SWI-Prolog forum collected 41 topics that mentions s(CASP), 22 of which also mention SWISH.

As Forsante Oy we used dynamic manipulation of the s(CASP) program to reduce the number of variables introduced in the body of clauses. Such variables introduce *forall* reasoning that proved to be a major performance bottleneck. For example, we have facts of the format `measurement(What, Unit, Value)` and rules that depend whether some measurement has been performed. Such rules may be written down as below. Instead, we add facts **measured/1** derived from all **measurement/3** facts.

```
measured(What) :- measurement(What,_,_).
```

[3]Note that it is not needed to store a program on the server. The SWISH web page stores your programs in the browser's *local storage*.

Although we have not performed a systematic analysis of the cost for dynamically collecting and compiling the s(CASP) for each query this part of the evaluation never came above a few percent of the total s(CASP) evaluation time according to SWI-Prolog's profiling tools.

7. Conclusions and future work

Embedding s(CASP) in Prolog by evaluating a normal Prolog program using s(CASP) semantics by dynamically collecting, compiling and evaluating the s(CASP) query proved to be a powerful technique. This way of embedding is supported by the established Prolog development toolchain. All normal Prolog features for loading, modifying, introspection and modularity are supported. To deal naturally with classical negation and global constraints we need some preprocessing while loading such programs as well as wrappers around the Prolog database predicates, e.g., **scasp_assert/1/.**

We consider the embedding successful. Future versions may incorporate minor changes to the interface. Further work on s(CASP) will concentrate on better natural language output with support for multiple languages. Other areas of interest are performance and testing.

References

[1] V. Lifschitz, Answer set planning, in: D. D. Schreye (Ed.), Logic Programming: The 1999 International Conference, Las Cruces, New Mexico, USA, November 29 - December 4, 1999, MIT Press, 1999, pp. 23–37.

[2] K. Marple, G. Gupta, Dynamic consistency checking in goal-directed answer set programming, Theory Pract. Log. Program. 14 (2014) 415–427. URL: https://doi.org/10.1017/S1471068414000118.

[3] J. Arias, M. Carro, E. Salazar, K. Marple, G. Gupta, Constraint answer set programming without grounding, Theory Pract. Log. Program. 18 (2018) 337–354. URL: https://doi.org/10.1017/S1471068418000285. doi:10.1017/S1471068418000285.

[4] J. Wielemaker, T. Schrijvers, M. Triska, T. Lager, Swi-prolog, Theory Pract. Log. Program. 12 (2012) 67–96. URL: https://doi.org/10.1017/S1471068411000494.

[5] J. Wielemaker, J. Arias, G. Gupta, s(CASP) for SWI-Prolog, in: J. A. et al. (Ed.), Proceedings of the ICLP 2021 Workshops, Porto, Portugal (virtual), September 20th-21st, 2021, volume 2970 of *CEUR Workshop Proceedings*, CEUR-WS.org, 2021. URL: http://ceur-ws.org/Vol-2970/gdeinvited4.pdf.

[6] J. Arias, M. Carro, Z. Chen, G. Gupta, Justifications for goal-directed constraint answer set programming, in: F. R. et al. (Ed.), Proceedings 36th ICLP Technical Communications 2020, UNICAL, Rende (CS), Italy, 18-24th September 2020, volume 325 of *EPTCS*, 2020, pp. 59–72. URL: https://doi.org/10.4204/EPTCS.325.12. doi:10.4204/EPTCS.325.12.

[7] J. Wielemaker, T. Lager, F. Riguzzi, SWISH: swi-prolog for sharing, CoRR abs/1511.00915 (2015). URL: http://arxiv.org/abs/1511.00915. arXiv:1511.00915.

[8] J. Morris, Constraint Answer Set Programming as a Tool to Improve Legislative Drafting: A Rules as Code Experiment, Association for Computing Machinery, New York, NY, USA, 2021, p. 262–263. URL: https://doi-org.vu-nl.idm.oclc.org/10.1145/3462757.3466084.