s(CASP) for SWI-Prolog

Jan Wielemaker[1,2], Joaquín Arias[3] and Gopal Gupta[4]

[1]SWI-Prolog Solutions b.v., Amsterdam, The Netherlands
[2]Vrije Universiteit Amsterdam, De Boelelaan 1105, 1081 HV Amsterdam, The Netherlands
[3]Universidad Rey Juan Carlos, Calle Tulipán s/n, 28933 Móstoles (Madrid), Spain
[4]The University of Texas at Dallas, Dallas, USA

Abstract

s(CASP) is related to ASP. Unlike ASP, which is traditionally solved using *grounding* and a SAT solver, s(CASP) is solved using top-down goal directed search without grounding. This allows s(CASP) to solve problems that cannot be grounded, while the generated proof tree is a good basis for giving a justification for the answer. s(CASP) supports both negation as failure (NAF) and classical negation. These features make s(CASP) particularly suitable for commonsense reasoning tasks that require a justification of the answer. Currently, `scasp` is an executable generated using Ciao.

We ported s(CASP) to SWI-Prolog. The primary aim of this is to provide s(CASP) as a *library* that allows for embedding and managing multiple s(CASP) programs from multiple threads. Here, 'Managing' means being able to construct and modify an s(CASP) program dynamically and dynamically run queries against s(CASP) programs.

Keywords

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

1. Introduction

Answer Set Programming (ASP) [1] is nowadays an established paradigm in the logic programming community. Traditionally, ASP notably targets difficult search problems. ASP solvers normally *ground* the input program and then use a SAT solver to find stable models.[1] Disadvantages of this technique is that not all programs have a finite grounding and that it is hard to generate a human understandable *justification* for an answer [2]. In contrast, s(CASP) [3] is *goal directed* (like Prolog) and does not ground the program. As a result it can integrate Constraint Logic Programming (CLP) that allows, for example, to express $X > 3$ without any further knowledge about X. In addition, the solver produces a *stack* that represents how each atom or its negation has been derived. This stack is input to the justification generation. s(CASP) is implemented as a meta interpreter in Prolog.

While s(CASP) is, in its current implementation, not particularly suitable for hard problems, it is suitable for automation of reasoning tasks normally carried out by humans such as reasoning

GDE'21: ICLP'21 (Virtual) Workshop on Goal-directed Execution of Answer Set Programs, September 20, 2021
jan@swi-prolog.com (J. Wielemaker); joaquin.arias@urjc.es (J. Arias); gupta@utdallas.edu (G. Gupta)
0000-0001-5574-5673 (J. Wielemaker); 0000-0003-4148-311X (J. Arias); 0000-0001-9727-0362 (G. Gupta)

[1]Being completely declarative, ASP solvers are not restricted in the techniques they use to find an answer set. Modern solvers are high optimised.

in legal or medical domains. s(CASP) is claimed to be suitable for Commonsense Reasoning.[2]

As the family of ASP languages is purely declarative without any side effects the solvers require some other language that prepares the ASP program, runs the solver and takes action depending on the output of the solver. Prolog is an ideal language for embedding ASP systems because manipulating clauses, handling bindings to logical variables, reasoning about the model and interpreting the justification all fit perfectly with Prolog. In our case, the s(CASP) implementation is already in Prolog and produces possible bindings, models and justifications on backtracking.

This article explains our port of s(CASP) to SWI-Prolog and discusses several alternatives to making s(CASP) accessible from Prolog. We report on unfinished work. This article is primarily input for discussion.

2. Porting s(CASP) to SWI-Prolog

The GIT repository https://gitlab.software.imdea.org/ciao-lang/sCASP provides the source for s(CASP) for Ciao. The source reuses the compiler of s(ASP) by Kyle Marple [4]. Operation between variables in the s(ASP) implementation are handled 'manually'. The s(ASP) execution maintains a mapping between variables names and their current bindings (values as well as disequality constraints). The s(CASP) execution lets Prolog take care of all operations that it can handle natively, instead of interpreting them. Therefore, a large part of the environment for the s(CASP) program is carried implicitly in the Prolog environment. Since s(CASP) and Prolog shared many characteristics (e.g., the behavior of variables), this results in flexibility of implementation and gives a large performance improvement. [3]

We ported s(CASP) to SWI-Prolog. The aim of this project is twofold: (1) create a uniform codebase, avoiding duplicate work and dead code that resulted from building s(CASP) (using Ciao style) on top of s(ASP) (using SWI-Prolog style) and (2) provide s(CASP) as an embedded language in Prolog in addition to the stand-alone executable. With (1) we aim at a modular and industrial strength implementation of s(CASP). With (2) we recognise that an ASP system can only function as a reasoning component in an application written in some other language that takes care of assembling the ASP model, running queries and making use of the bindings, model and justification. We argued that Prolog is an ideal language for this purpose in the introduction.

We realised two changes in the design of s(CASP) to better facilitate embedding in Prolog. First of all, we replaced the DCG (Prolog grammar rules) based parser by the Prolog parser. This is possible by defining an adequate set of Prolog operators.[3] The terms read are validated to satisfy the restrictions imposed by s(CASP) and converted into the same format that was emitted by the original parser. Second, where the original implementation generated the justification directly from the resulting s(CASP) solver's stack, the current implementation creates a Prolog representation from the stack that naturally represents the justification while hiding the internals of the solver.

[2] See https://personal.utdallas.edu/~gupta/csg/

[3] s(ASP) allows for predicate names that start with an underscore, as in `_p(X)`. This can be handled in SWI-Prolog, but not in standard ISO Prolog.

```
?- p(X).
sCASP model: [p(X),not q(X)],
sCASP justification
   query ←
      p(X) ←
         not q(X) ←
            chs(p(X)) ∧
      o_nmr_check ;
X = 1,
sCASP model: [p(1)],
sCASP justification
   query ←
      p(1) ∧
      o_nmr_check ;
false.
```

Figure 1: Running an s(CASP) query as a normal Prolog query

3. Accessing s(CASP) from Prolog

Currently, s(CASP) programs can be embedded into Prolog using two directives that create a *block*. The opening directive changes the Prolog operator tables and activates rules for **term_expansion/2** that validate the s(CASP) terms and does some simple transformations that avoid conflicts with Prolog. The closing directive reverts the changes to the operator table, compiles the loaded terms into the internal s(CASP) representation and creates *wrappers* that make the exported s(CASP) predicates accessible as normal Prolog predicates from the enclosing Prolog module.

```
:- begin_scasp(Unit, Exports).
<sCASP program>
:- end_scasp.
```

We give a minimal example 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.
```

After loading this Prolog program, **p/1** is made available as a normal Prolog predicate as shown in figure 1. Calling this predicate makes the bindings available as normal Prolog bindings while alternative solutions are made available through Prolog backtracking. The model and

justification are stored in *backtrackable global variables* as provided by **b_setval/2** in SWI-Prolog. By using *backtrackable global variables* we can provide access to these extensions to the normal Prolog answer (only the bindings) while full sharing of variables and constraints are retained.

The model is represented as a list of Prolog terms, where each term may be nested as `not(Term)` (not provable), `-(Term)` (false) or `not(-(Term))` (not provable that `Term` is false). The justification is a tree represented as a term `-(Term, ListOfChildren)`. Here `ListOfChildren` is the empty list for facts and a list of -/2 terms otherwise. In addition to the negation symbols, terms may be nested in one of the terms below.

- `proved(Term)` Term was already proved.
- `chs(Term)` Term is assumed because it appears in a loop with an even number of negations
- `assume(Term)` Marks the loop detected by `chs(Term)`

The model is made available by **scasp_model/1** and the justification tree using **scasp_justification/2**, where the second argument is an option list that may be used to adjust the level of detail in the tree.

The SWI-Prolog toplevel displays answers to a query as a valid Prolog body term. For traditional Prolog this is a conjunction of unification (=/2) calls that establish the binding or `true` if no variables are bound. If some of the variables are subject to constraints the constraints are reified using **copy_term/3** and added as goals to the answer, for example, using clp(fd) we may get:

```
?- A #> 2, A #< 5.
A in 3..4.
```

To accommodate CHR, which has a notion of a global constraint store, SWI-Prolog provides a hook that can be used to add additional goals to the answer. This hook is used to make the model and justification visible in the toplevel. This is not entirely satisfactory because the printed output is no longer valid Prolog input that has the same effect as the original query.

4. Some notes on portability

The current codebase is SWI-Prolog specific. While many Prolog implementations have some way to embed code by temporarily changing the operator table and doing macro expansion, there is no standard. The same applies for integrating the model and justification into the Prolog REPL loop.

Ideally the data representations such as the program representation going into the s(CASP) compilation phase, the s(CASP) intermediate program representation, the model and justification are synchronised between (Prolog) s(CASP) implementations. That allows for sharing the program transformation and solver code as well as code using the output of the embedded s(CASP) system.

5. Discussion and future work

When porting s(CASP) to SWI-Prolog we set ourselves the following goals: (1) cleanup of the code base to improve maintainability and modularity and (2) allow using s(CASP) as embedded language and library from Prolog. This is work in progress. The current implementation can handle one or more s(CASP) programs embedded in one or more Prolog modules. The implementation is *thread-safe*.

The current implementation has no interface for composing s(CASP) programs dynamically. This is a vital feature for Inductive Logic Programming (ILP) as well as for assembling a suitable program based on currently known case data.

Another approach to execute a goal using s(CASP) semantics would be to automatically re-interpret a Prolog program under s(CASP) semantics. One way to do that would be to declare predicates as s(CASP) predicates using e.g., `:- scasp p/1, q/2.`. Alternatively we could define a predicate `scasp(Goal)`. Both approaches would need to establish the reachable program, verify it satisfies the limitations imposed by s(CASP), compile the program to the s(CASP) intermediate representation and run the s(CASP) solver. This approach provides a trivial API for manipulating the s(CASP) program using **assert/1** and **retract/1**. It is not clear how s(CASP) *global constraints* fit into this picture.

We also plan to support embedding s(CASP) into SWISH, providing an online environment for teaching and exchanging ideas.

References

[1] V. W. Marek, M. Truszczynski, Stable models and an alternative logic programming paradigm, in: K. R. Apt, V. W. Marek, M. Truszczynski, D. S. Warren (Eds.), The Logic Programming Paradigm - A 25-Year Perspective, Artificial Intelligence, Springer, 1999, pp. 375–398. URL: https://doi.org/10.1007/978-3-642-60085-2_17. doi:10.1007/978-3-642-60085-2_17.

[2] J. Arias, M. Carro, Z. Chen, G. Gupta, Justifications for goal-directed constraint answer set programming, in: F. Ricca, A. Russo, S. Greco, N. Leone, A. Artikis, G. Friedrich, P. Fodor, A. Kimmig, F. A. Lisi, M. Maratea, A. Mileo, F. Riguzzi (Eds.), Proceedings 36th International Conference on Logic Programming (Technical Communications), ICLP Technical Communications 2020, (Technical Communications) 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.

[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] G. Gupta, E. Salazar, K. Marple, Z. Chen, F. Shakerin, A case for query-driven predicate answer set programming, in: G. Reger, D. Traytel (Eds.), ARCADE 2017, 1st International Workshop on Automated Reasoning: Challenges, Applications, Directions, Exemplary Achievements, Gothenburg, Sweden, 6th August 2017, volume 51 of *EPiC Series in Computing*, EasyChair, 2017, pp. 64–68. URL: https://doi.org/10.29007/ngm2. doi:10.29007/ngm2.