s(ASP) & s(CASP) Best Practices

April 29, 2021

Introduction

The approach to programming in s(ASP) is somewhere between the strategies used for SAT-based asp and the strategies for prolog. In addition, s(ASP) is a young technology with many rough edges. This document discusses the parts of s(ASP) that is different from SAT-based asp and prolog or still in need of work, requiring workarounds.

Variable Domains

Unspecified Domains

SAT-based asp solvers depend on the program being finitely groundable, and require safe programs. A safe program requires that a variable used in a negated call must also appear in a positive goal.

```
node(0).
node(1).
node(2).

hidden(0).
visible(N) :- node(N), not hidden(N).
```

Since s(ASP) does not ground the program, and variables can remain unbound, we do not need the program to be safe.

```
visible(N) :- not hidden(N).
```

Now, `visible/2` will be true for all values of `N` except zero.

Finite Domains

Like with prolog, s(ASP) does not have a built-in concept of a domain. We can implement a domain by creating a predicate that is true if the argument is in that domain.

```
veg(tomato).
veg(lettuce).
veg(onion).

meat(beef).
meat(chicken).
meat(pork).

ingredient(X) :- veg(X).
ingredient(X) :- meat(X).
```

In the above example, we create three domains. The domain of vegetables, the domain of meats, and the domain of ingredients. We can use these predicates to restrict a variable to a specific domain.

```
dish(hamburger).
dish(salad).

in(hamburger, beef).
in(hamburger, lettuce).
in(hamburger, tomato).
in(hamburger, onion).
in(salad, lettuce).
in(salad, tomato).
in(salad, onion).

-vegetarian(D) :- dish(D), meat(I), in(D,I).
```

Here we restrict `D` to the dish domain. So, we can read it as "a dish is not vegetarian if there exists some meat in the dish." This works well. However, since the concept of a domain does not exist in the language these "domain" predicates are just ordinary predicates. This does not present a problem in Prolog, but in s(ASP) this can lead to a slight disconnect between the logic of the program, and how we view the program. As an example, suppose we wanted to restrict a variable to vegetarian dishes. We may do a query like this: `?- not -vegetarian(D)`. However, the first result we receive is `D` is unbound and constrained to not be `hamburger` and `salad`. That is, a dish we know nothing about is vegetarian. Logically, this makes sense. Since in absence of any additional information we follow an open world assumption: any dish that we do not specify will not have any ingredients, and therefore no meat. From our point of view, however, we may be viewing the dish domain to be a finite domain containing only what we have specified. We may expect that `D` be restricted to the dish domain.
The solution for this is to never directly call the NAF-negation of a predicate containing a "finite domain" variable in the head, and restrict the variable before calling it.

```
vegetarian(D) :- dish(D), not -vegetarian(D).
```

Now, `D` will be ground (restricted to the dish domain) before the call to `not -vegetarian(D)`.

Constraints

The general strategy for a SAT-based asp solver is "generate and test". Some rules generate possible worlds, and constraints in the form of odd loops or, more commonly, headless rules filter out (kill) unwanted worlds. The general strategy for s(ASP) is to make use of "local" constraints. Local constraints involve killing the world as it is generated. Since s(ASP) is a goal-directed language, local constraints may decrease the amount of backtracking needed. To fully understand how to use local constraints you must first understand how s(ASP) handles global constraints.

Global Constraints

Global constraints can be specified through an odd loop or a headless rule. An odd loop is a cycle formed by rules for with the success of some goal depends on its own negation. This of course is a contradiction. In stable models if there is a contradiction, then there is no model. However, if one of the rules involved in the cycle evaluates to false, then the cycle is broken.

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

The above program has no model. The truth value of p depends on its negation (the same is true for r). However, if we remove the fact for q, the resulting program now has a stable model. With this we can say that the odd loop enforces a constraint. In this case the constraint is that q must be false.

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

This code has an odd loop that enforces the constraint "q or s must be false". Headless rules provide a shorthand for writing such constraints. For instance the same constraint can be specified as:

```
:- q,s.
```

As a simple example of using global constraints, consider the graph coloring problem.

```
color(N,C) :- node(N), color(C), not other_color(N,C).
other_color(N,C) :- color(C), color(C2), C\=C2, color(N,C2).

:- edge(N1,N2), color(N1,C), color(N2,C).
```

The `color/2` and `other_color/2` form an even cycle, and can be viewed as forming all possible coloring of the graph. The headless rule states that neighbors cannot have the same color, so kill all such colorings. For each odd loop and headless rule s(ASP) generates a nmr check. This check negates the body of each rule and adds a new rule for just the head. This can be interpreted as the body of the rule is false or the head is true through some other means. Then the nmr check is appended to the query. To complicate matters further, since s(ASP) is executed ungrounded, the nmr check must be wrapped in a forall for each variable in the head. Consider this simple program:

```
p(X) :- q(Y), not p(Y).
```

The nmr check for this would look like:

```
chk_11(X,Y) :- not q(Y).
chk_11(X,Y) :- q(Y), p(Y).
chk_11(X,Y) :- p(X).

chk_1(X) :- forall(Y, chk_11(X,Y)).

nmr :- forall(X,chk_1(X)).
```

Headless rules do not need the outer forall, but will have a forall for each variable that appears. The nmr check generated by the graph coloring program is:

```
chk_11(N1,N2,C) :- not edge(N1,N2).
chk_11(N1,N2,C) :- edge(N1,N2), not color(N1,C).
chk_11(N1,N2,C) :- edge(N1,N2), color(N1,C), not color(N2,C).

chk_1 :- forall(N1, forall(N2, forall(C, chk_11(N1,N2,C)))).

nmr :- chk_1.
```

Local Constraints

Consider the following graph:

```
edge(0,1).

edge(2,3).
edge(3,4).
edge(4,5).
edge(5,6).
.
.
.
```

If we want to know the color of node zero we can write a simple query:

```
?- color(0,C).
```

The color of node 0 depends only on the color of node 1, but the global constraint must be enforced for all nodes. This, of course, is the desired behavior if the entire graph cannot be colored. But, if there is a coloring, we do not care what that coloring actually is. So we are doing more work than necessary. In fact,

is some situations we may not even care if there is a solution for the entire graph as long as the part we are working with is consistent. In this situation it would be better to use a local constraint.
The same constraint needs to be enforced, but instead of killing a model after generation we will kill such models during generation. A general strategy of converting a global constraint to a local constraint is to use the global constraint as the body of a new rule. The predicate in the head of the rule can also be used to give the constraint a name. Let's take a look at what the local constraint for the graph coloring might look like.

```
same_as_neighbors(N,C) :- edge(N,N2), color(N2,C).
```

This constraint can be read as: "for all nodes N and color C, if there exists a neighbor of N assigned color C, then assigning N the color C will make it the same as one of its neighbors." We have localized the global constraint by parametrizing it. We have specified when our global constraint will be violated. Now we need to enforce the constraint. We can do this by appending `not same_as_neighbors(N,C)` to the end of the color rule:

```
color(N,C) :- node(N), color(C), not other_color(N,C), not same_as_neighbors(N,C).
other_color(N,C) :- color(C), color(C2), C\=C2, color(N,C2).
```

Now when we query `color(0,N)`, we only need to consider node 0's neighbors. This is not entirely equivalent to the global constraint that enforces the constraint for all nodes. As implied earlier, if the rest of the graph does not have a coloring, the query will still succeed. We can create the equivalent of the global constraint by creating a new rule and adding its head to our query or to a rule we will call.

```
global_constraint_violated :- same_as_neighbors(N,C).
```

This will do essentially the same thing as a headless rule, but we can move the constraint to any point in the computation. In fact, we may even use the local constraint rule with a headless rule to recreate the global constraint. In this way we get all the features of a global constraint, but the expressiveness of local constraints.

```
:- same_as_neighbors(N,C).
```

Workarounds

The s(ASP) system is a young technology, and as such has many areas that need improvement. This section covers some area of concerns that can affect the execution of programs.

Large Sequence of Facts

Often times we want to create a domain by specifying a large number of ground facts. These ground values are the members of the domain. The s(CASP) is a meta-interpreter in ciao prolog, and therefore indexes on the first argument. So, just like in prolog these values can be found quickly. However, when calling the dual rule, we must constrain a variable against all values. If we call the negation with a ground value, we must compare it to all the values.

```
id(0).
id(1).
id(2).

?- not id(3).
```

The dual for `code/1` is equivalent to `3\=0, 3\=1, 3\=2`. This is linear in the number of facts. So, we want to avoid such negation. It is, however, not so straight forward. We rarely call the negation of a domain predicate, but we may call the negation of a predicate that depends on the set of facts.

```
logged_in(Id) :- id(Id), property(logged_in, Id).
```

The query `?- logged_in(7).` can be proved efficiently using indexing. However, the execution of `?- not logged_in(7).` will be linear in the number of id's in the system. We can make use of the same trick introduced in subsection "Finite Domains". We do not directly call the negation of a predicate that depends on a large set of facts.

```
-logged_in(Id) :- id(id), not logged_in(Id).
```

Avoiding Forall

In prolog (and in s(ASP)), a variable in the body of a rule that does not appear in the head can be considered to be existentially quantified. This takes on a slightly different interpretation due to the goal directed nature of prolog and s(ASP).

```
p(X) :- q(X,Y).
```

We can interpret this rule as "for all X, if we can prove there exists a Y such that q(X,Y) is true, then p(X) is true". The distinction from mathematical logic comes from the word "prove". We do not have to do anything extra, however. If q(X,Y) succeeds, then we have proved the existence of a value for Y. However, s(ASP) has negation implemented as dual rules. This complicates things. Since the negation of an existential quantifier is a universal quantifier, and since we must prove the existence of a value for such an existentially quantified variable (called body variable in s(ASP) terminology), it's dual must prove that all possible values fail. We can interpret the dual for this example as "for all X, if we can prove that for every Y not q(X,Y) is true, then not p(X) is true".
This introduces another point of inefficiency. The forall algorithm is as follows. Given an unbound, unconstrained variable, say `X`, and a goal containing that variable, say `g(X)`:

1. Call `g(X)`, failing when `X` is bound.
2. If the call succeeds leaving `X` unbound but prohibiting it from being bound to certain values: for each element, e, of `X`'s prohibited list, call `g(X)` with `X` bound to e
3. If the original goal and every call in step 2 succeeds, then the forall succeeds. Otherwise, it fails.

Now consider the following program:

```
person(adam).
person(bill).
:
person(zachary).

owned(Object) :- person(Person), has(Person, Object).
```

In this code we specify a domain via a long list of facts, and the final rule states that "an Object is owned if there exists a person that has it". The dual of owned would look something like:

```
not_owned(Object) :- forall(Person, n_owned1(Object, Person)).
n_owned1(Object, Person) :- not person(Person).
n_owned1(Object, Person) :- person(Person), not has(Person, Object).
```

So if we want to check if an object is not owned, we must execute this forall. Following the algorithm we get:

- `n_owned1(Object, Person)` succeeds with `Person` constrained against `adam, bill, ...,zachary`
- for each `Person = adam`, `Person = bill`, ..., `Person = zachary`: `not n_owned1(Object, Person)` succeeds. (`has/2` has no rules).

There are no nice workarounds for this problem. Currently, the best way to mitigate the cost is to ensure that a minimum number of foralls are used in your code. That means encoding the knowledge with the thought of avoiding foralls. For this example, if I do not need the set of facts, except for situations similar to the `owned/1` predicate, we can use lists:

```
persons([adam, bill, ...  , zachary]).

owned(Object, Persons) :- persons(P), filter_has(P, Object, Persons).

owned(Object) :- owned(Object, P), P\=[].
-owned(Object) :-owned(Object, []).
```

Disunification of Nonground Terms

Currently, s(ASP) expects that at least on of the arguments used in disunification be ground. In the original s(ASP) code, disunification of two nonground terms causes a fatal error. This is to comply with the semantics. The s(CASP) system instead fails, allowing the system to backtrack. This does not comply strictly to the semantics, but allows the computation to continue and provide a more meaningful answer.
The only workaround is to keep it in mind as we code. Since disunifications can be generated automatic, it is difficult (perhaps impossible) to eliminate the problem.

```
same(X,X).
```

The dual will be equivalent to:

```
not same(X, Y) :- X\=Y.
```

To make matters more complicated, `same/2)` could be negated in a nmr check even if we never explicitly use the negation.

Encoding Knowledge

Find All

In Prolog, the findall predicate can be used to find all successful bindings for a goal. This, however, is not so straight forward for s(ASP). Since s(ASP) allows for cycles, this complicates things. At the moment there are three ways we can implement such behavior. One way is to simply pre-supply the list.
If we have a predicate, `people/1`, we can make its argument a list of constants instead of having a fact per constant.

```
people([fred, janet, bill, gary, sarah]).
```

Now we can loop through the list of people as with any data.

```
students([],[]).
students([H|T], [H|T2]) :- student(H), students(T,T2).
students([H|T], T2) :- not student(H), students(T,T2).
```

Another way is to implement the behavior of findall directly.

```
findall_students(Acc, Out) :- people(H), student(H),
                              not member(Acc, H), findall_students([H|Acc], Out).
findall_students(Acc, Out) :- people(H), student(H), member(Acc, H),
                              findall_students(Acc, Out).
findall_students(Acc, Out) :- people(H), not student(H),
                              findall_students(Acc, Out).
findall_students(Acc,Acc).
```

This will work, but introduces a caveat. Consider the following predicate that intends to check all students are passing:

```
all_passing :- findall_students([], Students), all_passing(Students).
all_passing([]).
all_passing([H|T]) :- passing(H), all_passing(T).
```

Now assume the first student is passing, but one of the later ones is not passing. The goal `?- all_passing.` will succeed. This is because, after collecting all the students, `all_passing(Students)` fails. This causes execution to backtrack back into `findall_students` to before the failing student is added to the list, proceeding to the rule that always succeeds. Now we produced a list of all students except the failing one. The final way to get this behavior is to use the prolog findall predicate exposed by s(CASP). This is an "unsupported" feature, and if the code to prove the goal is not essentially prolog, the behavior is undefined.