THE S(CASP) GOAL-DIRECTED ANSWER SET PROGRAMMING SYSTEM

Tutorial and User Manual

ELMER SALAZAR, JOAQUIN ARIAS, SOPAM DASGUPTA,
YANKAI ZENG, PARTH PADALKAR, AND GOPAL GUPTA

Contents

Chapter 1

Introduction

s(CASP) is a system for executing predicate answer set programs in a goal-directed (query-driven) manner. The s(CASP) system has its origins in the discovery of coinductive logic programming (Simon et al. 2006; Gupta et al. 2007). The first system to be developed was the Galliwasp propositional ASP system (Marple and Gupta 2012; Marple et al. 2012). Subsequently, Galliwasp was generalized to predicates, yielding the s(ASP) system (Marple, Salazar, and Gupta 2017b; Marple et al. 2017a). The s(ASP) system implemented many novel concepts such as full constructive negation, partial answer sets containing predicates, body variables, and generalized dual rules. The s(ASP) system was re-implemented and extended with constraints over real numbers (Holzbaur 1995) that culminated in the s(CASP) system (Arias et al. 2018). This tutorial/manual is for the s(CASP) system. We assume that the reader is familiar with logic programming and answer set programming (ASP). Many expository accounts can be found for ASP, we recommend the textbook by Gelfond and Kahl (Gelfond and Kahl 2014). We strongly recommend the user to read the companion article on automating commonsense reasoning (Gupta et al. 2022). The s(CASP) system is being used by many groups around the world for applications ranging from automating legal reasoning, to medical treatment automation, to software assurance. The s(CASP) system has also been re-implemented within the renowned SWI-Prolog system.

Both ASP and s(CASP) are based on the stable model semantics. Where most implementations of ASP are SAT-solver based, s(CASP) is a top-down, goal-directed system. It can be described as Prolog with stable model semantics-based negation. There are plenty of differences between Prolog, ASP, and s(CASP), but also plenty of overlap.

```
1 father(adam,bill).
2 father(adam,brian).
3 father(bill,charlie).
4 mother(alice, bill).
5 mother(alice, brian).
6 mother(briana, charlie).
```

```
parent(X,Y) :- father(X,Y).
parent(X,Y) :- mother(X,Y).
ancestor(X,Y) :- parent(X,Y).
ancestor(X,Y) :- parent(X,Z), ancestor(Z,Y).
```

Here is a program that runs on all three systems, but lacks a central feature that illustrates the difference between Prolog and ASP: negation-as-failure. Negation-as-failure is a form of negation that allows us to assume the negation of something we cannot prove, and allows us to depend on the absence of information. While negation-as-failure is at the core of ASP and s(CASP), it is an extension to Prolog. Let's add a few rules to our example.

```
% s(CASP)/ASP
has_sibling(X) :- parent(P,X), parent(P,Y), X \= Y.
only_child(X) :- parent(P,X), not has_sibling(X).
```

```
%Prolog
has_sibling(X) :- parent(P,X), parent(P,Y), X\=Y.
only_child(X) :- parent(P,X), \+ has_sibling(X).
```

Prolog uses a different syntax for negation-as-failure. In the example so far this is the only difference, but if we look deeper in how negation-as-failure is implemented we will find bigger differences. The previous version of the example is what is known as *safe*. A safe program is one for which every variable appears in at least one positive goal (a call to a predicate that is not negated). This is required by ASP since it is grounded before being solved, but is not required for s(CASP) and Prolog.

```
% s(CASP)
has_sibling(X) :- parent(P,X), parent(P,Y), X \= Y.
only_child(X) :- not has_sibling(X).
```

```
%Prolog
has_sibling(X) :- parent(P,X), parent(P,Y), X\=Y.
only_child(X) :- \+ has_sibling(X).
```

Though it will run in both s(CASP) and Prolog, the results will be a bit different. This is because s(CASP) has constructive negation. When an unbound variable is used in a negative goal it will be bound to (or constrained against) whatever ensures negation succeeds.

If we were to run the query `?- only_child(X).` we will have different results from s(CASP) and Prolog. In Prolog the query will fail. The goal `only_child(X)` calls `\+ has_sibling(X)` which in turn calls `has_sibling(X)`. Since `brian` and `bill` share the same parents, the call will succeed making `\+ has_sibling(X)` fail.

In s(CASP) the query will succeed and there are two possibilities for `X`. One possibility will bound `X` to `charlie`. The second possibility will leave `X` unbound but constrained against all possible children. This happens due to the nature of negation-as-failure. Since if `X` is bound to anything other than the possible children, then we cannot prove `X` has a sibling.

Disunification in s(CASP) is also constructive. If we query `?- X\=brian, only_child(X)` in s(CASP) `X` will be constrained against `brian` and X will unify with `charlie` through `only_child(X)`. The query will succeed. In Prolog, however, `X\=brian` will fail because this is the same as `\+ (x=brian)`.

Due to implementation issues, an unbound variable cannot be disunified with another unbound variable. Although not consistent with the semantics, s(CASP) will fail at these points to allow execution to continue.

1.1 Dual Program

Constructive negation is made possible by the *dual program* of a s(CASP) program. Rules expressing the constructive negation of the predicates in the original ASP program are synthesized by the s(CASP) compiler.

```
1 % Original Rule
2 p(X,Y) :- q(X),not r(Y).
3
4 % Dual Rule
5 not p(X,Y) :- not q(X).
6 not p(X,Y) :- q(X),r(Y).
```

It provides a means to constructively determine under which conditions and constraints calls to non-propositional predicates featuring variables would have failed. If we want to know when a rule such as the one on line 2 succeeds, the dual program computes the constraints on `Y` under which the call `r(Y)` would fail. Notice that on line 6, we must prove `q(X)` is true. Goals that appear, in the original rule, before the goal we will negate are kept when generating the dual rules. It is possible that the truth value, of the goal to be negated, to depend on the bindings of earlier goals.

The dual rule mechanism is an extension of the usual ASP semantics compatible with programs that can be finitely grounded.[1]

The union of the original program, the dual program, and global constraints (Section 2.4) can be output by invoking `scasp --code`.

1.1.1 Forall

Consider the following rule.

```
1 p :- q(X).
```

[1] Note that, in the presence of function symbols and constraints on dense domains, this is in general not the case for s(CASP) programs.

The predicate `p` is true if there exists a possible binding for `X` such that `q(X)` is true. As can be seen from this explanation, logically, `X` is existential. The dual would then need to state that `p` is not true if for all possible bindings for `X`, `q(X)` is not true.

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

This is represented with a special internal predicate `forall/2`. The `forall(X,not q(X))` will succeed if it can prove that `not q(X)` is true for all possible bindings of `X`. We will go into more detail in chapter 4.

1.1.2 Example

Next, let's consider a full example.

```
edge(a,b).
edge(b,c).
edge(b,d).
edge(c,e).
edge(d,a).

path(X,Y) :- edge(X,Y).
path(X,Y) :- edge(X,Z),path(Z,Y).
```

The `path/2` predicate can calculate whether or not there is a path from `X` to `Y` in the directed graph specified by the `edge/2` predicate. Given the above program, s(CASP) would generate the dual program below.

```
not edge(Var0,Var1) :-
     not o_edge_1(Var0,Var1),
     not o_edge_2(Var0,Var1),
     not o_edge_3(Var0,Var1),
     not o_edge_4(Var0,Var1),
     not o_edge_5(Var0,Var1).

not o_edge_1(Var0,Var1) :-
     Var0 \= a.
not o_edge_1(Var0,Var1) :-
     Var0 = a,
     Var1 \= b.

not o_edge_2(Var0,Var1) :-
     Var0 \= b.
not o_edge_2(Var0,Var1) :-
     Var0 = b,
     Var1 \= c.

not o_edge_3(Var0,Var1) :-
     Var0 \= b.
```

```
22 not o_edge_3(Var0,Var1) :-
23       Var0 = b,
24       Var1 \= d.
25
26 not o_edge_4(Var0,Var1) :-
27       Var0 \= c.
28 not o_edge_4(Var0,Var1) :-
29       Var0 = c,
30       Var1 \= e.
31
32 not o_edge_5(Var0,Var1) :-
33       Var0 \= d.
34 not o_edge_5(Var0,Var1) :-
35       Var0 = d,
36       Var1 \= a.
37
38 not path(Var0,Var1) :-
39       not o_path_1(Var0,Var1),
40       not o_path_2(Var0,Var1).
41
42 not o_path_1(Var0,Var1) :-
43       not edge(Var0,Var1).
44
45 not o_path_2(Var0,Var1) :-
46       forall(Var2,not o_path_2(Var0,Var1,Var2)).
47
48 not o_path_2(Var0,Var1,Var2) :-
49       not edge(Var0,Var2).
50 not o_path_2(Var0,Var1,Var2) :-
51       edge(Var0,Var2),
52       not path(Var2,Var1).
```

In the dual rules, `Var0` corresponds to `X`, `Var1` to `Y`, and `Var2` to `Z` as referenced in the original rules.

Now, suppose we want to know if there is not a path from `c` to `d`. We can query `?- not path(c,d)`. When the dual rule is called, `c` will be bound to `Var0`, and `d` will be bound to `Var1`. The call to `not path(c,d)` results in the call to `not o_path_1(c,d)` and then `not o_path_2(c,d)`. Both of these predicates were internally generated to facilitate constructive negation. The call to `not o_path_1(c,d)` ensures there is no edge between `c` and `d`. In this case, there is not.

There are two versions of `not o_path_2`. The three parameter version of `not o_path_2` tries two things. The first rule, checks to see if there is no edge from `Var0` to `Var2` (equivalent to `X` and `Z`, resp.). The second rule, checks to see if there is no path from `Var2` to `Var1` (equivalent to `Y` and `Z`,resp.).

The two parameter `not o_path_2` uses a forall. The forall will call the three parameter version of `not o_path_2`, using `Var2` as the unbound forall variable. In other words, s(CASP) will check that, for all possible values of `Var2` (`Z` in the original rule), there is no edge from `c` to `Var2` or there is no path from `Var2` to `d`.

There is only one edge from `c`, and that is to `e`. So, we must use the second rule to make sure there is no path from `e` to `d`. Since there is no edge from `e`, the recursive call to `not path(Var2,Var1)` will succeed. Go ahead and trace through the recursive `not path(Var2,Var1)` and see how it succeeds.

Chapter 2

Cycles

2.1 Introduction

The semantics of s(CASP) allow the meaning of a program to be affected by the cycles in the program. This means that cycles are detected and processed, unlike traditional prolog where such cycles will lead to nontermination. This detection, however, is not perfect as we will see later.

Cycles in s(CASP) can be divided into three categories.

Positive cycles form when the proof of a predicate depends on that predicate and there are no negations. As an example consider what follows.

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

In this program `p` is true if `q` is true, which in turn is true if `p` is true.

Even cycles form when the proof of a predicate depends on on that predicate and there are an even number of intervening negations.

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

In this program, `p` is true if `q` is not true. From the dual of `q` (not shown here), `q` is not true if `r` is not true. From the dual for `r`, `r` is not true if `p` is true. These three rules form a cycle, and we can count the number of negations (two), which is even.

Odd cycles form when the proof for a predicate depends on the predicate's negation. Thus there are an odd number of intervening negations.

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

In this program, p is true if q is true. From rule 2, q is true if r is not true, and from the dual r is not true if p is not true. So, p is true if p is not true. Odd cycles always form contradictions.

Each category of cycles has a different affect on the meaning of the program.

2.2 Positive Cycles

A positive cycle is a loop that contains no negations. In Prolog, such cycles add nothing to the program's model, but in practice will result in non-termination. In s(CASP), these cycles are detected and fail to be consistent with Prolog semantics.

```
1 jill_eats :- jack_eats.
2 jack_eats :- jill_eats.
```

Consider the query, `?- jill_eats`. Then, Jill eats if Jack eats, and Jack eats if Jill eats. We started with `jill_eats` and determine that its truth value depends on itself. So, we fail. Likewise, if we queried `?- not jill_eats` it will succeed. This works with variables, as well.

```
1 jill_eats(X) :- jack_eats(X).
2 jack_eats(X) :- jill_eats(X).
```

If we query `jill_eats(X)` the `X` in rule 1 will unify with the `X` in rule 2. We can see this better by grounding a variable.

Consider the query `?- jill_eats(mexican_food).` From rule 1, `X` unifies `mexican_food`. So, `jill_eats(mexican_food)` is true if `jack_eats(mexican_food)` holds and `jack_eats(mexican_food)` is true if `jill_eats(mexican_food)` holds. This is no different if the unification happens in the course of executing a rule.

```
1 jill_eats(mexican_food) :- jack_eats(X).
2 jack_eats(X) :- jill_eats(X).
```

The query `?- jill_eats(X)` will fail. The variable `X` in the query will unify with `mexican_food`. The result, `jill_eats(mexican_food)`, is true if there exists some other `X` such that `jack_eats(X)` is true and that is true if `jill_eats(X)` is true. If we differentiate between the first `X` and the second, not considering them the same despite both unifying with `mexican_food`, We may face non-termination. In this example that is exactly what would happen.

We could instead use unification. In this way the two `X`'s will be considered to be the same, but this approach is flawed.

```
1 p(0).
2 p(X) :- p(Y).
```

If we query `p(1)`, `X` in rule 2 will be bound to `1` and `p(1)` will be true if `p(Y)` is true for some `Y`. However, if we are detecting cycles based on unification, `p(Y)` will unifiy with `p(1)` and a positive cycle is detected. To remedy this we use a structural way of determining a match, called an *exact match*. Continuing the example, `Y` is an unbound variable and `X` is bound to `1`. They are not an exact match. So, `p(Y)` is called, and `Y` is bound to `0`. If there were indeed a positive cycle, two terms with the exact same structure will eventually appear in the sequence of calls.

The exact match allows us to avoid some problems like the one above, but it is not perfect. Due to the way goals are executed, s(CASP) falsely detects a positive cycle.

Suppose we queried `X\=0,p(X)`, instead of `p(1)`. Now, `X` and `Y` will have the same structure. Consider the following variation.

```
1 p(0).
2 p(X) :- p(Y),X=1.
```

In this program, the goal `p(Y)` can use the first rule to bind `Y` to zero, and `X` will be bound to one. This is not a positive cycle, but s(CASP) cannot detect this. When it is processing the goal `p(Y)` it does not know about the other rule, nor about the second goal, `X=1`. So, it will fail.

2.2.1 Uses of Positive Cycles

Positive cycles are more of an artifact of the semantics, and so far we have not found much in the way of intentional uses for positive cycles. We can however take advantage of the fact that s(CASP) fails for positive cycles to avoid loops in our code. Here is an example.

```
1 edge(a,b).
2 edge(b,c).
3 edge(X,Y) :- edge(Y,X).
```

If we tried to find all the solutions for, say, `edge(a,X)` in prolog, this would result in nontermination. The third rule results in unbound recursion. In s(CASP) however, when calling `edge(Y,X)`, using rule 3 will result in a failure, terminating the search.

2.3 Even Cycles/ Even Loops Over Negation (ELON)

Even cycles are loops with an even number of negations. When an even cycle is detected, we assume success. The goal that depends on that even cycle will always succeed. Stable model semantics is a multi-world semantics. Even cycles determine the possible worlds (or models) that can be generated. So a predicate `p` in an even cycle will be true in one world and false in another.

```
jack_eats(X):- not jill_eats(X).
jill_eats(X):- not jack_eats(X).
```

If we query `?- jill_eats(X)`, `jill_eats(X)` will be true if `jack_eats(X)` is not true. From the dual program (see section 1.1), `jack_eats(X)` will be false if `jill_eats(X)` is true. So, `jill_eats(X)` depends on itself and there are two intervening negations: one in rule 2 and one in rule 1. In this case, we assume `jill_eats(X)` is true.

Similarly if we run the query `?- jack_eats(X)`, it will expand into `not jill_eats(X)` which in turn `not (not jack_eats(X))` where in we assume `jack_eats(X)` is true.

We have two possibilities: either `jill_eats(X)` is true or `jack_eats(X)` is true. This enables the creation of multiple possible models. Odd loops (below) can constrain the generated possible models.

Even cycles are one of the ways of introducing nondeterminism. It can be used to model uncertainty and multiple scenarios, among others.

Suppose I want a program that models getting food.

```
cook_something :- ingredients_at_home, not eat_at_restaurant.
eat_at_restaurant :- not cook_something.
ingredients_at_home.
```

In this simple program, we can see that `cook_something` and `eat_at_restaurant` form an even cycle if `ingredients_at_home` is true. So, this program has two models (or *worlds*). In one world I cook something to eat and in another I eat at a restaurant. I can then add to the program in such a way that it depends on what world I am in. For instance, I can add a rule that says I save money if I do not eat at a restaurant.

```
save_money :- not eat_at_restaurant.
```

Now, if I query `save_money`, it will succeed – with the assumption that I cook something. Since it is only true in that world. Also notice that if the fact for `ingredients_at_home` is removed, the even cycle no longer exists. This is because the body of rule 1 will be false. So, `cook_something` must be false. Therefore there is only one world. The one where I eat at a restaurant.

Abduction can also be realised through the use of even cycles. Consider the following.

```
buy_meal :- hot_day, cold_meal.
buy_meal :- cold_day, hot_meal.
buy_meal :- not hot_day, not cold_day, meal_cheap.
:- hot_day, cold_day.
```

Suppose we want to decide the type of meal and its price. We know the customer will buy a meal if it is a cold meal on a hot day or a hot meal on a cold day. Otherwise,

the customer will only buy the meal if it is cheap. Our intention is for `hot_day` and `cold_day` to be provided as facts depending on the day. A constraint ensures we don't make the day both cold and hot. The predicates `cold_meal`, `hot_meal`, and `meal_cheap` are to be *abducibles*.

```
meal_cheap :- not n_meal_cheap.
n_meal_cheap :- not meal_cheap.
cold_meal :- not hot_meal.
hot_meal :- not cold_meal.
```

Abducubles are represented in s(CASP) with simple even cycles. In this example the even cycles create four possible worlds. We have added an auxiliary predicate `n_meal_cheap` to represent when `meal_cheap` is false. For `cold_meal` and `hot_meal`, since they are mutually exclusive, we use them as each others opposites, just as we did with `cook_something` and `eat_at_restaurant` earlier. So, no new predicate is needed. Abduction is common in s(CASP). So, a directive (`#abducible`) is provided to do this work for us.

```
#abducible meal_cheap.
cold_meal :- not hot_meal.
hot_meal :- not cold_meal.
```

We could also use the directive with `cold_meal` and `hot_meal`, but then we would require a global constraint to ensure they are mutually exclusive. Now, we can provide the input as facts, and query `buy_meal` to determine what type of meal we should sell.

```
cold_day.
?- buy_meal.
```

The query result in the model `{buy_meal,cold_day,hot_meal}`. All other possible worlds are not generated because they require `cold_day` to be false, and we know to prepare hot meals to sell. In chapter 7, we discuss other specific techniques that use even cycles for representing knowledge.

2.4 Odd Cycles/ Odd Loops Over Negation (OLON)

Odd cycles represent contradictions. It requires that something be both true and false. The following is a particularly pathological case.

```
p :- not p.
```

In this program `p` is true only if it is not true – a contradiction. In stable model semantics, and therefore in s(CASP), a contradiction means there is no model, regardless of the rest of the program. If we were to add the above rule to any other s(CASP) program that does not contain additional rules for `p`, every query will fail. There is no model.

The odd cycle only happens, however, when a goal's success depends on it's negation.

If we were to add the fact `p.` to the above program, then the odd cycle becomes irrelevant. There is a rule that states `p` is true. Additionally, all other goals in the body of the rule that is part of the odd cycle must succeed. Otherwise, the head is false and the cycle is w relevant. We can use this for contradiction based reasoning – allowing us to define global constraints for the program.

```
p(X):- target(X), not p(X).
```

If `target(X)` is false, then the conjunction will be false, regardless of the truth value of pX)). So, `p(X)` must also be false. If `target(X)` is true, however, the value of the conjunction is dependent on `not p(X)` – forming an odd cycle. So for there to be a model, `target(X)` must be false.

Writing out an odd cycle every time we want to create a global constraint can clutter the code. Borrowing from traditional ASP, s(CASP) allows for such constraints to be specified with a headless rule.

```
:- target(X).
```

This has the meaning of `false :-target(X).`. If `target(X)` is true, for any binding of `X`, then we have true implies false – also a contradiction. Explicit constraints on the models of a program will typically be done with a headless rule. Odd cycles, however can still arise in the program due to the nature of the knowledge you are modeling. We will see an example of this in the big example in section 2.5.

Global constraints due to odd cycles or headless rules can be combined with even cycles when working with multiple worlds. Even cycles can generate potential worlds, and odd cycles can *kill* invalid worlds. Consider a situation where we want to now who will be attending a given event.

```
#abducable alice_goes
#abducable charlie_goes

bob_goes :- alice_goes.

:- alice_goes, charlie_goes.
```

In this program we specify `alice_goes` and `charlie_goes` as abducibles, which generate even cycles as explained in the previous section. So, there are four possible worlds – all possible combination of Alice and Charlie going or not going. Bob will go only if Alice goes. Finally, we have a headless rule that specifies a global constraint: Alice and Charlie will not both go. The body of the headless rule must be false for there to be a model. Let's consider two different queries.

First, if we query `bob_goes` then we will have to prove `alice_goes`. Since `alice_goes` is part of an even cycle, s(CASP) will end up assuming `alice_goes` is true. Then the global constraint must be checked. Since `alice_goes` is true, `charlie_goes` must be false. The final model is `{bob_goes,alice_goes,not charlie_goes}`.

On the other hand, if we directly query `charlie_goes`, s(CASP) will assume `charlie_goes` is true, because it is part of an even cycle. Then the global constraints will require `alice_goes` to be false. We end up with `{charlie_goes,not alice_goes}`. Since s(CASP) produces partial models nothing about `bob_goes` is reported, but it will also be false in the model.

Each headless rule or rule involved in an odd loop is called an OLON (odd loop over negation) rule. Constraints imposed on the program by an odd loop or headless rule are enforced by a constraint check, called the NMR check, appended to all queries. A sub-check rule is created for each OLON rule. If an OLON rule has any variables in the head, the corresponding variables in the sub-check will also be in the head.

```
% Original program
p(X):- condition_1(X), not p(X).
p(X):- not condition_2(X), not p(X).

% Subchecks
chk_p1(X):- not condition_1(X).
chk_p1(X):- p(X).
chk_p2(X):- condition_2(X).
chk_p2(X):- p(X).
```

Simply, a sub-check states the head of a rule is true or its body is false. The heads of these sub-checks are added the body of the NMR check rule responsible for enforcing these constraints. The forall/2 predicate (see Section tut-sec:forall) is used to ensure the sub-check holds for all bindings of its head variables.

```
nmr :- forall(X,chk_p1(X)), forall(X,chk_p2(X)).
```

A call to the NMR check is appended to each query.

2.5 Example

Suppose we have a group of people, and we want to decide who will be going on a trip. Each person, may or may not have a passport, and may or may not be busy. In addition, some people have an assistant that can go on trips for them. Each person can choose if they want to go on the trip, but only if the following is true.

- The person has a passport.
- The person is not busy.
- A assistant cannot go if someone they assist goes.

Putting these together is fairly straight forward.

```
go_on_trip(X) :- person(X),passport(X),
                 not busy(X),
                 not has_conflict(X),
                 not stay(X).
stay(X) :- not go_on_trip(X).
```

We model the person's choice as an even cycle. The person can either go on the trip or stay. In order to go on the trip, however, the person must have a passport, not be busy, and have no conflicts. The predicates `person/1`, `passport/1`, and `busy/1` are considered "input" predicates. Before the program is ran, these should be added to the program.

The predicate `has_conflict/1` is a constraint on `go_on_trip/1`. We need to define it.

```
has_conflict(X) :- assistant(X,Y), go_on_trip(Y).
```

The predicate `assistant/2` should be provided as "input", where `assistant(X,Y)` means `X` is `Y`'s assistant. A person has a conflict if they are an assistant, and someone they assist is going. This is fairly straight forward, but actually forms an odd cycle.

This constraint on `go_on_trip/1` implies that a person cannot be their own assistant or such a person cannot go for some other means. This arises naturally from the logic. If this is not what we meant, then we could add `X\=Y` after `assistant(X,Y)` to specify that it applies only when `X` and `Y` are different.

Either way, an NMR check will be generated. Since s(CASP) only looks at the goal `go_on_trip(Y)`, and does not know whether `X` and `Y` can be the same. So, to be safe, s(CASP) generates the NMR check. Note, that an NMR subcheck for a rule that is not an OLON rule will not affect the result of a query.

Chapter 3

CLP($\mathbb{Q}$) in s(CASP)

3.1 Introduction

Constraint logic programming is possible in s(CASP), and differs slightly from Prolog since s(CASP) has constructive negation. Consider a simple example.

```
1 p(a).
```

The query `?- not p(X)` returns the binding `X \= a` and the model `{not p(X | {X \=a})}`, representing the set of *not* $p(X)$ such that the atom $p(X)$ can be proven only when `X \= a`.[1] Note that `X` in the query appeared only in a negated atom, and did not need to be part of any non-negated atom. The query makes the program not *safe*. This would not work in traditional ASP systems.

This is augmented in s(CASP) with constraint processing capabilities. The s(CASP) system has a generic interface to enable plugging in constraint solvers. s(CASP) currently includes the CLP($\mathbb{Q}$) linear constraints solver by Holzbaur (1995), that supports the arithmetic constraints $<, >, =, \leq, \geq$. These arithmetic constraints are written as `#<` , `#>` , `#=` , `#=<` , `#>=` to distinguish them from Prolog's arithmetic operators. The implementation of s(CASP) under Ciao Prolog also support writing these arithmetic constraints as `#<`, `#>`, `#=`, `#=<`, `#>=` (which is the one we will use for the rest of the document). The linear constraints can be used to implement programs that require comparisons of points in (continuous) time and to solve the equations that arise from these comparisons (even when the variables involved are not instantiated).

```
1 p(X):- X #> 0.
```

For the query `?- not p(X)`, this program will return the model `{not p(X | {X #=< 0})}`. The notation `V|{C}` for a variable `V` is intended to mean that `V` is subject to the

[1] Uniqueness of names is assumed for constants and function names: any two constants or functions with different names represent different objects.

constraints in `{C}`.

The selection of CLP(ℚ) instead of the faster CLP(ℝ) is motivated by soundness. Since CLP(ℝ) internally uses floating-point numbers, rounding and approximations compromise accuracy and termination of some code. On the other hand, CLP(ℚ) represents rational numbers exactly and therefore it should not introduce any calculation errors. One example in which the use of floating-point numbers would be inadequate is the following clause which uses the factor `4/3` and that does not have an exact floating-point representation.

```
trajectory(filling,T1,level(X2),T2):-
    T1 #< T2,
    X2 #= X + 4/3*(T2-T1),
    max_level(Max),
    X2 #=< Max,
    holdsAt(level(X),T1).
```

The following code, from (Arias et al. 2022:Pag. 9) has variables that would be termed as *unsafe* in regular ASP systems: variables that appear in negated atoms in the body of a clause, but that do not appear in any positive literal in the same body.

```
p(X):- q(X, Z), not r(X).
p(Z):- not q(X, Z), r(X).

q(X, a):- X #> 5.
r(X):- X #< 1.
```

Since s(CASP) synthesizes explicit constructive goals for these negated goals, the aforementioned code can be run as-is in s(CASP). The query `?-p(A).` generates three different answer sets, one for each binding:

```
{ p(A| {A #> 5}), q(A| {A #> 5}, a), not r(A| {A #> 5}) }
  A #> 5
{ p(A| {A \= a}), not q(B| {B #< 1}, A| {A \= a}), r(B| {B #< 1}) }
  A \= a
{ p(a), not q(B| {B #< 1}, a), r(B| {B #< 1}) }
  A = a
```

The constraints `A #> 5`, `A \= a`, and `A = a` correspond to the bindings of variable `A` that make the atom from the query `?-p(A)` belong to the stable model.

3.2 Examples

In this section, we provide some examples of that use CLP(ℝ). The first three are variations of each other, and makes use of universally quantified variables. There is a brief explanation of the *forall* mechanism in the introduction, and details in chapter 4.

3.2.1 p/2 covers the whole domain

```
1 p(X,Y) :- X #=< Y.
2 p(X,Y) :- X #> Y.
3
4 q :- not p(X,Y).
5
6 ?- not q. %% Should succeed
```

The query `not q` will succeed. The call to `not q` will execute the dual rule for `q`. The dual rule will try to prove `p(X,Y)` for all possible bindings of `X` and `Y`. This will involve calling `p(X,Y)`. The first rule will constrain `X` and `Y` such that `X` must be less than or equal to `Y`. This proves that for all `X` less than or equal to `Y` `p(X,Y)` is true. The second rule will succeed with `X` greater than `Y`. Putting these together, `p(X,Y)` is true for all possible bindings of `X` and `Y`. The presence of the constraints reduces the domain to only rational numbers. So, we do not need to consider other possible bindings.

3.2.2 p/2 does not cover the whole domain

```
1     p(X,Y) :- X #=< Y.
2
3     q :- not p(X,Y).
4
5     ?-  not q. %% Should fail
```

In this example `p/2` does not holds for `X #> Y` (the corresponding clause is missing), therefore, the query `?- not q` fails.

3.2.3 p/2 almost covers the entire domain

```
1 p(X,Y) :- X #< Y.
2 p(X,Y) :- X #> Y.
3
4 q :- not p(X,Y).
5
6 ?-  not q. %% Should fail
```

In this example the query `?- not q` also fails because `p/2` does not hold for `X #= Y`.

3.2.4 Very complex example...

```
1 p(X, Y, Z):- all_pos(X, Y), all_pos(Y, Z), all_pos(X, Z).
2
```

```
3 all_pos(A, B):- A #> B.
4 all_pos(A, B):- A #=< B.
5
6 q:- not p(X, Y, Z).
7
8
9 ?-  not q. %% Should succeed
```

This is a complex example. The predicate `all_pos/2` is short for "all possible". It checks to see if `A` is greater than `B` or if it is less than or equal to `B`. This means `all_pos/2` will always succeed, but with two different partial models. The predicate `p/3` does a pair-wise check of its parameters against `all_pos/2`. So, `p/3` will be true if there is some combination of relations between `X`, `Y`, and `Z`. For instance, `X #> Y, X#>Z, Y#>Z` is one such valid combination. The query is true, if every possible combination is valid.

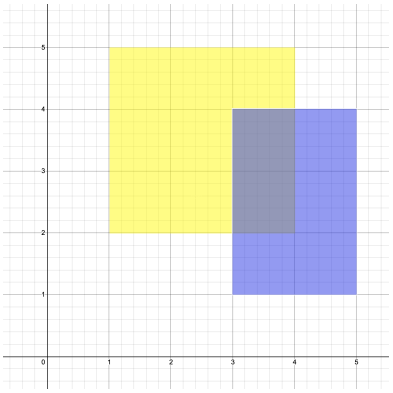
Note that depending on the clauses for `all_pos/2` selected to evaluate `p(X,Y,Z)` there are $3 * 2^3$ possible combinations for the variables `X,Y,Z` that the predicate *forall/2* has to check. Use the flag `--tree --long` to output the corresponding justification trees (see Section 4 for details).

3.2.5 Spatial reasoner

Finally, we present a complex example where operations over 2D objects are implemented using the logical operators: $\wedge$, $\vee$, and $\neg$.

```
1  % Union = ShA ∪ ShB
2  shape_union(IdA, IdB, convex([X,Y])) :-  convex(IdA,X,Y).
3  shape_union(IdA, IdB, convex([X,Y])) :-  convex(IdB,X,Y).
4  % Intersection = ShA ∩ ShB
5  shape_intersect(IdA, IdB, convex([X,Y])) :-
6       convex(IdA,X,Y), convex(IdB,X,Y).
7  % Complement = ¬ ShA
8  shape_complement(IdA, convex([X,Y])) :- not convex(IdA,X,Y).
9  % Subtract = ShA ∩ ¬ ShB
10 shape_subtract(IdA, IdB, convex([X,Y])) :-
11      convex(IdA,X,Y), not convex(IdB,X,Y).
```

Example 1. Let us consider as an example two rectangles (see figure on the right) which are represented with the following rules:



```
convex(r1, X, Y) :-
X#>=1, X#<4, Y#>=2, Y#<5.
convex(r2, X, Y) :-
X#>=3, X#<5, Y#>=1, Y#<4.
```

The application of two of the previous operations generates the following results:

```
?- shape_intersect(r1,r2,Int).
   Int = convex([A | { A#>=3, A#<4 }, B | { B#>=2, B#<4 }]) ?

?- shape_subtract(r1,r2,Sub).
   Sub = convex([A | { A#>=1, A#<3 }, B | { B#>=2, B#<5 }]) ? ;
   Sub = convex([A | { A#>=3, A#<4 }, B | { B#>=4, B#<5 }]) ?
```

Chapter 4

The Forall Mechanism

4.1 Introduction

Earlier chapters have mentioned the requirement for universally quantified variables in dual rules. The forall mechanism allows us to implement the quantifier operationally. Consider the following program with the dual rule for `p/0`.

```
1 p :- not q(X).
2 q(X) :- X=a.
3 q(X) :- X \= a.
4 not p :- forall(X, not p1(X)).
5 not p1(X) :- q(X).
```

Under the query `?- not p`, the interpreter will execute `forall(X,not p1(X))` with `X` unbound to determine if `not p1(X)` is true for all possible values of `X`, possibly within a constraint domain.

There have been a few different forall algorithms used with s(CASP), and any of these algorithms can be used as the forall mechanism of an execution by a flag. The current default forall algorithm is *C-forall* – named for its ability to handle CPL(Q) constraints.

The core idea is to iteratively narrow the store (set of constraints) under which our goal (second argument) is executed. The goal is executed, and one answer is selected. The goal is then re-executed but we use a modified store. We remove the constraints on the forall variable (first argument), adding constraints to avoid the first answer's constraints, but keep the new constraints on all other variables.

In our example above, the call to `not p1(X)` will bind `X` to `a`. We make the call a second time, but first constrain `X` so that it cannot be `a`. The first rule for `q/1` will then fail, but the second one will succeed. There is no change to the domain of `X`. It is still `X\=a`, and so we terminate with success.

C-forall terminates with success

Here is a generic visual example of the *C-forall* algorithm succeeding.

The answers $A_1, \ldots, A_4$ to the goal cover the whole domain, represented by the square. Therefore, *C-forall* should succeed. The answer constraints that the program can generate are depicted in picture (a). For simplicity in the pictures, we will assume that the answers A_i only restrict the domain of the forall variable, so it will not be necessary to deal with other variables separately since the constraints for all other variables will always be empty. Picture (b) shows the result of the first iteration of *C-forall* starting with constraint store $C_1 = \top$: answer A_1 is more restrictive than C_1 and therefore $C_2 = C_1 \wedge \neg A_1$ (in grey) is constructed. Picture (c) shows the result of the second iteration: the domain is further reduced. Finally, in picture (d) the algorithm finishes successfully because $A_3 \equiv C_3$, i.e., A_3 covers the remaining domain. Note that we did not need to generate A_4.

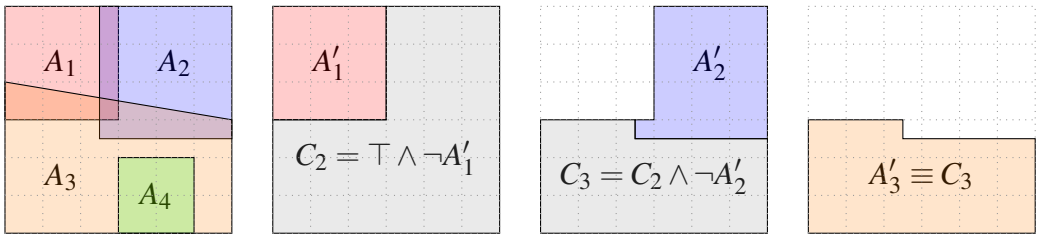


Figure 4.1: A *C-forall* evaluation that success.

C-forall terminates with failure

The following shows an example where the answer constraints do not cover the domain and therefore *C-forall* ought to fail. Again, we assume that the answers A_i only restrict the domain of the forall variable.

Picture (a) depicts the answer constraints the goal can generate. Note the gap in the domain not covered by the answers. Pictures (b) to (d) proceed as in the previous example. Picture (d) shows the final step of the algorithm: the execution of the goal under the store $C_4 = C_3 \wedge \neg A'_3$ fails because the solution A_4 does not have any element in common with C_4, and then *C-forall* also fails.

4.1.1 Examples

```
1 jack_eats(X):- not jill_eats(X,Y).
```

This program is equivalent to the following logical formula.

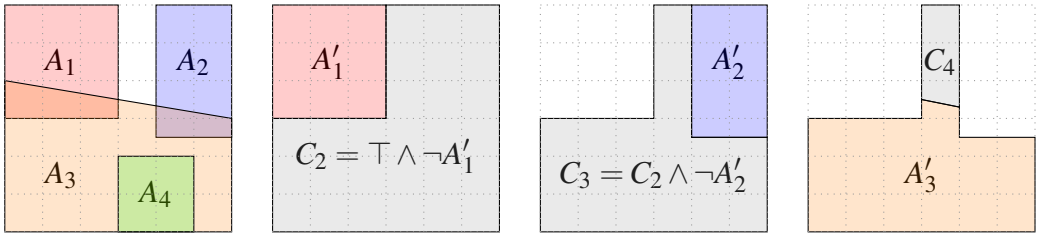


Figure 4.2: A *C-forall* evaluation that fails.

$$\forall X(jack\_eats(X) \iff \exists Y \neg jill\_eats(X,Y)). \tag{4.1}$$

The dual is computed by negating the right hand side.

$$\forall X \neg jack\_eats(X) \iff \forall Y\, jill\_eats(X,Y)). \tag{4.2}$$

The body variables, originally existentially quantified, are now universally quantified. This is represented by using forall/2, an internal predicate not available to the user.

```
% Original program
jack_eats(X):- not jill_eats(X,Y).

% Dual program
not jack_eats(X) :- forall(Y, jill_eats(X,Y)).
```

The predicate `forall/2` succeeds if the goal is true for all values of the variable. The goal in the second parameter is called. Upon success,

- if the variable is unbound, the forall succeeds, or
- if the variable is bound, failure and backtracking occur, or
- if the variable is negatively constrained (`X \= a`), the goal is executed for each constrained value. The forall succeeds only if all such calls succeed.

4.1.2 Versions of forall

There are 4 versions of `forall/2` available in s(CASP).

- `prev_forall` is the current default version but has limitations in the presence of constraints.
- `sasp_forall` is the algorithm used in the original s(ASP) system s(CASP) is based on (has slight differences w.r.t. `prev_forall`).

- `c_forall` is a preliminary implementation of `forall/2` that supports CLP($\mathbb{Q}$) however, it discards solutions –use with care.
- `all_c_forall` is a more exhaustive version of `c_forall`.

The versions `prev_forall` and `sasp_forall` are more efficient than `c_forall` or `all_c_forall` but they do not support constraints. The `all_c_forall` has a tendency to generate many redundant solutions, but `c_forall` may mistakenly miss a solution in its attempt to eliminate redundancy.

4.1.3 Differences between the forall versions

While `prev_forall` and `sasp_forall` are similar, they differ at the point at which they cut off the search for justifications. Consider the following examples to compare their behavior.

Example 2 (max.pl).

```
1 max(X) :- not smaller(X).
2 smaller(X) :- num(X), num(Y), X < Y.
3
4 num(1).
5 num(2).
6 num(3).
7 num(5).
8
9 ?-  max(C).
```

Using `sasp_forall` we obtain, as expected, two partial models.

```
% QUERY:?- max(C).

        ANSWER: 1 (in 0.402 ms)

MODEL:
{ max(C | {C \= 1,C \= 2,C \= 3,C \= 5}),  not smaller(C |
{C \= 1,C \= 2,C \= 3,C \= 5}),  not num(C | {C \= 1,C \= 2,C \= 3,C \= 5}) }
BINDINGS:
C \= 1,C \= 2,C \= 3,C \= 5 ? ;


        ANSWER: 2 (in 7.01 ms)

MODEL:
{ max(5),  not smaller(5),  num(5),  not num(Var1 |
{Var1 \= 1,Var1 \= 2,Var1 \= 3,Var1 \= 5}),  num(1),  num(2),  num(3) }
BINDINGS:
C = 5 ? ;
```

Using `prev_forall` results in only the first model being generated.

```
% QUERY:?- max(C).

        ANSWER: 1 (in 0.377 ms)

MODEL:
{ max(C | {C \= 1,C \= 2,C \= 3,C \= 5}),  not smaller(C |
{C \= 1,C \= 2,C \= 3,C \= 5}),  not num(C | {C \= 1,C \= 2,C \= 3,C \= 5}) }
BINDINGS:
C \= 1,C \= 2,C \= 3,C \= 5 ? ;
```

However, for the query `?-max(5)`, both implementations give the same model.

```
% QUERY:?- max(5).

        ANSWER: 1 (in 2.492 ms)

MODEL:
{ max(5),  not smaller(5),  num(5),  not num(Var0 |
{Var0 \= 1,Var0 \= 2,Var0 \= 3,Var0 \= 5}),  num(1),  num(2),  num(3) }
BINDINGS: ?
```

This is the model not produced by `prev_forall` for the previous query.

Example 3 (Hamiltonian.pl).

```
% fact for each vertex(N).
vertex(0).
vertex(1).
vertex(2).
vertex(3).

% fact for each edge edge(U, V).
edge(0, 1).
edge(1, 2).
edge(2, 3).
edge(3, 0).

edge(2, 0).
edge(0, 3).
edge(3, 1).

reachable(V) :- chosen(U, V), reachable(U).
reachable(0) :- chosen(V, 0).

% Every vertex must be reachable.
:- vertex(U), not reachable(U).

```

```
23 % Choose exactly one edge from each vertex.
24 other(U, V) :-
25     vertex(U), vertex(V), vertex(W),
26     edge(U, W), V \= W, chosen(U, W).
27 chosen(U, V) :-
28     edge(U, V), not other(U, V).
29
30 % You cannot choose two edges to the same vertex
31 :- chosen(U, W), chosen(V, W), U \= V.
32
33 # show chosen/2.
34 ?- chosen(1,2).
```

Using `prev_forall`, we obtain, as expected, two partial models.

```
% QUERY:?- chosen(1,2).

        ANSWER: 1 (in 1752.031 ms)

MODEL:
{ chosen(1,2), chosen(3,0), chosen(0,1), chosen(2,3) }
BINDINGS: ? ;


        ANSWER: 2 (in 1761.38 ms)

MODEL:
{ chosen(1,2), chosen(2,0), chosen(3,1), chosen(0,3) }
BINDINGS: ? ;
```

Using `sasp_forall` only results in only the first model.

```
% QUERY:?- chosen(1,2).

        ANSWER: 1 (in 1710.433 ms)

MODEL:
{ chosen(1,2), chosen(3,0), chosen(0,1), chosen(2,3) }
BINDINGS: ? ;
```

However, for the query `?-chosen(3,0)`, both implementations would generate the same (and unique) valid model.

```
% QUERY:?- chosen(3,0).

        ANSWER: 1 (in 1720.252 ms)

MODEL:
{ chosen(3,0), chosen(0,1), chosen(1,2), chosen(2,3) }
BINDINGS: ? ;
```

Note that this is the model that the implementation of `sasp_forall` discarded for the query `?- chosen(1,2)`, i.e., the implementation is correct but prunes some (valid) results.

We leave to the reader the task of checking the behavior using `c_forall` and `all_c_forall`, and move on to discuss some examples with constraints. For more information on CLP($\mathbb{Q}$) constraints see Section 3

Using `c_forall` vs. `all_c_forall`

Example 4.
```
p(X,Y) :- X #=< Y.
p(X,Y) :- X #> Y.
q :- not p(X,Y).
```

As we mentioned before, the query `?- not q` succeeds. The goal `not q` is true if `q` is false, and in turn `q` is false if `p(X,Y)` is true for all possible values of `X` and `Y`. The first rule says `p(X,Y)` is true if `X` is less than or equal to `Y`. The second rule says `p(X,Y)` is true if `X` is greater than `Y`. So, `p(X,Y)` is true regardless of the relationship between `X` and `Y`.

Since `--sasp_forall` and `--prev_forall` evaluate the variables one by one the results involving constraints among multiple variables is unsound.

Using `c_forall` we obtain a unique answer with its corresponding justification tree, while using `all_c_forall` two equivalent answers are generated.

```
% QUERY:?- not q.

        ANSWER: 1 (in 0.736 ms)

JUSTIFICATION_TREE:
not q :-
    not o_q_1 :-
        forall(Var4,forall(Var5,not o_q_1(Var4,Var5))) :-
            not o_q_1(Var0,Var1) :-
                p(Var0,Var1) :-
                    Var0 #=< Var1.
            not o_q_1(Var2,Var3) :-
                p(Var2,Var3) :-
                    Var2 #> Var3.
global_constraint.

MODEL:
{ not q,  p(Var0,Var1),  p(Var2,Var3) }
BINDINGS: ? ;
```

```
        ANSWER: 2 (in 0.631 ms)

JUSTIFICATION_TREE:
not q :-
    not o_q_1 :-
        forall(Var4,forall(Var5,not o_q_1(Var4,Var5))) :-
            not o_q_1(Var0,Var1) :-
                p(Var0,Var1) :-
                    Var0 #> Var1.
            not o_q_1(Var2,Var3) :-
                p(Var2,Var3) :-
                    Var2 #=< Var3.
global_constraint.

MODEL:
{ not q,  p(Var0,Var1),  p(Var2,Var3) }
BINDINGS: ?
```

Note that they only differ in the order in which the clauses of `p/2` are used to evaluate the *forall* predicate.

Example 5. The predicate `p/2` does not cover the entire domain:

```
1 p(X,Y) :- X #< Y.
2 p(X,Y) :- X #> Y.
3 q :- not p(X,Y).
```

The query `?- not q` results in no model. This is similar to the previous example, but there is no rule that allows `p(X,Y)` to be true when `X` is equal to `Y`.

As we mentioned before, using `--sasp_forall` or `--prev_forall` we would obtain incorrect answers. This is because constraints among multiple variables are not taken into account. These forall algorithms only see `X` and `Y` as unbound variables.

Using `c_forall` and/or `--all_c_forall` yields us the expected result.

```
% QUERY:?- not q.
no models
```

Example 6. Here is a more complex example involving constraints and variables with non trivial combinations.

```
1 p(X, Y, Z):- all_pos(X, Y), all_pos(Y, Z), all_pos(X, Z).
2 all_pos(A, B):- A #> B.
3 all_pos(A, B):- A #=< B.
4
5 q:- not p(X, Y, Z).
```

The query (`?- not q.`) succeeds. Using the default `c_forall` only one model is produced.

```
% QUERY:?- not q.

        ANSWER: 1 (in 5.417 ms)

MODEL:
{ not q,  p(Var0,Var1,Var2),  all_pos(Var0,Var1),  all_pos(Var1,Var2),
all_pos(Var0,Var2),  p(Var3,Var4,Var5),  all_pos(Var3,Var4),
all_pos(Var4,Var5),  all_pos(Var3,Var5),  p(Var6,Var7,Var8),
all_pos(Var6,Var7),  all_pos(Var7,Var8),  all_pos(Var6,Var8),
p(Var9,Var10,Var11),  all_pos(Var9,Var10),  all_pos(Var10,Var11),
all_pos(Var9,Var11),  p(Var12,Var13,Var14),  all_pos(Var12,Var13),
all_pos(Var13,Var14),  all_pos(Var12,Var14) }
BINDINGS: ? ;
```

Using `all_c_forall` gives 24 models differing only in the order in which the clauses have been evaluated. If we invoke `scasp --all_c_forall --tree --long` to obtain the justification trees, we can compare two of them.

```
not q :-
  not o_q_1 :-
    forall(...) :-
     not o_q_1(Var0,Var1,Var2) :-
         p(Var0,Var1,Var2) :-
            all_pos(Var0,Var1) :-
              Var0 #> Var1.
            all_pos(Var1,Var2) :-
              Var1 #> Var2.
            all_pos(Var0,Var2) :-
              Var0 #> Var2.
     not o_q_1(Var3,Var4,Var5) :-
         p(Var3,Var4,Var5) :-
            all_pos(Var3,Var4) :-
              Var3 #> Var4.
            all_pos(Var4,Var5) :-
              Var4 #=< Var5.
            all_pos(Var3,Var5) :-
              Var3 #> Var5.
     not o_q_1(Var6,Var7,Var8) :-
         p(Var6,Var7,Var8) :-
            all_pos(Var6,Var7) :-
              Var6 #> Var7.
            all_pos(Var7,Var8) :-
              Var7 #=< Var8.
            all_pos(Var6,Var8) :-
              Var6 #=< Var8.
     not o_q_1(Var9,Var10,Var11):-
         p(Var9,Var10,Var11) :-
            all_pos(Var9,Var10) :-
```

```
not q :-
  not o_q_1 :-
    forall(...) :-
     not o_q_1(Var0,Var1,Var2) :-
         p(Var0,Var1,Var2) :-
            all_pos(Var0,Var1) :-
              Var0 #> Var1.
            all_pos(Var1,Var2) :-
              Var1 #> Var2.
            all_pos(Var0,Var2) :-
              Var0 #> Var2.
     not o_q_1(Var3,Var4,Var5) :-
         p(Var3,Var4,Var5) :-
            all_pos(Var3,Var4) :-
              Var3 #> Var4.
            all_pos(Var4,Var5) :-
              Var4 #=< Var5.
            all_pos(Var3,Var5) :-
              Var3 #> Var5.
     not o_q_1(Var6,Var7,Var8) :-
         p(Var6,Var7,Var8) :-
            all_pos(Var6,Var7) :-
              Var6 #> Var7.
            all_pos(Var7,Var8) :-
              Var7 #=< Var8.
            all_pos(Var6,Var8) :-
              Var6 #=< Var8.
     not o_q_1(Var9,Var10,Var11) :-
         p(Var9,Var10,Var11) :-
            all_pos(Var9,Var10) :-
```

```
31              Var9 #=< Var10.
32            all_pos(Var10,Var11) :-
33              Var10 #> Var11.
34            all_pos(Var9,Var11) :-
35              Var9 #> Var11.
36        not o_q_1(Var12,Var13,Var14) :-
37            p(Var12,Var13,Var14) :-
38              all_pos(Var12,Var13) :-
39                Var12 #=< Var13.
40              all_pos(Var13,Var14) :-
41                Var13 #> Var14.
42              all_pos(Var12,Var14) :-
43                Var12 #=< Var14.
```

```
31              Var9 #=< Var10.
32            all_pos(Var10,Var11) :-
33              Var10 #> Var11.
34            all_pos(Var9,Var11) :-
35              Var9 #=< Var11.
36        not o_q_1(Var12,Var13,Var14) :-
37            p(Var12,Var13,Var14) :-
38              all_pos(Var12,Var13) :-
39                Var12 #=< Var13.
40              all_pos(Var13,Var14) :-
41                Var13 #> Var14.
42              all_pos(Var12,Var14) :-
43                Var12 #> Var14.
```

Chapter 5

Dynamic Consistency Checking (DCC)

5.1 Introduction

The performance of original s(CASP) implementations is not on par with other ASP systems: model consistency is checked once models have been generated, in keeping with the generate-and-test paradigm.

A denial (named global constraint in ASP literature) such as `:-p, q`, expresses that the conjunction of atoms `p` $\wedge$ `q` cannot be true: either `p`, `q`, or both, have to be false in any stable model. In predicate ASP the atoms have variables and a denial such as `:-p(X), q(X,Y)` means that:

$$false \leftarrow \exists x,y \; (\; p(x) \wedge q(x,y) \;)$$

i.e., `p(X)` and `q(X,Y)` can not be simultaneously true for any possible values of `X` and `Y` in any stable model. To ensure that the tentative partial model is consistent with this denial, the compiler generates a rule of the form

$$\forall x,y \; (chk_i \leftrightarrow \neg(\; p(x) \wedge q(x,y) \;) \;)$$

and to ensure that each sub-check (chk_i) is satisfied, they are included in the rule $nmr\_check \leftarrow chk_1 \wedge \cdots \wedge chk_k \wedge \ldots$, which is transparently called after the program query by the s(CASP) interpreter.

However, this generate-and-test strategy has a high impact on the performance of programs that create many tentative models and use denials to discard those that do not satisfy the constraints of the problem.

This limitation can be overcame using a variation of the top-down evaluation strategy, termed *Dynamic Consistency Checking*, which interleaves model generation and consistency checking. This makes it possible to determine when a literal is not compatible

with the *denials* associated to the global constraints in the program, prune the current execution branch, and choose a different alternative. This strategy is specially (but not exclusively) relevant in problems with a high combinatorial component.

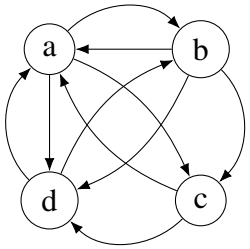
5.2 Example

Consider the Hamiltonian path problem, in which for a given graph we search for a cyclic path that visits each node of the graph only once.

```
1  reachable(V) :- chosen(a, V).
2  reachable(V) :- chosen(U, V), reachable(U).
3  chosen(U, V) :- edge(U, V), not other(U, V).     % Choose or not an
4  other(U, V) :- edge(U, V), not chosen(U, V).     % edge of the graph.
5
6  :- vertex(U), not reachable(U).      % Every vertex must be reachable.
7  :- chosen(U, W), U \= V, chosen(V, W).     % Do not choose edges to or
8  :- chosen(W, U), U \= V, chosen(W, V).     % from the same vertex.
9  #show chosen/2.
10
11 ?- reachable(a).                      % Is there a path from a to a?
```

The conditions of the problem are captured (i) in line 6 to discard tentative paths that do not visit all the nodes, and (ii) in lines 7-8 to discard paths that have edges violating the properties of the Hamiltonian path.

```
1  % Graph
2  vertex(a).
3  vertex(b).
4  vertex(c).
5  vertex(d).
6  edge(b, a).
7  edge(b, d).
8  edge(a, c).
9  edge(a, b).
10 edge(b, c).
11 edge(c, d).
12 edge(d, a).
13 edge(c, a).
14 edge(a, d).
15 edge(d, b).
```



For the query in line 11 of the program, using the graph above, there are three stable models – one for each Hamiltonian cycle.

```
{ chosen(a,c), chosen(c,d), chosen(d,b), chosen(b,a),... }
{ chosen(a,b), chosen(b,c), chosen(c,d), chosen(d,a),... }
{ chosen(a,d), chosen(d,b), chosen(b,c), chosen(c,a),... }
```

As mentioned before, the standard s(CASP) execution follows a generate-and-test scheme, choosing a cycle that reaches node `a` from node `a` and then discards any cycle in which:

- Not all vertices are reached (line 6), e.g., `{chosen(a,b),chosen(b,a)}`.
- Two chosen edges reach / leave the same vertex (line 7), e.g., `{chosen(a,b), chosen(d,b),chosen(b,a)}` or `{chosen(a,b),chosen(b,d),chosen(b, a)}`.

As a consequence, if the evaluation chooses an edge that breaks any of these conditions, trying combinations with the rest of the edges would be misused effort.

Evaluation under DCC

The concept of DCC is simple. For each denial, s(CASP) will generate DCC rules. Each rule is associated with a goal in the denial, and depends on all other goals.

```
1 p :- q.
2 q.
3 r.
4
5 :- p,r.
6
7 dcc(p) :- r.
8 dcc(r) :- p.
```

During execution, whenever a ground goal would be added to the model, s(CASP) checks the associated goals. So, if we query `p`, we will check `q`, which succeeds, and therefore `p` succeeds. Before adding `p` to the model, s(CASP) ensures `dcc(p)` fails. In this case, that does not happen, meaning there is no model.

When using DCC mode, the original NMR check is still added to the query as normal. If we were to query, `q` there is no DCC rule associated with it, but the NMR check still ensures `p` and `r` are not both true.

It is important to emphasize that during execution, DCC rules are only used if the proven goal is ground. In the case it is not, the NMR check will still ensure the denial is enforced, but since the performance improvement is based on early detection of violations of the denial, there will be no such improvement.

Now, consider the previous hamiltonian path example. By invoking `scasp --dcc hamiltonian.pl graph_4.pl`, s(CASP) evaluates the query `?-reachable(a)` following a goal-directed strategy.

1. The query unifies with the clause in line 1 but the goal `chosen(a,a)` fails because `edge(a,a)` does not exist.
2. From the clause in line 2, `chosen(b,a)` is added to the tentative model, because no DCC rule succeeds. The goal `reachable(b)` is then called.
3. The goal `reachable(b)` unifies with the clause in line 1 and `chosen(a,b)` is added, because it is consistent with `chosen(b,a)`.
4. As the query succeeds for the model `{chosen(b,a),chosen(a,b), reachable(a),reachable(b),...}`, s(CASP) invokes `nmr_check`.
5. `nmr_check` executes checks for all the denials. The following code corresponding to line 6.

```
1 chk1 :- forall(U, not chk1_1(U)).
2 not chk1_1(U) :- not vertex(U).
3 not chk1_1(U) :- vertex(U), reachable(U).
```

All vertices (`vertex(U)`) must be reachable (`reachable(U)`). For vertices `U = a` and `U = b`, `reachable(a)` and `reachable(b)` are already in the model, so there is nothing to check. But for vertex `U = c`, `reachable(c)` is not in the model and therefore `reachable(c)` has to be invoked while checking the denials.

6. From the clause in line 1, `chosen(a,c)` is selected to be added to the model, but it is discarded by DCC, because of the DCC rule `dcc(chosen(V,W),[chosen(U,W),U \=V])`, corresponding to the denial in line 7. Note that this DCC rule is instantiated to `dcc(chosen(a,c),[c \=b, chosen(a,b)]` and the literal `chosen(a,b)` is already in the model.
7. The evaluation backtracks and continues the search using another edge.

The denial in line 6 makes the interpreter to select edges to reach all vertices. The interleaving of the dynamic consistency checking prunes the search, which improves performance up to 90×.

Chapter 6

Analysing Programs

6.1 Verbosity

Verbosity in s(CASP) is used to trace the calls during program execution. It reveals the intrinsic executing mechanism of the s(CASP) to users, and provides information that can be used for debugging. There are several verbosity options in s(CASP): `-v`, `-v0`, `-v1`, `-v2`, `-v3`, `-v4`, `-v5`, `-v6`, `-w`.

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

In `-v` **mode**, s(CASP) prints the predicate whenever it is called, until a model is found. If we query `not p` then we will get the following trace.

```
Trace:              Trace Continued:
(0) not p           (1) not o_chk_1
(1) not o_p_1       (2) not o__chk_1_1
(2) not q           (3) not q
(3) not o_q_1       (1) not o_chk_2
(4) p               (2) not o__chk_2_1
(4) not p           (3) p
(4) not s           (3) not p
(0) o_nmr_check     (3) not s
```

The numbers before the predicates (in parenthesis) indicate the depth of the predicate in the proof tree, i.e., the number of times it calls inside a call.

In `-v0` **mode**: s(CASP) does the same thing as `-v`, but additionally, it prints the current proof tree at each subgoal.

In `-v1` **mode**: it traces failures. Whenever it fails to find a proof for a given subgoal,

s(CASP) will explain how the query fails. Querying `p` will provide the following failure trace.

```
FAILURE to prove the literal:   q

FAILURE to prove the literal:   p
```

The call to `p` depends on `q`, which in turn depends on `s`. There are no rules for `s` so the call to `q` fails. Since `q` fails, `p` also fails. These are logged as they happen. Notice, however, that `s` does not show up. Goals that have no corresponding rule are considered a failure of the rule that contains them. So, they do not show up as failures.

In `-v2` **mode**: the behaviour is similar to the relation between `-v` and `-v0`, `-v2` is the tree version of `v1`.

The call tracing and failure tracing can work simultaneously. In this case, s(CASP) would still display the calling details. Once the program comes to a failure it would also trace how the failure happens. The `-v` and `-v0` options, however, also note when a branch fails. Querying `p` with only the `-v` option provides the following trace.

```
(0) p
(1) q
(2) s
 FAIL
 FAIL
```

The two FAILs indicate the failure to prove `q` and `p`.

In `-v3` **mode**: it will log when denials are detected via DCC. See chapter 5 for more information about DCC.

In `-v4` **mode**: it will print *denials* that fail. Denials refer to the OLON rules in the program.

```
1 p.
2 q.
3 :- p,q.
```

In this program both `p` and `q` are true, but we have a headless rule that states both `p` and `q` cannot both be true. So, all queries will fail, since there is no model. With the `-v4` flag, the denial (the headless rule in this case) that fails will be printed.

```
FAILURE to prove the denial 1.      :- p,q.
```

In `-v5` **mode**: the predicates `!display/1`, `!nl/0`, and `!format/2` will work as expected the respective predicates without `!`.

In `-v6` **mode**: the predicate `!pause/1` will pause the execution and allow output tree/-model/bindings.

Without the flags `-v5`, and `-v6` these predicates are silent (but they appear in the long version of the justification tree).

In `-w` mode: it will give warnings when positive cycles fail due to *variant loops* or when s(CASP) fails due to the disunification of two unbound variables. In chapter 2, we discuss the limitations of positive cycle detection, and in chapter 1, we mention that unbound variables cannot be disunified due to practical reasons. In both cases s(CASP) will fail (which is technically incorrect, but allows execution to continue), i.e., a variant loop is when a positive cycle is detected due to goals having the same structure, such as two unbound variables, but are not provably the same. Since this may or may not be a real positive cycle, s(CASP) fails. This may lead to some correct solutions being ignored. Here is a simple example.

```
1 p(X) :- p(Y).
```

If we query `p(X)`, the query will fail, and (with the -w flag) a warning will be displayed.

```
1 WARNING: Failing in a positive loop due to an exact match.
2         Current call:   p(_9273)
3         Previous call:  p(_9273)
```

The `_9273` is an internal variable name used by s(CASP). If we were to add a fact for `p(1)`. The query would then succeed with `X=1`, but if the fact came after rule 1, then we would still get the warning.

6.2 Justification Tree

s(CASP) provides top-down evaluation trees to generate minimal justifications where it is possible to control which literals should appear. The following is an example of a justification tree.

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

This tree can be generated by s(CASP) in response to the query `p` for the following program.

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

The justification trees are generated by s(CASP) when given the `--tree` option. The current implementation also supports three levels of detail: `--short`, `--mid`, `--long`. Which can be specified as a separate flag. An additional option (`--neg`/`--pos`) determines if the NAF negated literals should be included. By default, `--mid` and `--neg` are assumed when the `--tree` flag is specified.

`--short` shows the literals, which could be negated literals, specified by the `#show` directives.

`--mid` shows all user-defined predicates. This includes classically negated (starting with a '-'), but not the NAF negations.

`--neg` extends `--short` and `--mid` trees to include the NAF negations of included literals.

`--pos` disables `--neg`.

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

By default, when only the `--tree` flag is specified, the justification tree uses `--mid` and `--neg`. The usage of the justification tree would be as below.

```
scasp --tree [--long/--mid/--short][--neg/--pos]implement_file
```

6.2.1 Partial Models, Constraints, and Justifications

Let's consider the following program adapted from (García et al. 2013).

```
1 opera(D) :- not home(D).
2 home(D) :- not opera(D).
3 home(monday).
```

Bob will either go to the opera or stay home on any given day `D`. On Monday, Bob will stay at home. 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`.

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

Constrained and unbound variables can appear in the top-down derivation steps. In this example we used the `--long` flag when generating the tree.

```
% LONG LEVEL 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.
```

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)` holds. Being a negative literal, it needs to be resolved using the dual rules for `home/1`, as specified in chapter 1.1).

```
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.
```

So, `not home(A)` succeeds because the two goals in its body succeed.

1. `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`.
2. `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}`.

We can also provide s(CASP) the `--mid` (the default) or `--short` options to simplify the justification tree. Continuing the example, the `--mid` level tree removed all auxiliary predicates (e.g. dual predicates, `forall`, etc.), and the `--short` level tree further removed other user-defined predicates, leaving literals specified by `#show` directive only. We have added a show directive before generating the trees.

```
#show opera/1.
```

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

```
% SHORT LEVEL EVALUATION TREE:
opera(A | {A \= monday}) :-
    chs(opera(A |
{A \= monday})).
global_constraint.
```

6.2.2 Justifications of Global Constraints

Let us modify our example code 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 constraints against `A` 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. We can look at the long tree of the modified program.

```
% LONG 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)).
```

The first part of the justification tree is similar to the previous long tree, with the addition that variable `A` is constrained not to be equal to `tuesday`. To follow the justification of the global constraints, let us remember that `:-baby(D),opera(D).` is $\forall x \cdot \neg(baby(x) \wedge opera(x))$. The compiler introduced the predicate `forall/2` (See chapter 4). The `o_chk_1/1` predicate is the subcheck generated due to the headless rule. It is part of the NMR check that is append to the query in order to enforce the global constraint. More information can be found in chapter 2. 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}`.

Thus, the constraint holds for days that are not Tuesdays, and for Tuesdays. Therefore, the constraints holds for all days.

6.3 Readable Output

Apart from predicate and symbol based evaluation, s(CASP) also provides human language translation for the input and output rules. This readable output translation applies not only to the code explanation, but to the models and justification trees as well. The human language translation makes the code, tree, and answers more readable for

non-technical readers.

There are two options related to the human-readable translation: `--plain` and `--human`. The default option, `plain`, keeps the original format as literals, and `human` translates these symbolic literals to English.

6.3.1 Predefined Natural Language Patterns

Here is the s(CASP) pseudo-natural language justification for the program and query from the opera program example, using only predefined patterns. This can be generated by using the `--tree` and `--human` options when running the program. Let's continue using the previous *opera* example.

```
'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.
```

Each line corresponds to a literal in the justification tree shown before and is generated as follows.

`name(Arg1, ..., Argn):` *'name' holds (for arg_1,..., 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:` *there is no evidence that*, followed by the pattern for `name/n`.

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

The `--human` flag can also be used with the `--code` flag to get a human readable version of dual program generated by s(CASP). Let's use the originali (unmodified) opera example.

```
% USER PREDICATES:
'opera' holds (for Var0), if
     there is no evidence that 'home' holds (for Var0).

'home' holds (for Var0), if
     there is no evidence that 'opera' holds (for Var0).
'home' holds (for monday).

'global_constraints' holds, if
     The global constraints hold.


% DUAL RULES:
```

```
there is no evidence that 'opera' holds (for Var0), if
     'rule 1' holds (for Var0).

'rule 1' holds (for Var0), if
     'home' holds (for Var0).

there is no evidence that 'home' holds (for Var0), if
     'rule 1' holds (for Var0) and
     'rule 2' holds (for Var0).

'rule 1' holds (for Var0), if
     'opera' holds (for Var0).

'rule 2' holds (for Var0), if
     Var0 is not equal monday.


% INTEGRITY CONSTRAINTS:
The global constraints hold.
```

6.3.2 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.

Let us consider the two following examples.

```
#pred opera(D) :: 'Bob goes to the opera on @(D:day)'.
#pred home(D) :: 'Bob stays home on @(D)'
#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.

```
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.
```

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.

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*

Messages can be defined for different instantiation patterns of the same literal.

```
#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)'.
```

Let's close this section by using the `--code` flag with the `--human` on the original example, but with our user-defined patterns.

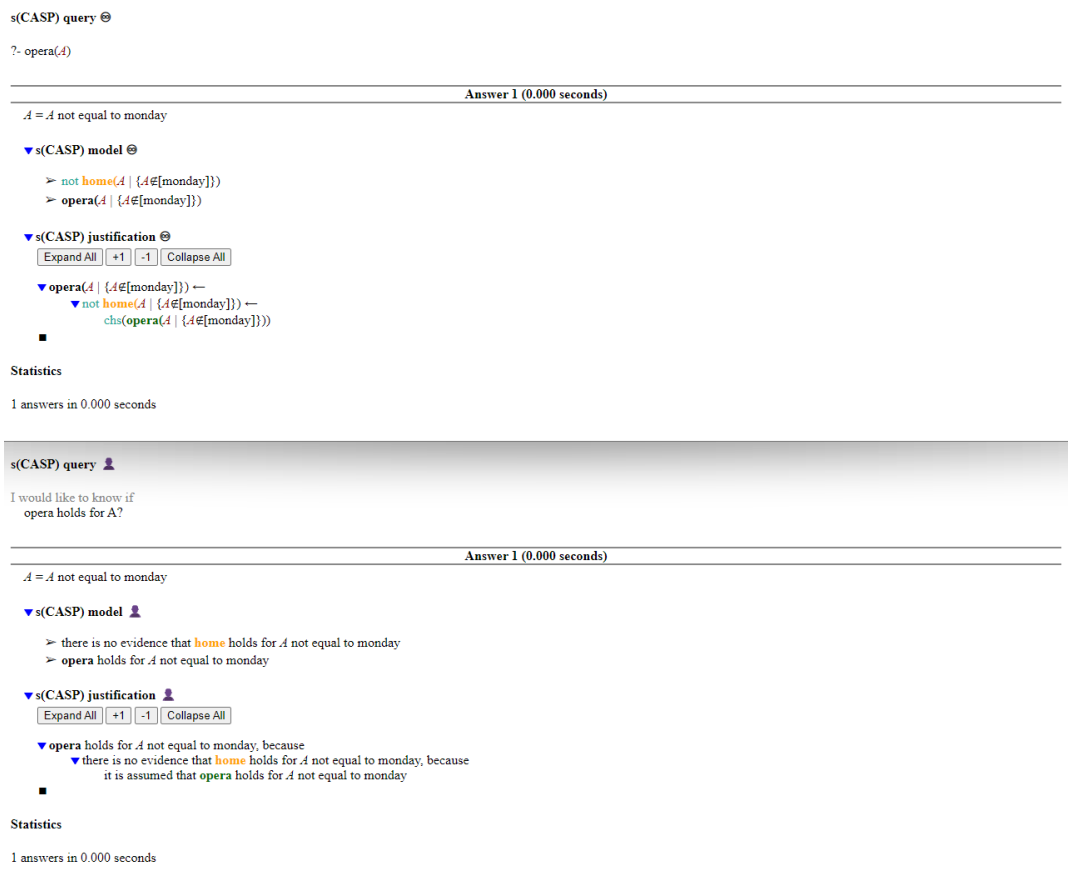
```
% USER PREDICATES:
Bob goes to the opera on Var0, a day,, if
     Bob does not stay at home on Var0.

Bob stays home on Var0, if
     there is no evidence that Bob goes to the opera on Var0, a day,.
Bob stays home on monday.

'global_constraints' holds, if
     The global constraints hold.


% DUAL RULES:
there is no evidence that Bob goes to the opera on Var0, a day,, if
     'rule 1' holds (for Var0).

'rule 1' holds (for Var0), if
     Bob stays home on Var0.

Bob does not stay at home on Var0, if
     'rule 1' holds (for Var0) and
     'rule 2' holds (for Var0).

'rule 1' holds (for Var0), if
     Bob goes to the opera on Var0, a day,.

'rule 2' holds (for Var0), if
     Var0 is not equal monday.


% INTEGRITY CONSTRAINTS:
The global constraints hold.
```

6.4 HTML Output

The `--html` option allows users to save the s(CASP) output as an HTML file. Usually the `--html` option comes along with the justification tree. The usage of the `--html` option is as below.

```
scasp [--tree][--human]--html output_file_name exec_file_name
```

The output file name could be `-`, referring to the default output file name.



The top half of the image above shows the output HTML file of the opera example, and was generated by executing following command.

```
scasp --tree --long --html output.html opera.pl,
```

The bottom half adds the `--human` option, making the rules human readable.

The literal and human-readable conversion can also be done by clicking the rightmost icon at the title of each module, but the `--plain` or `--human` option sets the default displaying mode. The HTML page also supports expanding or collapsing lines. The

triangle in front of each lines allows users to show or collapse this line. Another way to expand or collapse is to click the buttons, where `+1` means to expand one more line while `-1` refers to collapse the last line.

Chapter 7

Knowledge Representation and Reasoning

While we tend to think in terms of algorithms and data structures when programming, s(CASP) is designed with viewing a problem in terms of knowledge representation and reasoning in mind. Unlike Prolog, s(CASP) sacrifices many non-logical operations, such as input and output, in an effort to maintain its semantics, and make it conductive of such uses. This chapter will cover a few ways of approaching problems from this point of view.

7.1 Uninterpreted Function Symbols

7.1.1 Introduction

One useful way s(CASP) differs from ASP is in the possibility of using arbitrary uninterpreted function symbols to build, for example, data structures. While in mainstream ASP implementations these could give rise to an infinite grounded program (i.e., if the program does not have the *bound-term-depth* property), the s(CASP) execution model can deal with them similarly to Prolog, with the added power of the use of constructive negation in the execution and in the returned models.

7.1.2 Example

The predicate `member/2` below, from (Arias et al. 2021:Pag. 11), models the membership to a list as it is usual done in (classical) logic programming. The query is intended to derive the conditions for which `B` is not a member of the list.

```
1 member(X, [X|Xs]).
2 member(X, [_|Xs]):- member(X, Xs).
3 list([1,2,3,4,5]).
4 ?- list(A), not member(B, A).
```

This program and query returns the following model and bindings.

```
{ list([1,2,3,4,5]),
   not member(B| {B \= 1,B \= 2,B \= 3,B \= 4,B \= 5}, [1,2,3,4,5]),
   not member(B| {B \= 1,B \= 2,B \= 3,B \= 4,B \= 5}, [2,3,4,5]),
   not member(B| {B \= 1,B \= 2,B \= 3,B \= 4,B \= 5}, [3,4,5]),
   not member(B| {B \= 1,B \= 2,B \= 3,B \= 4,B \= 5}, [4,5]),
   not member(B| {B \= 1,B \= 2,B \= 3,B \= 4,B \= 5}, [5]),
   not member(B| {B \= 1,B \= 2,B \= 3,B \= 4,B \= 5}, []) }
   A = [1,2,3,4,5], B \= 1, B \= 2, B \= 3, B \= 4, B \= 5
```

In other words, for variable `B` to not be a member of the list `[1,2,3,4,5]` it has to be different from each of its elements.

7.2 Classical Negation

s(CASP) allows for two forms of negation. Negation-as-failure (or default negation) as discussed in chapter 1, and *classical negation* (sometimes called *strong negation*) which needs to be proven. In s(CASP) classical negation is represented by prepending a minus (-) sign.

```
1 -human(X) :- reptile(X).
```

This program says that a reptile is definitely not a human. When classical negation is used a global constraint is added by the system to ensure that a term cannot be both true and false.

```
1 flies(X) :- bird(X).
2 -flies(X) :- penguin(X).
3 bird(X) :- penguin(X).
4 penguin(sam).
```

This program has no model. Since `sam` is a penguin and a penguin is a bird, `sam` flies. However rule 2 tells us that penguins don't fly. So, `sam` both flies and doesn't fly: a contradiction. This constraint is enforced by the NMR check as if the headless rule `:- flies,-flies` was added to the program. Classical negation is used when we want a proof that something is false, and negation-as-failure is used when we simply want there to be no evidence that it is true.

```
1 student_admitted(S) :- -failed_entry_exam(S), not refused_offer(S).
```

A student will be admitted if the entry exam is definitely not failed and there is no evidence that the offer was refused. We can extend the program with the following

knowledge.

```
-failed_entry_exam(sam).
```

Now, we know that Sam did not fail the entry exam, but we do not know if Sam refused the offer. If we query `student_admitted(sam)`, we discover that Sam is admitted. Since we do not know if Sam refused the offer, NAF negation assumes it is false. Therefore, `not refused_offer(S)` is true.

This example illustrates that NAF negation is weaker. As long as you do not have evidence that it is true, then it is false. Classical negation is stronger requiring explicit evidence that it is false.

As mentioned, however, classical negation is treated as a separate predicate with an automatically generated constraint. As seen in the penguin example this can lead the program having no model, rather than overriding what we know to be true. This is a detail that must be taken into account when working with classical negation in s(CASP). In addition, since the constraint is enforced via the NMR check, it will be appended to the query – increasing the amount of code executed.

Still, having both classical negation and negation-as-failure allows for better knowledge representation. This should be more clear in the coming sections.

7.3 Default Logic

Stable models, and by extension s(CASP), is great for modeling default logic. Let's consider the typical example of flight being a property of birds.

```
flies(X) :- bird(X).
```

According to the above program all birds fly, but in reality we know this is not true. In section 7.2 classical negation is used to denote that penguins do not fly. This led to a contradiction. In fact there are several flightless birds and even other reasons (such as having a wounded wing) a bird might not be able to fly. For such birds unable to fly, we can call them abnormal birds.

```
flies(X) :- bird(X), not abnormal_bird(X).
abnormal_bird(X) :- penguin(X).
abnormal_bird(X) :- wounded_wing(X).
bird(X) :- penguin(X).
```

By default if we know that something is a bird, we assume it flies. If there is evidence that that bird is a penguin or has a wounded wing, we say it does not fly. That is, being an abnormal bird is an exception to the default rule that birds fly. For instance, suppose there is a bird, Tweety.

```
bird(tweety).
```

The query `flies(tweety)` will succeed. The goal `bird(X)` will succeed due to the fact. The goal `not abnormal_bird(tweety)` will call the dual rule for `abnormal_bird/1`. The dual rules will in turn call `not penguin(tweety)`, which is true because there is evidence Tweety is a penguin, and `not wound_wing(tweety)`, which is also true because there is no evidence `wounded_wing(tweety)` is true.

Now, if we add the knowledge that Tweety is injured, the result changes.

```
1 wounded_bird(tweety).
```

Now, `not wounded_wing(tweety)` will fail, since `wounded_wing(tweety)` is true, and `not abnormal_bird(tweety)` will be false, since `abnormal_bird(tweety)` is true. Thus, the goal `flies(tweety)` will fail. The default assumption is that Tweety flies, because he is a bird. However, Tweety has a wounded wing, and therefore is an exception to this assumption.

We can also strengthen an exception by using classical negation.

```
1 flies(X) :- bird(X), not abnormal_bird(X), not -flies(X).
2 abnormal_bird(X) :- penguin(X).
3 abnormal_bird(X) :- wounded_wing(X).
4 bird(X) :- penguin(X).
```

If there is direct evidence that something does not fly, then it does not fly. Unlike the example in section 7.2 this program is not inconsistent. As an example, suppose we have a bird, Polly, and we know that Polly cannot fly, but do not know why.

```
1 bird(polly).
2 -flies(polly).
```

The query `not flies(polly)` will succeed. The dual rules will check `not bird(polly)`, which will fail; `abnormal_bird(polly)`, which will also fail; and then `-flies(polly)`, which will succeed. After `not flies(polly)` succeeds, the NMR check will be called. Recall (from chapter 7.2), classical negation causes s(CASP) to generate a constraint as if there is a headless rule `:- flies(X), -flies(X)`. Since `not flies(polly)` succeeds, flies(polly) must be false. The dual rules are defined to have this property.

7.4 Preferences

Preferences can be considered an extension of default logic. Instead of a single layer of defaults, we can have multiple layers.

```
1 manager_eats :- manager_hungry, manager_has_no_meeting.
2 senor_employee_eats :- senor_employee_not_busy,
3                        senor_employee_needs_lunch,
4                        not manager_eats.
```

```
5 senor_employee_needs_lunch :- senor_employee_hungry.
6 senor_employee_needs_lunch :- senor_employee_busy_later.
7 intern_eats :- not manager_eats, not senor_employee_eats.
```

There are three levels of preferences. The lowest level being `intern_eats`. The intern gets to have lunch only if the manager and senor employee is not taking off lunch. We are assuming that only one person can take time off for lunch at any given time. The senor employee will not take lunch off if the manager does, but also has other conditions. In particular, if the employee is hungry or knows there will be no time later. Finally, the highest level is `manager_eats`. The manager will eat if the manager is hungry and there is no meeting. We can decide who eats by creating a simple predicate for that.

```
1 assign_lunch_time :- manager_eats.
2 assign_lunch_time :- senor_employee_eats.
3 assign_lunch_time :- intern_eats.
```

There will be a single answer when we query `assign_lunch_time` depending on the facts provided. As an example consider the following.

```
1 manager_hungry.
2 senor_employee_busy_later.
```

The answer will include `senor_employee_eats`. The call to `manager_eats` will fail. There is no fact for `manager_has_no_meeting`. Then, `senor_employee_eats` will succeed since the employee will be busy later and the manager will not eat lunch. The intern will not eat because the senor employee is. If we remove the fact for `senor_employee_busy_later` then `senor_employee_eats` will fail and the `intern_eats` will succeed.

We can make make this more explicit by using the concepts of preference levels. We will use predicates to represent each level of preference, separating the preference logic from the knowledge.

```
1 intern_eats :- not level1.
2 senior_employee_eats :- senor_employee_needs_lunch, not level2.
3 manager_eats :- manager_hungry, manager_has_no_meeting.
4
5 level1 :- level2.
6 level1 :- senior_employee_eats.
7 level2 :- manager_eats.
```

Logically, this program is the same as the previous one. If we query `assign_lunch_time`, `manager_eats` will fail because `manager_has_no_meeting` is not true. Then, `senior_employee_eats` will be true. The goal `senior_employee_needs_lunch` will be true because the employee will be busy later. Finally, `not level2` will succeed, since `level2` fails only when `manager_eats` is false. This is guaranteed by the dual rule.

One thing to note about this pattern is that `level2` is used to trigger `level1`. This is what establishes the preferences. Without this line it is possible for the manager and the intern to eat at the same time. The intern is blocked only by `level1`.

The benefit of this organization, besides keeping the rules clean, is that we can add predicates to a preference level by adding the corresponding rule. As a simple example, suppose we want to add in a junior employee.

```
1 junior_employee_eats :- junior_employee_hungry, not level2.
2 level1 :- junior_employee_eats.
```

The junior employee will eat when the employee is hungry, as long as level 2 has not been triggered. In this case, that would be when the manager eats. We also placed `junior_employee` in level 1, which would stop the intern from eating when the junior employee eats, but not stop the senior employee.

7.5 Degrees of Truth

One "shortcoming" of logic systems like s(CASP) is its binary nature. Something is either true or it is false, but as humans we often consider things in degrees of truth. There may be something we believe is true, but are not 100% certain. One method would be to use preferences – where each level represents a different degree of truth. However, with negation as failure and classical negation we can encode this uncertainty into our rules using five degrees of truth. Absolute truth can be represented by a call to a predicate.

```
1 have_lunch :- hungry.
```

Absolute falsity can be represented by a call to the classical negation of a predicate.

```
1 have_lunch :- -full.
```

Complete uncertainty (50/50) can be represented by requiring there be no proofs for either.

```
1 have_lunch :- not hungry,not -hungry.
```

Here there is no evidence of being hungry or not being hungry, so have lunch just in case. If we want to know if something might be true, we just need to show that there is no proof (or certainty) that is it false.

```
1 have_lunch :- not -hungry.
```

As long as we don't know if we are hungry, we will eat lunch. Similarly, if we want to know if something might be false we just need to show that there is not proof that it is true.

```
1 have_lunch :- not full.
```

As long as we don't know for certain we are full, we will eat lunch. This can also be used when using interactive mode. Suppose we want to know about our hunger. We can query `hungry`. If it succeeds then we know it is true, but if it fails we know that `not hungry` will succeed. That is, `hungry` might be false. We can then query `-hungry`. If it succeeds we know we are not hungry. If it fails, then `not -hungry` will succeed and we know we have complete uncertainty.

Let's put this together in a simple example. Here is what could be an excerpt from an AI for a game of poker.

```
raise :- great_hand.
call :- good_hand.
fold :- bad_hand.
```

In this program, the AI makes a decision based on the quality of its hand. We can add a bit more nuance to it by considering the opponent.

```
raise :- great_hand.
raise :- opponent_bluff.
call :- good_hand, not -opponent_bluff.
fold :- bad_hand.
fold :- good_hand, -opponent_bluff.
```

In this version, the AI takes into account it's belief about whether or not the opponent is bluffing. We can see that the AI calls if it has a good hand, and the opponent *might* be bluffing. The AI will fold with a good hand if the opponent is definitely bluffing.

7.6 Other Knowledge Patterns

Default logic, preferences, and degrees of truth could be described as a form of knowledge pattern. When we encounter knowledge that fits the pattern we know to use the corresponding technique. Other patterns exist. In (Chen et al. 2016), several patterns are identified. When writing a s(CASP) program for your domain consider what knowledge patterns may exist.

7.7 Constraints on Logic

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(CASP) is to make use of "local" constraints. Local constraints involve killing the world as it is generated. Local constraints make rules more meaningful, and, since s(CASP) is a goal-directed language, local constraints may decrease the amount of backtracking

needed. That being said, there is a role for global constraints in s(CASP). In this section we will take a look at both global and local constraints on programs.

7.7.1 Global Constraints

Global constraints can be specified through an odd loop or a headless rule as discussed in Section 2.4. 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, kill all such colorings. For each odd loop and headless rule s(CASP) 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(CASP) 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.
```

The NMR check may be additional overhead, increasing backtracking. Some cases

may minimize overhead by making use of dcc as presented in chapter 5. Another problem with the graph color example, is that `color/2` is relatively meaningless by itself. It is part of an even loop that generates every possible coloring of the graph. In this case a local constraint may make the rule more meaningful.

There is a place for global constraints, however. If the constraint cannot be viewed as part of the definition of the predicate, then a global constraint may be better.

```
1 latch_open :- press_button.
2 button_working :- no_safety_concerns.
3 :- latch_open,not button_working.
```

In this example, we are enforcing a safety constraint. The logic for `latch_open` and `button_working` are defined independently. In this case we do not want one to depend on the other. The global constraint, implemented as a headless rule, ensures there is no world in which the latch is open when the button has been deactivated for safety concerns.

7.7.2 Local Constraints

Local constraints implement the constraints on a predicate as a goal in the body. 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 that 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.

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

The constraint rule 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 specialized when our 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:

```
1       color(N,C) :- node(N), color(C), not other_color(N,C),
2                     not same_as_neighbors(N,C).
3 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.

```
1 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).
```

7.7.3 Implementation Concerns

As noted earlier, the preferred way to approach logic constraints in s(CASP) is through local constraints, except where local constraints do not fit (such as in the latch example). We believe this, in the general case, will reduce backtracking, and make rules in the program more meaningful and easier to read. Due to implementation limitations, however, there may be times where this introduces a new cost. In particular, the graph coloring program using local constraints will still have an NMR check. Since `color/2` depends on it's negation, and odd loop is detected and a subcheck for the rules will be generated. There is no actual odd loop, because the bindings for the two goals will be different, but s(CASP) has no way of detecting that during preprocessing.

7.8 Nondeterminism in s(CASP)

There are several ways of representing multiple answers. Prolog uses the nondeterministic nature of rule execution and the generation of partial models. This method views individual answers as part of a single model for the program. ASP on the other hand uses multiple worlds. Each answer is represented by a different model. Since s(CASP) is a hybrid between the two pardigms, both methods of representing answers are available. This leads to the question "which method should be used?".

As an example, let's see a simple program written both ways.

```
% Prolog Way
teach(smith) :- available(smith).
teach(jones) :- available(jones).
available(smith).
available(jones).
```

In this program both Smith and Jones will teach if they are available. By querying `teach(X)`, we get two answers. We can use this to mean that either one can teach.

```
% ASP way
teach(smith) :- available(smith), not teach(jones).
teach(jones) :- available(jones), not teach(smith).
available(smith).
available(jones).
```

In this program, Smith teaching and Jones teaching are mutually exclusive. They form an even cycle. When we query `teach(X)`, we will once again get two answers. The difference, this time is that it is never the case that both can teach. One worl Smith cna teach and in the other Jones can teach. In the Prolog method it is possible that both can teach, since there is only a single model. As an example, consider adding the following to both programs.

```
full_load(smith).
:- full_load(X), teach(X).
```

Now, when we query `teach(X)` we will get different results. The ASP way will give a single answer. The world in which Smith teaches is killed due to the headless rule. In the Prolog way, the same thing happens, but since there is only a single model that model is killed. So, there will be no answers for the Prolog way program.

There is no clear answer as to which method is the best method, and may be more a matter of preference or which best fits the way knowledge you are encoding is acquired. When getting started with s(CASP) it may be better to stick to the style you are most familiar with. If you are coming from Prolog, use the prolog way. If you are coming from ASP use the ASP way. Our *official advice* is to stick to the Prolog way of representing multiple answer unless you have a clear reason to use multiple worlds. No matter what method you choose, you should remain consistent and be cognizant of how you are representing different answers. Otherwise, it is easy to become confused.

One last comment on this topic. If you are using numerical constraints (chapter 3), you will definitely want to stick to the Prolog way whenever possible.

```
p(X) :- X #> 3.
```

The implementation of CLP($\mathbb{Q}$) has been modified to work with constructive negation, but is otherwise the same semantics. In the above program, there is a single model for which `p(X)` is true for all possible rational number bindings for `X` greater than three. Similar to the previous example, if we added a constraint that kills the model, then there can be no model.

```
:- p(2).
```

Since, `p(2)` is true, the entire model is killed.

7.9 Current Limitations

The s(CASP) 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.

7.9.1 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 work around this by not directly calling the negation of a predicate that depends on a large set of facts. Instead, ground the variable.

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

7.9.2 Finite Domains

One area of s(CASP) that might catch a user off guard is the way finite domains are handled. In s(CASP), no domain is actually finite.

```
p(1).
```

The predicate, `p/1`, has an infinite domain, despite have only a single grounded fact. If we were to query, `not p(X)`, it would succeed, adding a constraint `X\=1`. This is the intended behavior of s(CASP), but can conflict with the concept in the programmer's head.

```
student(alice).
student(bob).
student(charlie).
student(dan).
student(erica).

grade(alice,a).
```

```
8  grade(charlie,a).
9
10 needs_no_improvement(X) :- student(X),grade(X,a).
```

If we wish to find the students that need improvements, we would query `not need_no_improvement(X)`. The first result, however, will be `X\=alice,X\=bob, X\=charlie,X\=dan,X\=erica`. This is because, needing no improvement depends on the person being a student. This is logically correct, but probably not what the programmer wanted.

To work around this, you can prepend the query by the domain – `student(X), not needs_no_improvement(X)`. Now, `student(X)` will bind `X` to a student before checking if `X` needs improvement.

7.9.3 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.

```
1 same(X,X)
```

The dual will be equivalent to:

```
1 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.

7.9.4 CLP and the Zeno Time Problem

Consider the following.

```
1 p(X,Y) :- p(Z,X).
```

This program forms an even cycle. So, a query to `p(A,B)`, will fail due to that cycle. However, if we modify the program to use CLP($\mathbb{Q}$) to constraint the values of X and Y we get different behavior.

```
1 p(X,Y) :- X #< Y, p(Z,X).
```

Now, the query `p(A,B)` does not terminate. The constraints cause each call to be structurally different. If s(CASP) did not treat them as structurally different, then other programs may be rendered incorrect.

So, instead the above program is no different to s(CASP) as the following.

```
1 p(X) :- Y is X+1, p(Y).
```

Querying `p(0)` will not terminate, because the parameter of each recursive call will be a different integer. Solving this problem with constraints is an active area of research. Currently, there is no work around, and instead the programmer must be careful that such situations do not arise.

Chapter 8

Using the s(CASP) program

This chapter is a reference to the s(CASP) program and the various commandline flags available.

8.1 Usage

```
Usage:  scasp [options] InputFile(s)
```

Despite the usage line above, taken from the s(CASP) help message, the options do not have to come before the input files. The available options are more detailed below. As shown, s(CASP) can take more than one input file. These input files are treated as if they were concatenated into one file.

This allows the program to be split into multiple files, perhaps each with different source. For instance, we can have a file with the logic, and another with facts to operate on – provided by the user.

8.2 Commandline Options

-h, -?, --help Print the usage statement and the most commonly used flags.

--help_all Print extended help. Includes all options presented in this section.

--version Output the current version of s(CASP) and exit.

-i, --interactive Run the REPL (interactive loop). This allows you to pose queries to the program.

-a, --auto Run the program in batch mode, with no user interaction. This is the default behavior. A query should be provided in the program itself. If there are multiple

queries specified in the file, only the last one is used.

--timeout[=MS] To handle cases where computation takes too long, or the code gets stuck in an infinite loop like with the zeno time problem (see section 7.9.4), s(CASP) will time out. The default is 1000 milliseconds, but this option allows the timeout to be adjusted to the programs needs.

-sN, -nN Produce N answers. N should be positive. As a special case, if N is zero, then all answers are produced.

-d, --plaindual Generating dual rules involve considering all the ways a goal could fail. This means for each goal in the body, we must check to see if its negation can succeed (see section 1.1). Because a later goal may depend on an earlier goal binding a value to a variable, s(CASP) keeps all previous goals when checking such a negations. This option, disables this behavior. This option should be used with caution. Only propositional programs are guaranteed to work correctly when this option is selected.

-r[=d] When CLP($\mathbb{Q}$) is used (see chapter 3), rational numbers are displayed as fractions. This option will cause s(CASP) to display them as real numbers (decimal numbers). The parameter *d*, determines the precision of the displayed number. If *d* is not given then `d = 5` is used.

--code After preprocessing the program to include dual rules and the NMR check, print the modified program, and then exit.

--tree When an answer is produced, print the justification tree. See section 6.2.

--plain Affects `--tree` and `--code` options. When specified, these options use the plain s(CASP) terms/rules. This is the default behavior for `--tree` and `--code`.

--human Affects `--tree` and `--code` options. When specified, these options will be produce human readable output. See section 6.3.

--long This option affects `--tree`, and causes the long version of the justification tree to be printed. This will include internal rally generated predicates used to implement dual rules and NMR checks. See section 6.2.

--mid This option affects `--tree`, and causes the mid-sized version of the justification tree to be printed. Only user defined predicates will be printed. See section 6.2. This is the default behavior.

--short This option affects `--tree`, and causes the short version of the justification tree to be printed. Only predicates explicitly specified by the `#show` directive will be printed. See section 6.2.

--neg This option affects the `--tree` option. Includes negative goals, when printing the justification tree. See section 6.2. This is the default behavior.

--pos This option affects the `--tree` option. Negative goals are not included when printing the justification tree. See section 6.2.

--html[=name] Generate an html file that includes the model and a collapsable justification tree. When *name* is given, then "name.html" will be used as the file name to store the html. If *name* is not given then the name of the first inut file given s(CASP) will be used as *name*. See section 6.4.

--quiet Do not print model and justification tree. Useful when combined with `--html`.

-v, --verbose While executing a query, trace user-predicate calls. See section 6.1.

-v0 While executing a query, trace user-predicate calls. Each call is accompanied by the current justification tree. See section 6.1.

-v1 Trace user-predicate failures. When a goal cannot be proven, print a message. See section 6.1.

-v2 Trace user-predicate failures. When a goal cannot be proven, print a message. Each failed goal is accompanied by the current justification tree. See section 6.1.

-v3 Trace dcc detections. See chapter 5 about dcc, and section 6.1 about this flag.

-v4 Trace denials failures. When an NMR subcheck fails, print out the rule that generated that subcheck. See section 6.1.

-w, --warning Enable warning messages for failures in variant loops and disequality. See section 2.2 for variant loops, section 7.9.3 for disequality, and section 6.1 for this flag.

--dcc Activate the Dynamic Consistency Check. Executes NMR subchecks only if an involved predicate is used. See chapter 5.

--co Activate co-induction (include warnings). Normally when a positive cycle is detected, the current execution fails and backtracks (see section 2.2). This option changes this behavior, instead allowing the execution to succeed like with even cycles. Whenever execution succeeds in this way, s(CASP) will print a warning to inform the user.

--all_c_forall The current forall algorithm in s(CASP) can lead to certain solutions to be discarded. This option causes scasp to use a version of the algorithm that does not discard solutions, but may lead to a very large number of duplicate solutions. See chapter 4.

--prev_forall Use the previous forall algorithm. This algorithm does not discard solutions, and does not give duplicate solutions. It does not work with numerical constraints (CLP($\mathbb{Q}$)). See chapter 4.

--sasp_forall Use the original forall algorithm used by the original s(ASP) system. This algorithm is not as efficient as the other algorithms, but is simpler. See chapter 4.

--no_olon Do not compile olon rules. For debugging purposes. See section 2.4.

--no_nmr Do not compile NMR rules. For debugging purposes. See section 2.4.

--variant Normally when a positive cycle is detected, the current execution fails and backtracks (see section 2.2). This option changes this behavior, instead allowing the execution to continue. Instead, failing only when a goal is exactly encountered again. This may lead to non-termination.

--machine Generates models or contra-models without human outputs. With this option, s(CASP) will simply print yes/no for a query.

--update Automatically update s(CASP).

-c, –compiled Does not do preprocessing. This option assumes you previously generated the code with the --code option. This allows the programmer to manually make modifications to the preprocessed code. In general this option should be avoided.

Bibliography

Arias, J., Carro, M., Chen, Z., and Gupta, G. (2021). **Modeling and Reasoning in Event Calculus using Goal-Directed Constraint Answer Set Programming**. In: *CoRR* abs/2106.14566. arXiv: 2106.14566. URL: https://arxiv.org/abs/2106.14566.

Arias, J., Carro, M., Chen, Z., and Gupta, G. (2022). **Modeling and Reasoning in Event Calculus using Goal-Directed Constraint Answer Set Programming**. In: *Theory and Practice of Logic Programming* 22(1), pp. 51–80. DOI: 10.1017/S1471068421000156.

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

Chen, Z., Marple, K., Salazar, E., Gupta, G., and Tamil, L. (2016). **A Physician Advisory System for Chronic Heart Failure Management Based on Knowledge Patterns**. In: *Theory and Practice of Logic Programming* 16(5-6), pp. 604–618. DOI: 10.1017/S1471068416000429.

García, A. J., Chesñevar, C. I., Rotstein, N. D., and Simari, G. R. (2013). **Formalizing Dialectical Explanation Support for Argument-Based Reasoning in Knowledge-Based Systems**. In: *Expert Systems and Applications* 40(8), pp. 3233–3247. DOI: 10.1016/j.eswa.2012.12.036.

Gelfond, M. and Kahl, Y. (2014). **Knowledge Representation, Reasoning, and the Design of Intelligent Agents: The Answer-Set Programming Approach**. Cambridge University Press.

Gupta, G., Salazar, E., Varanasi, S., Basu, K., Arias, J., Arias, J., Shakerin, F., Min, R., Fang, L., and Wang, H. (2022). **Automating Commonsense Reasoning with ASP and s(CASP)**. In: *ICLP'22 Workshops: 2nd Workshop on Goal-directed Execution of Answer Set Programs (GDE'22)*. https://utdallas.edu/~gupta/csr-scasp.pdf. CEUR-WS.org, pp. 1–16.

Gupta, G., Bansal, A., Min, R., Simon, L., and Mallya, A. (2007). **Coinductive Logic Programming and its Applications**. In: *23rd Int'l. Conference on Logic Programming*. Springer, pp. 27–44.

Holzbaur, C. (1995). **OFAI CLP(Q,R) Manual, Edition 1.3.3**. Tech. rep. TR-95-09. Vienna: Austrian Research Institute for Artificial Intelligence.

Marple, K., Bansal, A., Min, R., and Gupta, G. (2012). **Goal-Directed Execution of Answer Set Programs**. In: *Principles and Practice of Declarative Programming,*

PPDP'12, Leuven, Belgium - September 19 - 21, 2012. Ed. by D. D. Schreye, G. Janssens, and A. King. ACM, pp. 35–44. DOI: 10.1145/2370776.2370782.

Marple, K. and Gupta, G. (2012). **Galliwasp: A goal-directed answer set solver**. In: *International Symposium on Logic-Based Program Synthesis and Transformation*. Springer, pp. 122–136.

Marple, K., Salazar, E., Chen, Z., and Gupta, G. (2017a). **The s(ASP) Predicate Answer Set Programming System**. In: *The Association for Logic Programming Newsletter*.

Marple, K., Salazar, E., and Gupta, G. (2017b). **Computing Stable Models of Normal Logic Programs Without Grounding**. In: *CoRR* abs/1709.00501.

Simon, L., Mallya, A., Bansal, A., and Gupta, G. (2006). **Coinductive Logic Programming**. In: *ICLP'06*. Vol. 4079. LNCS. Springer, pp. 330–345.