Constraint Answer Set Programming without Grounding ∗

JOAQUIN ARIAS, MANUEL CARRO
IMDEA Software Institute and *Universidad Politécnica de Madrid*
(*e-mail:* joaquin.arias@{imdea.org,alumnos.upm.es}, manuel.carro@{imdea.org,upm.es})

ELMER SALAZAR, KYLE MARPLE, and GOPAL GUPTA
University of Texas at Dallas
(*e-mail:* {ees101020,kmarple1,gupta}@utdallas.edu)

submitted 1 January 2003; revised 1 January 2003; accepted 1 January 2003

Abstract

Extending ASP with constraints (CASP) enhances its expressiveness and performance. This extension is not straightforward as the grounding phase, present in most ASP systems, removes variables and the links among them, and also causes a combinatorial explosion in the size of the program. Several methods to overcome this issue have been devised: restricting the constraint domains (e.g., discrete instead of dense), or the type (or number) of models that can be returned. In this paper we propose to incorporate constraints into s(ASP), a goal-directed, top-down execution model which implements ASP while retaining logical variables both during execution and in the answer sets. The resulting model, s(CASP), can constrain variables that, as in CLP, are kept during the execution and in the answer sets. s(CASP) inherits and generalizes the execution model of s(ASP) and is parametric w.r.t. the constraint solver. We describe this novel execution model and show through several examples the enhanced expressiveness of s(CASP) w.r.t. ASP, CLP, and other CASP systems. We also report improved performance w.r.t. other very mature, highly optimized ASP systems in some benchmarks.

1 Introduction

Answer Set Programming (ASP) has emerged as a successful paradigm for developing intelligent applications. It uses the stable model semantics (Gelfond and Lifschitz 1988) for programs with negation. ASP has attracted much attention due to its expressiveness, ability to incorporate non-monotonicity, represent knowledge, and model combinatorial problems. On the other hand, constraints have been used both to enhance expressiveness and to increase performance in logic programming. Therefore, it is natural to incorporate constraints in ASP systems. This is however not straightforward as ASP systems usually carry out an initial grounding phase where variables (and, therefore, the constraints linking them) disappear. Several approaches have been devised to work around this issue. However, since constraints need to be grounded as well, these approaches limit the range of admissible constraint domains (e.g., discrete instead of dense), the places where constraints can appear, and the type (or number) of models that can be returned. The integration of constraints with ASP is not as seamless as in standard constraint logic programming (CLP).

∗ Work partially supported by EIT Digital (https://eitdigital.eu), MINECO project TIN2015-67522-C3-1-R (TRACES), Comunidad de Madrid project S2013/ICE-2731 *N-Greens Software*, NSF IIS 1718945, and NSF IIS 1423419.

In this work we propose to restore this integration by incorporating constraints into the s(ASP) (Marple et al. 2017b) execution model. s(ASP) is a goal-directed, top-down, SLD resolution-like procedure which evaluates programs under the ASP semantics without a grounding phase either before or during execution. s(ASP) supports predicates and thus retains logical variables both during execution and in the answer sets. We have extended s(ASP)'s execution model to make its integration with generic constraint solvers possible. The resulting execution model and system, called s(CASP), makes it possible to express constraints on variables and extends s(ASP)'s in the same way that CLP extends Prolog's execution model. Thus, s(CASP) inherits and generalizes the execution model of s(ASP) while remaining parametric w.r.t. the constraint solver. Due to its basis in s(ASP), s(CASP) avoids grounding the program and the concomitant combinatorial explosion. s(CASP) can also handle answer set programs that manipulate arbitrary data structures as well as reals, rationals, etc. We show, through several examples, its enhanced expressiveness w.r.t. ASP, CLP, and other ASP systems featuring constraints. We briefly discuss s(CASP)'s efficiency: on some benchmarks it can outperform mature, highly optimized ASP systems.

Several approaches have been proposed to mitigate the impact of the grounding phase in ASP systems. In the case of large data sets, *magic set* techniques have been used to improve grounding for specific queries (Alviano et al. 2012). For programs which use uninterpreted function symbols, techniques such as *external sources* (Calimeri et al. 2007) have been proposed.

Despite these approaches, grounding is still an issue when constraints are used in ASP. Variable domains induced by constraints can be unbound and, therefore, infinite (e.g., `X #> 0` with $X \in \mathbb{N}$ or $X \in \mathbb{Q}$). Even if they are bound, they can contain an infinite number of elements (e.g. `X #> 0` $\land$ `X#<1` in $\mathbb{Q}$ or $\mathbb{R}$). These problems have been attacked using different techniques:

- Translation-based methods (Balduccini and Lierler 2017), which convert both ASP and constraints into a theory that is executed in an SMT solver-like manner. Once the input program is translated, they benefit from the features and performance of the target ASP and CLP solvers. However, the translation may result in a large propositional representation or weak propagation strength.
- Extensions of ASP systems with constraint propagators (Banbara et al. 2017; Janhunen et al. 2017) that generate and propagate new constraints during the search and thus continuously check for consistency using external solvers using e.g. conflict-driven clause learning. However, they are restricted to finite domain solvers (hence, dense domains cannot be appropriately captured) and incrementally generate ground models, lifting the upper bounds for some parameters. This, besides being a performance bottleneck, falls short of capturing the true nature of variables in constraint programming.

Due to the requirement to ground the program, causing a loss of communication from elimination of variables, the execution methods for CASP systems are complex. Explicit hooks sometimes are needed in the language, e.g., the `required` builtin of EZCSP (Balduccini and Lierler 2017), so that the ASP solver and the constraint solver can communicate. Additionally, considerable research has been conducted on devising top-down execution models for ASP (Dal Palù et al. 2009; Baselice et al. 2009; Baselice and Bonatti 2010) that could be extended with constraints.

We have validated the s(CASP) approach with an implementation in Ciao Prolog which integrates Holzbaur's CLP($\mathbb{Q}$) (Holzbaur 1995), a linear constraint solver over the rationals.[1] The

[1] Note that while we used CLP($\mathbb{Q}$) in this paper, CLP($\mathbb{R}$) could also have been used.

s(CASP) system has been used to solve a series of problems that would cause infinite recursion in other top-down systems, but which in s(CASP) finitely finish, as well as others that require constraints over dense and/or unbound domains. Thus, s(CASP) is able to solve problems that cannot be straightforwardly solved in other systems.

2 Background: ASP and s(ASP)

ASP (Gelfond and Lifschitz 1988; Brewka et al. 2011) is a logic programming and modelling language. An ASP program Π is a finite set of *rules*. Each rule $r \in \Pi$ is of the form:

$$a \leftarrow b_1 \wedge \ldots \wedge b_m \wedge not\ b_{m+1} \wedge \ldots \wedge not\ b_n.$$

where a and $b_1, \ldots, b_n$ are atoms and *not* corresponds to *default* negation. An atom is an expression of form $p(t_1, \ldots, t_n)$ where p is a predicate symbol of arity n and t_i, are *terms*. An atom is *ground* if no variables occur in it. The set of all *constants* appearing in Π is denoted by C_Π. The head of rule r is $h(r) = \{a\}$[2] and the body consists of positive atoms $b^+(r) = \{b_1, \ldots, b_m\}$ and negative atoms $b^-(r) = \{b_{m+1}, \ldots, b_n\}$. Intuitively, rule r is a justification to *derive* that a is true if all atoms in $b^+(r)$ have a derivation and no atom in $b^-(r)$ has a derivation. An interpretation I is a subset of the program's Herbrand base and it is said to satisfy a rule r if $h(r)$ can be derived from I. A model of a set of rules is an interpretation that satisfies each rule in the set. An answer set of a program Π is a minimal model (in the set-theoretic sense) of the program

$$\Pi^I = \{h(r) \leftarrow b^+(r) \mid r \in \Pi, b^-(r) \cap I = \emptyset\}$$

which is called the *Gelfond-Lifschitz reduct* of Π with respect to I (Gelfond and Lifschitz 1991). The set of all answer sets of Π is denoted by $AS(\Pi)$. ASP solvers which compute the answer sets of non-ground programs use the above semantics by first applying, to each rule $r \in \Pi$, all possible substitution from the variables in r to elements of C_Π (this procedure is called *grounding*). To make this grounding possible, the rules of the program should be *safe*, i.e., all variables that appear in a rule have to appear in some positive literal in the body. The rule is termed *unsafe* otherwise.

A difference between ASP and Prolog-style (i.e., SLD resolution-based) languages is the treatment of negated literals. Negated literals in a body are treated in ASP using their logical semantics based on computing stable models. The *negation as failure* rule of Prolog (i.e., SLDNF resolution (Clark 1978)) makes a negated call succeed (respectively, fail) iff the non-negated call fails (respectively, succeeds). To ensure soundness, SLDNF has to be restricted to ground calls, as a successful negated goal cannot return bindings. However, SLDNF increases the cases of non-termination w.r.t. SLD.

s(ASP) (Marple et al. 2017a, 2017b) is a top-down, goal-driven interpreter of ASP programs written in Prolog (`http://sasp-system.sourceforge.net`). The top-down evaluation makes the *grounding* phase unnecessary. The execution of an s(ASP) program starts with a *query*, and each answer is the resulting *mgu* of a successful derivation, its justification, and a (partial) stable

[2] Disjunctive ASP programs (i.e., programs with disjunctions in the heads of rules) can be transformed into non-disjunctive ASP programs by using *default* negation (Ji et al. 2016).

model. This partial stable model is a subset of the ASP stable model (Gelfond and Lifschitz 1988) including only the literals necessary to support the query with its output bindings.[3]

Example 1 (Assuming an extended Herbrand Base). Given the program below:

```
married(john).                    :- not married(X).
```

most ASP systems are not able to compute its stable model (not even an empty one), because the global constraint is unsafe. On the other hand, s(ASP) is able to compute queries to programs with unsafe rules by assuming that the unsafe variables take values in an *extended Herbrand Universe*, and not just that of the terms which can be constructed from the symbols in the program. Therefore, using this alternative semantics `:- not married(X).` corresponds to $\neg\exists x.\neg married(x) \equiv \forall x.married(x)$ and since the program only has evidence of one married individual (`john`), there is no stable model (i.e., it cannot be derived that all possible individuals are married). However, if we add the (unsafe) fact `married(X)` (i.e., $\forall x.married(x)$) to the program, the resulting stable model will be {`married(X)`} — every element of the universe is married.

s(ASP) has two additional relevant differences w.r.t. Prolog: first, s(ASP) resolves negated atoms *not* l_i against *dual rules* of the program (Section 2.1), instead of using negation as failure. This makes it possible for a non-ground negated call `not p(X)` to return the results for which the positive call `p(X)` would fail. Second, and very important, the dual program is **not** interpreted under SLD semantics: a number of very relevant changes related to how loops are treated (see later) are introduced.

2.1 Dual of a Logic Program

The dual of a predicate `p/1` is another predicate that returns the `X` such that `p(X)` is not true. It is used to give a constructive answer to a goal `not p(X)`. The dual of a logic program is another logic program containing the dual of each predicate in the program (Alferes et al. 2004). To synthesize the dual of a logic program P we first obtain Clark's completion (Clark 1978), which assumes that the rules of the program completely capture all possible ways for atomic formulas to be true, and then we apply De Morgan's laws:

1. For each literal p/n that appears in the head of a rule, choose a tuple $\vec{x}$ of n distinct, new variables $x_1, \ldots, x_n$.
2. For each i-th rule of a predicate p/n of the form $p_i(\vec{t_i}) \leftarrow B_i$, with $i = 1,\ldots,k$, make a list $\vec{y_i}$ of all variables that occur in the body B_i but do not occur in the head $p_i(\vec{t_i})$, add $\exists\vec{y_i}$ to the body and rename the variables that appear in the head $\vec{t_i}$ with the tuple $\vec{x}$, obtained in the previous step, resulting in a predicate representing $\forall\vec{x}\ (p_i(\vec{x}) \leftarrow \exists\vec{y_i}\ B_i)$. Note that $\vec{x}$ are local, fresh variables. This step captures the standard semantics of Horn clauses.
3. With all these rules and using Clark's completion, we form the sentences:

$$\begin{array}{lll} \forall\vec{x}\ (\ p(\vec{x}) & \longleftrightarrow & p_1(\vec{x}) \vee \ldots \vee p_k(\vec{x})\ ) \\ \forall\vec{x}\ (\ p_i(\vec{x}) & \longleftrightarrow & \exists\vec{y_i}\ (b_{i.1} \wedge \ldots \wedge b_{i.m} \wedge \neg\, b_{i.m+1} \wedge \ldots \wedge \neg\, b_{i.n})\ ) \end{array}$$

[3] Note that the subset property holds only when the Gelfond–Lifschitz transformation is applied assuming an *extended Herbrand Base* obtained by extending the set of constants in the program, C_Π, with an infinite number of new elements.

4. Their semantically equivalent duals $\neg p/n$, $\neg p_i/n$ are:
$$\begin{array}{lcl} \forall \vec{x}\ (\ \neg p(\vec{x}) & \longleftrightarrow & \neg(p_1(\vec{x}) \vee \ldots \vee p_k(\vec{x}))\ ) \\ \forall \vec{x}\ (\ \neg p_i(\vec{x}) & \longleftrightarrow & \neg\ \exists \vec{y}_i\ (b_{i.1} \wedge \ldots \wedge b_{i.m} \wedge \neg\ b_{i.m+1} \wedge \ldots \wedge \neg\ b_{i.n})\ ) \end{array}$$
5. Applying De Morgan's laws we obtain:
$$\begin{array}{lcl} \forall \vec{x}(\ \neg p(\vec{x}) & \longleftrightarrow & \neg p_1(\vec{x}) \wedge \ldots \wedge \neg p_k(\vec{x})\ ) \\ \forall \vec{x}\ (\neg p_i(\vec{x}) & \longleftrightarrow & \forall \vec{y}_i\ (\neg b_{i.1} \vee \ldots \vee \neg\ b_{i.m} \vee\ b_{i.m+1} \vee \ldots \vee\ b_{i.n})\ ) \end{array}$$
which generates a definition for $\neg p(\vec{x})$ and a separate clause with head $\neg p_i(\vec{x})$ for each positive or negative literal $b_{i.j}$ in the disjunction. Additionally, a construction to implement the universal quantifier introduced in the body of the dual program is necessary (Section 2.3).

Definitions for the initially negated literals $\neg b_{i.m+1} \ldots \neg b_{i.n}$ and for each of the *new* negated literals $\neg b_{i.1} \ldots \neg b_{i.m}$ are similarly synthesized. At the end of the chain, unification has to be negated to obtain disequality, e.g., $x = y$ is transformed into $x \neq y$ (Section 2.2).

Example 2. Given the program below:

```
1  p(0).
2  p(X) :- q(X), not t(X,Y).
3  q(1).
4  t(1,2).
```

the resulting dual program is:

```
1  not p(X) :- not p1(X), not p2(X).
2  not p1(X) :- X \= 0.
3  not p2(X) :-
4      forall(Y, not p2_(X,Y)).
5  not p2_(X,Y) :- not q(X).
6  not p2_(X,Y) :- q(X), t(X,Y).
7  not q(X) :- not q1(X).
8  not q1(X) :- X \= 1.
9  not t(X,Y) :- not t1(X,Y).
10 not t1(X,Y) :- X \= 1.
11 not t1(X,Y) :- X = 1, Y \= 2.
```

For efficiency, the generation of the dual diverges slightly from the previous scheme. The dual of a body $B \equiv l_1 \wedge \ldots$ is the disjunction of its negated literals $\neg B \equiv \neg l_1 \vee \ldots$, which generates independent clauses in the dual program. To avoid redundant answers, every clause for a negated literal $\neg l_i$ includes calls to any positive literal l_j with $j < i$. E.g., clause 6 from the previous program, `not p2(X,Y):- q(X),t(X,Y)`, would only need to be `not p2(X,Y):- t(X,Y)`. However, the literal `q(X)` is included to avoid exploring solutions already provided by clause 5, `not p2(X,Y):- not q(X)`. The same happens with clauses 10 and 11.

2.2 *Constructive Disequality*

Unlike Prolog's *negation as failure*, disequality in s(ASP), denoted by "\=" , represents the constructive negation of the unification and is used to construct answers from negative literals. Intuitively, `X\=a` means that `X` can be any term not unifiable with `a`. In the implementation reported in (Marple et al. 2017b) a variable can only be disequality-constrained against ground terms, and the disequality of two compound terms may require backtracking to check all the cases: `p(1,Y) \= p(X,2)` first succeeds with `X\=1` and then, upon backtracking, with `Y\=2`.

The former restriction reduces the range of valid programs, but this does not seem to be a problem in practice: since positive literals are called before negative literals in the dual program, the number of cases where this situation may occur is further reduced. Since this is orthogonal to the implementation framework, it can be improved upon separately. The second characteristic impacts performance, but can again be ameliorated with a more involved implementation of disequality which carries a disjunction of terms.

Algorithm 1: *forall*

```
forall receives V, a variable name, and Goal, a callable goal.
V starts unbound
Execute Goal.
if Goal succeeded then                                      Let us check the bindings of V
    if V is unbound then forall succeeds                    Goal's success is independent of V
    else if V is bound, then backtrack to step 4 and try other clauses
    else                                  V has been constrained to be different from a series of values
        Re-execute Goal, successively substituting the variable V with each of these values
        if Goal succeeds for each value then forall succeeds
        else forall fails                   There is at least one value for which Goal is not true
    end
else forall fails                         There are infinitely many values for which Goal is not true
```

2.3 `Forall` Algorithm

In (Marple et al. 2017b) the universal quantifier is evaluated by `forall(V,Goal)` which checks if `Goal` is true for all the possible values of `V`. When `forall/2` succeeds, the evaluation continues with the quantified variable unbound. Multiple quantified variables are handled by nesting: $\forall v_1, v_2.Goal$ is executed as `forall(V1,forall(V2,Goal))`. The underlying idea is to verify that for any solution with `V \= a` (for some `a`), `Goal` also succeeds with `V=a` (Algorithm 1).

Example 3. Consider the following program with the dual rule for `p/0`:

```
p :- not q(X).                 not p :- forall(X, not p1(X)).
q(X) :- X = a.                 not p1(X) :- q(X).
q(X) :- X \= a.
```

Under the query `?- not p`, the interpreter will execute `forall(X,not p1(X))` with `X` unbound. First, `not p1(X)` is executed and calls `q(X)`, succeeding with `X=a`. Then, since `X` is bound, the interpreter backtracks and succeeds with `X \= a` (second clause of `q/1`). Now, since `X` is constrained to be different from `a`, the interpreter re-executes `not p1(X)` with `X=a` which succeeds (first clause of `q/1`). Since there are no more constrained values to be checked, the evaluation of the query finishes with success. Note that leaving `X` unbound after the success of `forall(X,p(X))` is consistent with the interpretation that the answer set {`p(X)`} corresponds to $\forall x.p(x)$.

2.4 Non-Monotonic Checking Rules

Non-monotonic rules are used by s(ASP) to ensure that partial stable models are consistent with the global constraints of the program. Given a consistency rule of the form $\forall\vec{x}(p_i(\vec{x}) \leftarrow \exists\vec{y}\ B_i \wedge \neg p_i(\vec{x}))$, and in order to avoid contradictory rules of the form $p_i(\vec{a}) \leftarrow \neg p_i(\vec{a})$, all stable models must satisfy that at least one literal in B_i is false (i.e., $\neg B_i$) or, for the values $\vec{a}$ where B_i is true, $p_i(\vec{a})$ can be derived using another rule. To ensure that the partial stable model is consistent, the s(ASP) compiler generates, for each consistency rule, a rule of the form:

$$\forall\vec{x}(\ chk_i(\vec{x})\ \longleftrightarrow\ \ \forall\vec{y}_i(\neg B_i \vee p(\vec{x})\ )\ )$$

To ensure that each sub-check (chk_i) is satisfied, the compiler introduces into the program the rule $nmr\_check \leftarrow chk_1 \wedge \ldots \wedge chk_k$, which is transparently called after the program query.

	s(CASP)	s(ASP)
hanoi(8,T)	**1,528**	13,297
queens(4,Q)	**1,930**	20,141
One hamicycle	**493**	3,499
Two hamicycle	**3,605**	18,026

Table 1: Speed comparison: s(CASP) vs. s(ASP) (time in ms).

Example 4. Given the program below:

```
:- not s(1, X).
p(X):- q(X), not p(X).
```

the resulting *NMR* check rules are:

```
nmr_check :-
    chk1,
    forall(A, chk2(A)).
chk1 :- forall(X,s(1,X)).
chk2(X) :- not q(X).
chk2(X) :- q(X), p(X).
```

Infinite Loops Finally, in order to break infinite loops, s(ASP) uses three techniques to deal with *odd loops over negation*, *even loops over negation*, and *positive loops* (Marple et al. 2017b, Gupta et al. 2007). Since they are not essential for this paper, a summary is included in Appendix B, for the reader's convenience.

3 s(CASP): Design and Implementation

S(CASP) (available together with the benchmarks used in this paper at `https://gitlab.software.imdea.org/joaquin.arias/sCASP`) extends s(ASP) by computing partial stable models of programs with constraints. This extension makes the following contributions:

- The interpreter is reimplemented in Ciao Prolog (Hermenegildo et al. 2012). The driving design decision of this reimplementation is to let Prolog take care of all operations that it can handle natively, instead of interpreting them. Therefore, a large part of the environment for the s(CASP) program is carried implicitly in the Prolog environment. Since s(CASP) and Prolog shared many characteristics (e.g., the behavior of variables), this results in flexibility of implementation (see the interpreter code sketched in Figure 1 and in full in Appendix A) and gives a large performance improvement (Table 1). Note that all the experiments in this paper were performed on a MacOS 10.13 machine with an Intel Core i5 at 2GHz.
- A new solver for disequality constraints.
- The definition and implementation of a generic interface to plug-in different constraint solvers. This required, in addition to changes to the interpreter, changes to the compiler which generates the dual program. This interface has been used, in this paper, to connect both the disequality constraint solver and the CLP($\mathbb{Q}$) solver.
- The design and implementation of *C-forall* (Algorithm 2), a generic algorithm which extends the original *forall* algorithm (Algorithm 1) with the ability to evaluate goals with variables constrained under arbitrary constraint domains. In addition to being necessary to deal with constraints, this extension generalizes and clarifies the design of the original one.

```
??(Query) :-
   solve(Query,[],Mid),
   solve_goal(nmr_check,Mid,Out),
   print_just_model(Out).
solve([], In, ['$success'|In]).
solve([Goal|Gs], In, Out) :-
   solve_goal(Goal, In, Mid),
   solve(Gs, Mid, Out).
solve_goal(Goal, In, Out) :-
   user_defined(Goal), !,
   pr_rule(Goal, Body),
   solve(Body, [Goal|In], Out).
solve_goal(Goal, In, Out) :-
   call(Goal),
   Out = ['$success',Goal|In].
```

Fig. 1: (Very abridged) Code of the s(CASP) interpreter.

3.1 s(CASP) Programs

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

$$a \leftarrow c_a \wedge b_1 \wedge \ldots \wedge b_m \wedge not\ b_{m+1} \wedge \ldots \wedge not\ b_n.$$

where the difference w.r.t. an ASP program is c_a, a simple constraint or a conjunction of constraints. A query to an s(CASP) program is of the form $\leftarrow\ c_q \wedge l_1 \wedge \ldots \wedge l_n$, where c_q is also a simple constraint or a conjunction of constraints. The semantics of s(CASP) extends that of s(ASP) following (Jaffar and Maher 1994). During the evaluation of an s(CASP) program, the interpreter generates constraints whose consistency w.r.t. the current constraint store is checked by the *constraint solver*. The existence of variables both during execution and in the final models is intuitively justified by adopting an approach similar to that of the S-semantics (Gabbrielli and Levi 1991).

3.2 The Interpreter and the Disequality Constraint Solver

The s(CASP) interpreter carries the environment (the call path and the model) implicitly and delegates to Prolog all operations that Prolog can do natively, such as handling the bindings due to unification, the unbinding due to backtracking, and the operations with constraints, among others. The clauses of the program, their duals, and the NMR-checks are created by the compiler by generating rules of the predicate `pr_rule(Head,Body)`, where `Head` is an atom and `Body` is the list of literals. While the s(CASP) interpreter performs better than s(ASP), little effort has been invested in optimizing it (see Section 5), and there is ample room for improvement.

Figure 1 shows a highly simplified sketch of the code that implements the interpreter loop in s(CASP), where:

- `??(+Query)` receives a query and prints the successful path derivations.
- `solve(+Goals,+PathIn,-PathOut)` reproduces SLD resolution.
- `solve_goal(+Goal,+PathIn,-PathOut)` evaluates the user-defined predicates and hands over to Prolog the execution of the builtins using `call/1`. The `PathOut` argument encodes the derivation tree in a list.

Every '`$success`' constant denotes the success of the goals in the body of a clause and means that one has to go up one level in the derivation tree. Several '`$success`' constants in a row mean, accordingly, that one has to go up the same number of levels.

In s(CASP), constructive disequality is handled by a disequality constraint solver, called CLP($\neq$), implemented using attributed variables that makes disequality handling transparent to

Algorithm 2: C-forall

```
1  C-forall receives the variable V, the callable goal Goal, and an initial constraint store, $C_i$ ($i = 1$).
2  V starts unbound.                                   The constraint store of V is empty, $C_{V.i} = \top$
3  Execute Goal with $C_i$ as the current constraint store.     Its first answer constraint store is $A_1$
4  if the execution of Goal succeeds then      Check $A_{V.i}$, the domain of V in the answer constraint
5      if $A_{V.i} \equiv C_{V.i}$ then                There was no refinement in the domain of V
6          C-forall succeeds                          V is not relevant for the success of Goal
7      else                                  The domain of V has been restricted, $A_{V.i} \sqsubset C_{V.i}$
8          $C_{i+1} = C_i \wedge A_{\overline{V}.i} \wedge \neg A_{V.i}$       Remove from V the elements for which Goal succeeds
9          Return to step 3 and re-execute Goal under $C_{i+1}$
                                      Check whether Goal is true for the rest of the elements of V
10     end
11 else C-forall fails                    There is a non-empty domain for which Goal is not true
```

the user code. The current implementation of CLP($\neq$) does not address the restrictions described in Section 2.2; however, as mentioned before, since the solver is independent of the interpreter, its improvements are orthogonal to the core implementation of s(CASP).

The interpreter checks the call path before the evaluation of user-defined predicates to prevent inconsistencies and infinite loops (Marple et al. 2017b) (see Appendix B). The call path is a list constructed with the calls, and the bindings of the variables in these calls are automatically updated by Prolog.

- When a positive loops occurs, the interpreter fails only if the looping goal and its ancestor are equal (i.e., `p(X):-...,p(X)`). Termination properties are enhanced if a tabling system featuring variant calls or entailment (Arias and Carro 2016) is used as implementation target, so that all programs with a finite grounding or with the constraint-compact property terminate.
- However, when the current call is equal to an already-proven ancestor, the evaluation succeeds to avoid its re-computation and to reduce the size of the justification tree.

3.3 Integration of Constraint Solvers in s(CASP)

Holzbaur's CLP($\mathbb{Q}$) (Holzbaur 1995) solver was integrated in the current implementation of s(CASP). Since the interpreter already deals with the CLP($\neq$) constraint solver, only two details have to be taken in consideration:

- The compiler is extended to support CLP($\mathbb{Q}$) relations $\{<, >, =, \geq, \leq, \neq\}$ during the construction of the dual program and the NMR rules.
- Since it is not possible to decide at compile time whether equality will be called with CLP($\mathbb{Q}$) or Herbrand variables, its dual \= is extended to decide at run-time whether to call the CLP($\mathbb{Q}$) solver or the disequality solver.

Finally, to make integrating further constraint solvers easier, the operations that the s(CASP) interpreter requires from the CLP($\mathbb{Q}$) solver are encapsulated in a single module that provides the interface between the interpreter and the constraint solver. Additional constraint solvers only need to provide the same interface.

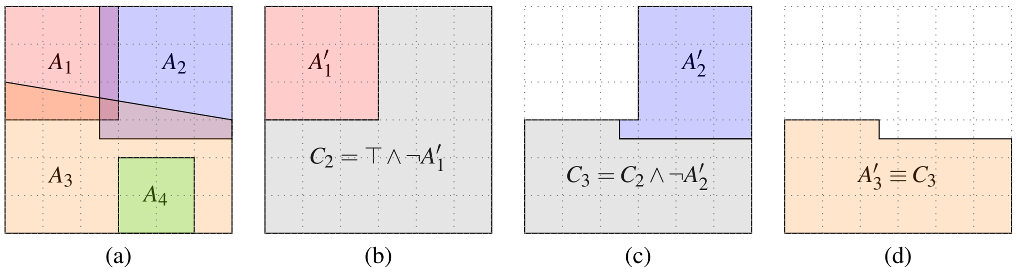

Fig. 2: A *C-forall* evaluation that succeeds.

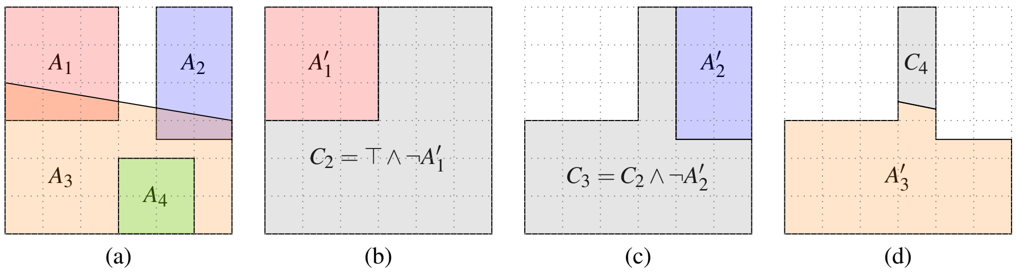

Fig. 3: A *C-forall* evaluation that fails.

3.4 Extending `forall` for Constraints

Extending s(ASP) to programs with constraints requires a generalization of *forall* (Algorithm 1) which we will call *C-forall* (Algorithm 2). A successful evaluation of `Goal` in s(CASP) returns, on backtracking, a (potentially infinite) sequence of models and answer constraint stores $A_1, A_2, \ldots$. Each A_i relates variables and constants by means of constraints and bindings (i.e., syntactical equality constraints). The execution of `forall(V,Goal)` is expected to determine if `Goal` is true for all possible values of `V` in its constraint domain.

In what follows we will use $\overline{\mathtt{V}}$ to denote the set variables in `Goal` that are not `V`: *vars*(`Goal`) = $\{\mathtt{V}\} \cup \overline{\mathtt{V}} \wedge \mathtt{V} \notin \overline{\mathtt{V}}$. The core idea is to iteratively narrow the store C under which `Goal` is executed by selecting **one** answer A and re-executing `Goal` under the constraint store $C \wedge A_{\overline{\mathtt{V}}} \wedge \neg A_{\mathtt{V}}$, where $A_{\mathtt{V}}$ is the projection of A on `V` and $A_{\overline{\mathtt{V}}}$ is the projection of A on $\overline{\mathtt{V}}$. The iterative execution finishes with a positive or negative outcome.

*Example 5 (*C-forall *terminates with success).* Figure 2 shows an example where the answers $A_1, \ldots, A_4$ to `Goal` cover the whole domain, represented by the square. Therefore, *C-forall* should succeed. The answer constraints that the program can generate are depicted on picture (a). For simplicity in the pictures, we will assume that the answers A_i only restrict the domain of `V`, so it will not be necessary to deal with `V` and $\overline{\mathtt{V}}$ separately since $A_{\overline{\mathtt{V}}}$ will always be empty, and therefore $A_{\mathtt{V}.i} = A_i$. Picture (b) shows the result of the first iteration of *C-forall* starting with $C_1 = \top$: 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.

```
forall(V, Goal) :-
    empty_store(Store),             % V has no attached constraints
    eval_forall(V, Goal, [Store]).  % start the evaluation of Goal
eval_forall(_, _, []).              % it's done, forall succeeds
eval_forall(V,Goal,[Store|Sts]):-
    copy(V, Goal, NV, NGoal),       % copy to keep V unbound
    apply(NV, V, Store),            % add the constraint to NV
    once(NGoal),                    % if fails, the forall fails
    dump(NV, V, AnsSt),             % project the answer store
    (   equal(AnsSt, Store)         % if there is no refinement in NV
    ->  true                        % then, it's done, continue
    ;   dual(AnsSt, AnsDs),         % else, the answer's dual/duals
        add(AnsDs, Store, NSt),     % is/are added to Store
        eval_forall(V, Goal, NSt)   % to evaluate Goal
    ),
    eval_forall(V, Goal, Sts).      % continue the evaluation
```

Fig. 4: Code of the predicate `forall/2` implemented in s(CASP).

Termination for an infinite number of answer sets The previous example points to a nice property: even if there were an infinite number of answer sets to `Goal`, as long as a finite subset of them covers the domain of `V` and this subset can be finitely enumerated by the program, the algorithm will finish. This is always true for constraint-compact domains, such as disequality over a finite set of constants or the gap-order constraints (Revesz 1993). Note that this happens as well in the next example, where *C-forall* fails.

Example 6 (C-forall *terminates with failure).* Figure 3 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 `V`. Picture (a) depicts the answer constraints that `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 `Goal` under the store $C_4 = C_3 \wedge \neg A_3'$ fails because the solution A_4 of `Goal` does not have any element in common with C_4, and then *C-forall* also fails.

Figure 4 shows a sketch of the code that implements *C-forall* in the s(CASP) interpreter, written in Prolog/CLP. In this setting, `Goal` carries the constraint stores C_i and the answer stores A_i implicitly in its execution environment. We know that the interpreter will call `forall(V,Goal)` with a fresh, unconstrained `V`, because the executed code is generated by the s(CASP) compiler. Therefore, the projection of C_1 onto `V` is an empty constraint store, which we introduce explicitly to start the computation.

The call `copy(V,Goal,NV,NGoal)` copies `Goal` in `NGoal` sharing only $\overline{\texttt{V}}$, while `V` is substituted in `NGoal` by a fresh variable, `NV`. In the main body of `eval_forall/3`, `Store` always refers to `V`, while `NGoal` does not contain `V`, but `NV`. The call `apply(NV,V,Store)` takes the object `Store` and makes it part of the global store but substituting `V` for `NV` so that the execution of `NGoal` can further constrain `NV` while `V` remains untouched. Note, however, that in the first iteration, `NV` will always remain unconstrained, since the constraint store that `apply(NV,V,Store)` applies to it is empty ($C_{\texttt{V}.1} = \top$). However, in the following iterations, `Store` will contain the successive constraint stores $C_{\texttt{V}.i+1}$.

When `once(NGoal)` succeeds, the constraint store $C_i \wedge A_{\overline{\texttt{V}}.i}$ is implicit in the binding of $\overline{\texttt{V}}$.

Therefore, the execution of `eval_forall(V,Goal,Store)` carries this constraint store implicitly because `Goal` and `NGoal` share $\overline{\text{V}}$. Finally, the predicate `dump(NV,V,AnsSt)` projects the constraint store after the execution of `NGoal` on `NV`, rewrites this projection to substitute `NV` for `V`, and leaves the final result in `AnsSt`, generating $A_{\text{V}.i}$. Note that, in some sense, it is transferring constraints in the opposite direction to what `dump/3` did before. If the call `equal(AnsSt, Store)` succeeds, it means that $A_{\text{V}.i} \equiv C_{\text{V}.i}$ and therefore the `forall` succeeds (for the branch that was being explored, see below).

Otherwise, we have to negate the projection of the answer onto `V`, i.e., construct $\neg A_{\text{V}.i}$. The negation of a conjunction generates a disjunction of constraints and most constraint solvers cannot handle disjunctions natively. Therefore, the predicate `dual(AnsSt,AnsDs)` returns in `AnsDs` a list with the components $\neg A_{\text{V}.i.j}$ of this disjunction, $j = 1, 2, \ldots, \mathit{length}(\texttt{AnsDs})$. Then, `add(AnsDs,Store,NSt)` returns in `NSt` a list of stores, each of which is the conjunction of `Store` with one of the components of the disjunction in `AnsDs`, i.e., a list of $C_{\text{V}.i} \wedge \neg A_{\text{V}.i.j}$, for a fixed i. There may be cases where this conjunction is inconsistent; `add/3` captures them and returns only the components which are consistent. Note that if a conjunction $C_{\text{V}.i} \wedge \neg A_{\text{V}.i.j}$ is inconsistent, it means that $\neg A_{\text{V}.i.j}$ has already been (successfully) checked.

Each of the resulting constraint stores will be re-evaluated by `eval_forall/3`, where `apply/3` will apply them to a new variable `NV`, in order to complete the implicit construction of C_{i+1} before the execution of `once(NGoal)`. `forall/2` finishes with success when there are no pending constraint stores to be processed (line 4).

Example 7 (`C-forall` execution negating a constraint conjunction). Given the program below, consider the evaluation of `forall(A,p(A))`:

```
p(X) :- X #>= 0, X #=< 5.        p(X) :- X #< 3.
p(X) :- X #> 1.                  p(X) :- X #< 1.
```

In the first iteration $C_1 = \top$. The first answer is $A_1 = \{X \geq 0 \wedge X \leq 5\}$, which is more restrictive than C_1, so we compute $\neg A_1 = \{X < 0 \vee X > 5\}$. First, `p/1` is evaluated with $C_{2.a} = \{\top \wedge X < 0\}$ obtaining $A_{2.a} = \{X < 0\}$ using the third clause. Since $A_{2.a} \equiv C_{2.a}$, we are done with $C_{2.a}$. But we also have to evaluate `p/1` with $C_{2.b} = \{\top \wedge X > 5\}$. Using the second clause, $A_{2.b} = \{X > 5\}$ is obtained and since $A_{2.b} \equiv C_{2.b}$, the evaluation succeeds.

4 Examples and Evaluation

The expressiveness of s(CASP) allows the programmer to write programs / queries that cannot be written in [C]ASP without resorting to a complex, unnatural encoding. Additionally, the answers given by s(CASP) are also more expressive than those given by ASP. This arises from several points:

- s(CASP) inherits from s(ASP) the use of unbound variables during the execution and in the answers. This makes it possible to express constraints more compactly and naturally (e.g., ranges of distances can be written using constraints)
- s(CASP) can use structures / functors directly, thereby avoiding the need to encode them unnaturally (e.g., giving numbers to Hanoi movements to represent what in a list is implicit in the sequence of its elements).
- The constraints and the goal-directed evaluation strategy of s(CASP) makes it possible to use direct algorithms and to reduce the search space (e.g., by putting bounds on a path's length).

```
1  valid_stream(P,Data) :-
2       stream(P,Data),
3       not cancelled(P, Data).
4
5  cancelled(P, Data) :-
6       higher_prio(P1, P),
7       stream(P1, Data1),
8       incompt(Data, Data1).
9
10 higher_prio(PHi, PLo) :-
11      PHi #> PLo.
12 incompt(p(X), q(X)).
13 incompt(q(X), p(X)).
14
15 stream(1,p(X)).
16 stream(2,q(a)).
17 stream(2,q(b)).
18 stream(3,p(a)).
```

Fig. 5: Code of the stream reasoner.

4.1 *Stream Data Reasoning*

Let us assume that we deal with data streams, some of whose items may be contradictory (Arias 2016). Moreover, different data sources may have a different degree of trustworthiness which we use to prefer a given data item in case of inconsistency. Let us assume that `p(X)` and `q(X)` are contradictory and we receive `p(X)` from source S_1 and `q(a)` from source S_2. We may decide, depending on how reliable are S_1 and S_2, that: (i) `p(X)` is true because S_1 is more reliable than S_2; (ii) `q(a)` is true since S_2 is more reliable than S_1, and for any `X` different from `a` (i.e., `X \= a`), `p(X)` is also true; (iii) or, if both sources are equally reliable, them we have (at least) two different models: one where `q(a)` is true and another where `p(X)` is true.

Figure 5 shows the code for a stream reasoner using s(CASP). Data items are represented as `stream(Priority,Data)`, where `Priority` tells us the degree of confidence in `Data`; `higher_prio(PHi,PLo)` hides how priorities are encoded in the data (in this case, the higher the priority, the more level of confidence); and `incompt/2` determines which data items are contradictory (in this case, `p(X)` and `q(X)`). Note that `p(X)` (for **all** `X`) has less confidence than `q(a)` and `q(b)`, but `p(a)` is an exception, as it has more confidence than `q(a)` or `q(b)`. Lines 1-8, alone, define the reasoner rules: `valid_stream/2` states that a data stream is valid if it is *not cancelled* by another contradictory data stream with more confidence.

The confidence relationship uses constraints, instead of being checked afterwards. *C-forall*, introduced by the compiler in the dual program (Appendix C.1), will check its consistency. For the query `?- valid_stream(Pr,Data)`, it returns: {`Pr=1, Data=p(A),A\=a, A \=b`} because `q(a)` and `q(b)` are more reliable than `p(X)`; {`Pr=2, Data=q(b)`}; and {`Pr=3, Data=p(a)`}. The justification tree and the model are in Appendix C.2.

The constraints and the goal-directed strategy of s(CASP) make it possible to resolve queries without evaluating the whole stream database. For example, the rule `incompt(p(X),q(X))` does not have to be grounded w.r.t. the stream database, and if timestamps were used as trustworthiness measure, for a query such as `?- T#>10,valid_stream(T,p(A))` the reasoner would validate streams received after `T=10` regardless how long they extend in the past.

4.2 *Yale Shooting Scenario*

In the spoiling Yale shooting scenario (Janhunen et al. 2017), there is a gun and three possible actions: *load*, *shoot*, and *wait*. If we load the gun and shoot within 35 minutes, the turkey is killed. Otherwise, the gun powder is spoiled. The executable plan must ensure that we kill the turkey within 100 minutes, assuming that we are not allowed to shoot in the first 35 minutes.

```
duration(load,25).
duration(shoot,5).
duration(wait,36).
spoiled(Armed) :- Armed #> 35.
prohibited(shoot,T) :- T #< 35.

holds(0,St,[]) :- init(St).
holds(F_Time, F_St, [Act|As]) :-
   F_Time #> 0,
   F_Time #= P_Time + Duration,
   duration(Act, Duration),
   not prohibited(Act, F_Time),
   trans(Act, P_St, F_St),
   holds(P_Time, P_St, As).
init(st(alive,unloaded,0)).

trans(load, st(alive,_,_),
            st(alive,loaded,0)).
trans(wait, st(alive,Gun,P_Ar),
            st(alive,Gun,F_Ar)) :-
   F_Ar #= P_Ar + Duration,
   duration(wait,Duration).
trans(shoot, st(alive,loaded,Armed),
             st(dead,unloaded,0)) :-
   not spoiled(Armed).
trans(shoot, st(alive,loaded,Armed),
             st(alive,unloaded,0)) :-
   spoiled(Armed).
```

Fig. 6: s(CASP) code for the Yale Shooting problem.

The ASP + constraint code, in (Janhunen et al. 2017) and Appendix D.1, uses *clingo[DL/LP]*, an ASP incremental solver extended for constraints. The program is parametric w.r.t. the step counter n, used by the solver to iteratively invoke the program with the expected length of the plan. In each iteration, the solver increases n, grounds the program with this value (which, in this example, specializes it for a plan of exactly n actions) and solves it. The execution returns two plans for $n = 3$: {`do(wait,1),do(load,2),do(shoot,3)`} and {`do(load,1),do(load,2),do(shoot,3)`}.

The s(CASP) code (Figure 6) does not need a counter. The query `?- T#<100,holds(T,st(dead,_,_),Actions)`, sets an upper bound to the duration `T` of the plan, and returns in `Actions` the plan with the actions in reverse chronological order: {`T=55, Actions=[shoot,load,load]`}, {`T=66, Actions=[shoot,load,wait]`}, {`T=80, Actions=[shoot,load,load, load]`}, {`T=91, Actions=[shoot,load,load, wait]`}, {`T=91, Actions=[shoot,load,wait, load]`}, {`T=96, Actions=[shoot,load,shoot, wait, load]`}.

4.3 The Traveling Salesman Problem (TSP)

Let us consider a variant of the traveling salesman problem (visiting every city in a country only once, starting and ending in the same city, and moving between cities using the existing connections) where we want to find out only the Hamiltonian cycles whose length is less than a given upper bound. Solutions for this problem, with comparable performance, using ASP and CLP(*FD*) appear in (Dovier et al. 2005) (also available in Appendix E.1 and E.2). The ASP encoding is more compact, even if the CLP(*FD*) version uses the non-trivial library predicate `circuit/1`, which does the bulk of the work. We will show that s(CASP) is more expressive also in this problem.

Finding the (bounded) path length in ASP requires using a specific, ad-hoc builtin that accesses the literals in a model and calls it from within a global constraint. Using *clasp* (Hölldobler and Schweizer 2014), it would be as follows:

```
cycle_length(N) :- N = #sum [cycle(X,Y) : distance(X, Y, C) = C].
:- cycle_length(N), N >= 10.        % Cycles whose length is less than 10
```

```
1  % Every node must be reachable.
2  :- node(U), not reachable(U).
3  reachable(a) :- cycle(V,a).
4  reachable(V) :- cycle(U,V),
5                  reachable(U).
6
7  % Only one edge to each node.
8  :- cycle(U,W), cycle(V,W), U \= V.
9
10 % Only one edge from each node.
11 cycle(U,V) :-
12   edge(U,V), not other(U,V).
13 other(U,V) :-
14   node(U), node(V), node(W),
15   edge(U,W), V \= W, cycle(U,W).
16
17 travel_path(S,Ln,Cycle) :-
18   path(S,S,S,Ln,[],Cycle).
19 path(_,X,Y,D,Ps,[X,[D],Y|Ps]) :-
20   cycle_dist(X,Y,D).
21 path(S,X,Y, D, Ps,Cs) :-
22   D #= D1 + D2,
23   cycle_dist(Z,Y,D1), Z \= S,
24   path(S,X,Z,D2,[[D1],Y|Ps],Cs).
25
26 edge(X,Y) :- distance(X,Y,D).
27 cycle_dist(U,V,D) :-
28   cycle(U,V), distance(U,V,D).
29
30 node(a).          node(b).
31 node(c).          node(d).
32
33 distance(b,c,31/10).
34 distance(c,d,L):-
35     L #> 8, L #< 21/2.
36 distance(d,a,1).
37 distance(a,b,1).
38 distance(a,d,1).
39 distance(c,a,1).
40 distance(d,b,1).
```

Fig. 7: Code for the Traveling Salesman problem.

where `#sum` is a builtin aggregate operator that here is used to add the distances between nodes in some Hamiltonian cycle.

The s(CASP) code in Figure 7 solves this TSP variant by modeling the Hamiltonian cycle in a manner similar to ASP and using a recursive predicate, `travel_path(S,Ln,Cycle)`, that returns in `Cycle` the list of nodes in the circuit (with the distance between every pair of nodes also in the list), starting at node `S`, and the total length of the circuit in `Ln`.

This example highlights the marriage between ASP encoding (to define models of the Hamiltonian cycle using the `cycle/2` literal) and traditional CLP (which uses the available `cycle/2` literals to construct paths and return their lengths). Note as well that we can define node distances as intervals (line 35) using a dense domain (rationals, in this case). This would not be straightforward (or even feasible) if only CLP(FD) was available: while CLP(FD) can encode CLP($\mathbb{Q}$), the resulting program would be cumbersome to maintain and much slower than the CLP($\mathbb{Q}$) version, since Gaussian elimination has to be replaced by enumeration, which actually compromises completeness (and, in the limit, termination). Additionally, in our proposal, constraints can appear in bindings and as part of the model. For example, the query `?- D#<10,travel_path(b,D, Cycle)` returns the model {`D=61/10,Cycle=[b, [31/10], c, [1], a, [1], d, [1], b]`}. For reference, Appendix E.3 shows the complete output.

4.4 Towers of Hanoi

We will not explain this problem here as it is widely known. Let us just remind the reader that solving the puzzle with three towers (the standard setup) and n disks requires at least $2^n - 1$ movements.

Known ASP encodings, for a *standard* solver, set a bound to the number of moves that can be done, as proposed in (Gebser et al. 2008) (available for the reader's convenience at Appendix F.1,

```
hanoi(N,T):-
   move_(N,0,T,a,b,c).
move_(N,Ti,Tf,Pi,Pf,Px) :-
   N #> 1, N1 #= N - 1,
   move_(N1,Ti,T1,Pi,Px,Pf),
   move_( 1,T1,T2,Pi,Pf,Px),
   move_(N1,T2,Tf,Px,Pf,Pi).
move_(1,Ti,Tf,Pi,Pf,_) :-
   Tf #= Ti + 1,
   move(Pi,Pf,Tf).
move(Pi,Pf,T):- not negmove(Pi,Pf,T).
negmove(Pi,Pf,T):- not move(Pi,Pf,T).

#show move/3. %s(CASP) directive
```

Fig. 8: s(CASP) code for the Towers of Hanoi.

	s(CASP)	**clingo 5.2.0**	**clingo 5.2.0**
		standard	*incremental*
$n = 7$	**479**	3,651	9,885
$n = 8$	**1,499**	54,104	174,224
$n = 9$	**5,178**	191,267	> 5 min

Table 2: Run time (ms) comparison for the Towers of Hanoi with n disks.

for `7` disks and up to `127` movements) or for an *incremental* solver, increasing the number n of allowed movements (from the *clingo 5.2.0* distribution, also available at Appendix F.2).

s(CASP)'s top down approach can use a CLP-like control strategy to implement the well-known Towers of Hanoi algorithm (Figure 8). Predicate `hanoi(N,T)` receives in `N` the number of disks and returns in `T` the number of movements needed to solve the puzzle. The resulting partial stable model will contain all the movements and the time in which they have to be performed. For reference, Appendix F.3 shows the partial stable model for `?- hanoi(7,T)`.

Table 2 compares execution time (in milliseconds) needed to solve the Towers of Hanoi with `n` disks by s(CASP) and *clingo 5.2.0* with the *standard* and *incremental* encodings. s(CASP) is orders of magnitude faster than both clingo variants because it does not have to generate and test all the possible plans; instead, as mentioned before, it computes directly the smallest solution to the problem. The standard variant is less interesting than s(CASP)'s, as it does not return the minimal number of moves — it merely checks if the problem can be solved in a given number of moves. The incremental variant is by far the slowest, because the program is iteratively checked with an increasing number of moves until it can be solved.

5 Conclusion and Future Work

We have reported on the design and implementation of s(CASP), a top-down system to evaluate constraint answer set programs, based on s(ASP). Its ability to express non-monotonic programs *à la* ASP is coupled with the possibility of expressing control in a way similar to traditional logic programming — and, in fact, a single program can use both approaches simultaneously, achieving the best of both worlds. We have also reported a very substantial performance increase w.r.t. the original s(ASP) implementation. Thanks to the possibility of writing pieces of code with control in mind, it can also beat state-of-the-art ASP systems in certain programs.

The implementation can still be improved substantially, as pointed out in the paper, and in particular we want to work on using analysis to optimize the compilation of non-monotonic check rules, being able to interleave their execution with the top-down strategy to discard models as soon as they are shown inconsistent, improve the disequality constraint solver to handle the pending cases, use dependency analysis to improve the generation of the dual programs, and apply partial evaluation and better compilation techniques to remove (part of) the overhead brought about by the interpreting approach.

References

ALFERES, J. J., PEREIRA, L. M., AND SWIFT, T. 2004. Abduction in Well-Founded Semantics and Generalized Stable Models via Tabled Dual Programs. *Theory and Practice of Logic Programming 4,* 4, 383–428.

ALVIANO, M., FABER, W., GRECO, G., AND LEONE, N. 2012. Magic Sets for Disjunctive Datalog Programs. *Artificial Intelligence 187*, 156–192.

ARIAS, J. 2016. Tabled CLP for Reasoning over Stream Data. In *Technical Communications of the 32nd Int'l Conference on Logic Programming (ICLP'16).* Vol. 52. OASIcs, 1–8. Doctoral Consortium.

ARIAS, J. AND CARRO, M. 2016. Description and Evaluation of a Generic Design to Integrate CLP and Tabled Execution. In *18th Int'l. ACM SIGPLAN Symposium on Principles and Practice of Declarative Programming (PPDP'16).* ACM Press, 10–23.

BALDUCCINI, M. AND LIERLER, Y. 2017. Constraint Answer Set Solver EZCSP and why Integration Schemas Matter. *Theory and Practice of Logic Programming 17,* 4, 462–515.

BANBARA, M., KAUFMANN, B., OSTROWSKI, M., AND SCHAUB, T. 2017. Clingcon: The Next Generation. *Theory and Practice of Logic Programming 17,* 4, 408–461.

BASELICE, S. AND BONATTI, P. A. 2010. A Decidable Subclass of Finitary Programs. *Theory and Practice of Logic Programming 10,* 4-6, 481–496.

BASELICE, S., BONATTI, P. A., AND CRISCUOLO, G. 2009. On Finitely Recursive Programs. *Theory and Practice of Logic Programming 9,* 2, 213–238.

BREWKA, G., EITER, T., AND TRUSZCZYŃSKI, M. 2011. Answer Set Programming at a Glance. *Communications of the ACM 54,* 12, 92–103.

CALIMERI, F., COZZA, S., AND IANNI, G. 2007. External Sources of Knowledge and Value Invention in Logic Programming. *Annals of Mathematics and Artificial Intelligence 50,* 3-4, 333–361.

CLARK, K. L. 1978. Negation as Failure. In *Logic and Data Bases*, H. Gallaire and J. Minker, Eds. Plenum.

DAL PALÙ, A., DOVIER, A., PONTELLI, E., AND ROSSI, G. 2009. GASP: Answer Set Programming with Lazy Grounding. *Fundamenta Informaticae 96,* 3, 297–322.

DOVIER, A., FORMISANO, A., AND PONTELLI, E. 2005. A Comparison of CLP(FD) and ASP Solutions to NP-Complete Problems. In *International Conference on Logic Programming*. Springer, 67–82.

GABBRIELLI, M. AND LEVI, G. 1991. Modeling Answer Constraints in Constraint Logic Programs. In *Proc. Eighth Int'l Conf. on Logic Programming*.

GEBSER, M., KAMINSKI, R., KAUFMANN, B., OSTROWSKI, M., SCHAUB, T., AND THIELE, S. 2008. A User's Guide to gringo, clasp, clingo, and iclingo.

GELFOND, M. AND LIFSCHITZ, V. 1988. The Stable Model Semantics for Logic Programming. In *International Conference on Logic Programming 1988*. 1070–1080.

GELFOND, M. AND LIFSCHITZ, V. 1991. Classical Negation in Logic Programs and Disjunctive Databases. *New Generation Computing 9,* 3/4, 365–386.

GUPTA, G., BANSAL, A., MIN, R., SIMON, L., AND MALLYA, A. 2007. Coinductive Logic Programming and its Applications. *Logic Programming*, 27–44.

HERMENEGILDO, M. V., BUENO, F., CARRO, M., LÓPEZ, P., MERA, E., MORALES, J., AND PUEBLA, G. 2012. An Overview of Ciao and its Design Philosophy. *Theory and Practice of Logic Programming 12,* 1–2 (January), 219–252. http://arxiv.org/abs/1102.5497.

HÖLLDOBLER, S. AND SCHWEIZER, L. 2014. Answer Set Programming and clasp, a Tutorial. In *Young Scientists' International Workshop on Trends in Information Processing (YSIP)*. 77.

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

JAFFAR, J. AND MAHER, M. 1994. Constraint Logic Programming: A Survey. *Journal of Logic Programming 19/20*, 503–581.

JANHUNEN, T., KAMINSKI, R., OSTROWSKI, M., SCHELLHORN, S., WANKO, P., AND SCHAUB, T. 2017. Clingo goes Linear Constraints over Reals and Integers. *TPLP 17,* 5-6, 872–888.

JI, J., WAN, H., WANG, K., WANG, Z., ZHANG, C., AND XU, J. 2016. Eliminating Disjunctions in Answer Set Programming by Restricted Unfolding. In *IJCAI*. 1130–1137.

MARPLE, K., SALAZAR, E., CHEN, Z., AND GUPTA, G. 2017a. The s(ASP) Predicate Answer Set Programming System. *The Association for Logic Programming Newsletter*.

MARPLE, K., SALAZAR, E., AND GUPTA, G. 2017b. Computing Stable Models of Normal Logic Programs Without Grounding. *CoRR abs/1709.00501*.

REVESZ, P. Z. 1993. A Closed-Form Evaluation for Datalog Queries with Integer (Gap)-Order Constraints. *Theoretical Computer Science 116,* 1, 117–149.

Appendix A s(CASP) interpreter

The next figure shows a sketch of the s(CASP) interpreter's code implemented in Ciao Prolog.

```
??(Query) :-
  solve(Query,[],Mid),
  solve_goal(nmr_check,Mid,Just),
  print_just_model(Just).

solve([],In,['$success'|In]).
solve([Goal|Gs],In,Out) :-
  solve_goal(Goal,In,Mid),
  solve(Gs,Mid,Out).

solve_goal(Goal,In,Out) :-
  user_defined(Goal),!,
  check_loops(Goal,In,Out).
solve_goal(Goal,In,Out) :-
  Goal = forall(Var,G),!,
  forall(V,G,In,Out).
solve_goal(Goal,In,Out) :-
  call(Goal),
  Out = ['$success',Goal|In].

check_loops(Goal,In Out) :-
  type_loop(Goal,In,Loop),
  solve_loop(Loop,Goal,In,Out).

solve_loop(odd,_,_,_) :- fail.
solve_loop(pos,_,_,_) :- fail.
solve_loop(eve,G,In,[chs(G)|In]).
solve_loop(pro,G,In,[pro(G)|In]).
solve_loop(cont,G,In,Out) :-
  pr_rule(G, Body),
  solve(Body,[G|In],Out).

forall(V,Goal,In,Out) :-
  empty_store(Store),
  eval_forall(V,Goal,[Store],In,Out).
eval_forall(_,_,[],In,In).
eval_forall(V,Goal,[Store|Sts],In,Out) :-
  copy(V,Goal, NV,NGoal),
  apply(NV, V,Store),
  solve([NGoal],In,['$success'|Out_1]),
  dump(NV, V,AnsSt),
  (  equal(AnsSt,Store)
  -> Out_2  =  Out_1
  ;  dual(AnsSt,AnsDs),
     add(AnsDs,Store,NSt),
     eval_forall(V,Goal,NSt,Out_1,Out_2)
  ),
  eval_forall(V,Goal,Sts,Out_2,Out).
```

Appendix B Handling Loops

Top-down evaluations may enter loops. Several techniques, notably tabling, have been used to enhance the termination properties of LP systems. This is more relevant in s(ASP) because the presence of negation introduces new types of loops:

- **Odd loop over negation**: it occurs when a cycle in the call graph contains an odd number of intervening negations. These loops are important because they place global constraints which restrict which literals can appear in a model. s(ASP) ensures that these global constraints are satisfied by introducing *non monotonic rules* (Section 2.4). The odd loops are detected with a static analysis of the call graph checking the number of negations between recursive calls.

  Example 8. The rules below, which are equivalent if p/0 can not be added to the model by another rule, generate odd loops and force the stable model to satisfy $\neg\, q(a)$.

  ```
  p :- q(a), not p.               :- q(a).
  ```

  Run-time check of odd loops When, during the execution, a call unifies with its negation in the call path, the execution fails and backtracks. Had it succeeded, it would have introduced a contradiction, and therefore the resulting partial stable model would have been discarded.

- **Even loop over negation**: This happens when a call unifies with an ancestor in the call path and there is an even, non-zero, number of intervening negated calls between them. In this case, the execution succeeds assuming that the recursive call (partially) supports the *negation* of those calls. The spirit underlying this assumption is similar to coinductive SLD resolution (Gupta et al. 2007), used to compute the greatest fixpoint of a program. Note that the Gelfond–Lifschitz method computes the fixpoint of the residual program, which is between the least fixpoint (computed by a top-down execution) and the greatest fixpoint. This assumption is safe because in cases where the evaluation tries to make this recursive call *true*, the *non monotonic rules* and the run-time detection of odd loops will discard the model.

  Example 9. Consider the next program (with its dual) and the query `?- p(a)`.

  ```
  p(X) :- not q(X).                 not p(X) :- not p1(X).
                                    not p1(X) :- q(X).
  q(X) :- not p(X).                 not q(X) :- not q1(X), not q2(X).
  q(b).                             not q1(X) :- p(X).
                                    not q2(X) :- X \= b.
  ```

  The call path `p(a)`⇝`not q(a)`⇝`not q1(a)`⇝`p(a)`, resulting from the query, shows that assuming `p(a)` we support both negated calls (i.e., `not q(a)` and `not q1(a)`). Note that `not q(a)` is only partially supported because it succeeds only if also `not q2(X)` succeeds. Therefore, while the query `?- p(a)` succeeds, the query `?- p(b)` fails.

- **Positive loops**: when a call unifies with an ancestor in the call path and there are no intervening negative calls between them, the original s(ASP) fails to avoid infinite loops. However, this behaviour compromises completeness and soundness. We work around this by checking that the call and its ancestor are equal (Section 3.2).

  Example 10. The next program generates infinitely many answers to the query `?- nat(X)`.

  ```
  nat(0).                nat(X) :- nat(Y), X is Y+1.
  ```

  However, if it fails, when the recursive call `nat(Y)` unifies with its ancestor in the call path (i.e., the query), it loses completeness as it only returns the answer `X=0`, and therefore, due to the presence of negation, it also loses soundness.

Appendix C Stream Data Reasoning Example

C.1 s(CASP) encoding of stream.pl

The next figure shows the code of stream.pl with the dual program and the NMR generated by the extended compiler of s(CASP).

```
valid_stream(P,Data) :-                    cancelled(P,Data) :-
    stream(P,Data),                            higher_prio(P1,P),
    not cancelled(P,Data).                     stream(P1,Data1),
                                               incompt(Data,Data1).
```

```

higher_prio(PHi,PLo) :-
    PHi #> PLo.

incompt(p(X),q(X)).
incompt(q(X),p(X)).

stream(1,p(X)).
stream(2,q(a)).
stream(2,q(b)).
stream(3,p(a)).

not incompt1(A,_,X) :-
    A \= p(X).
not incompt1(A,B,X) :-
    A = p(X),
    B \= q(X).

not incompt1(A,B) :-
    forall(X,not incompt1(A,B,X)).

not incompt2(A,_,X) :-
    A \= q(X).
not incompt2(A,B,X) :-
    A = q(X),
    B \= p(X).

not incompt2(A,B) :-
    forall(X,not incompt2(A,B,X)).

not incompt(A,B) :-
    not incompt1(A,B),
    not incompt2(A,B).

not higher_prio1(PHi,PLo) :-
    PHi #=< PLo.

not higher_prio(A,B) :-
    not higher_prio1(A,B).

not cancelled1(P,_,P1,_) :-
    not higher_prio(P1,P).
not cancelled1(P,_,P1,Data1) :-
    higher_prio(P1,P),
    not stream(P1,Data1).
not cancelled1(P,Data,P1,Data1) :-
    higher_prio(P1,P),
    stream(P1,Data1),
    not incompt(Data,Data1).

not cancelled1(P,Data) :-
    forall(P1,forall(Data1,not
cancelled1(P,Data,P1,Data1))).

not cancelled(A,B) :-
    not cancelled1(A,B).

not stream1(A,_,X) :-
    A \= 1.
not stream1(A,B,X) :-
    A = 1,
    B \= p(X).

not stream1(A,B) :-
    forall(X,not stream1(A,B,X)).

not stream2(A,_) :-
    A \= 2.
not stream2(A,B) :-
    A = 2,
    B \= q(a).

not o_stream3(A,_) :-
      A \= 2.
not o_stream3(A,B) :-
      A = 2,
      B \= q(b).

not stream4(A,_) :-
    A \= 3.
not stream4(A,B) :-
    A = 3,
    B \= p(a).

not stream(A,B) :-
    not stream1(A,B),
    not stream2(A,B),
    not stream3(A,B),
    not stream4(A,B).

not valid_stream1(P,Data) :-
    not stream(P,Data).
not valid_stream1(P,Data) :-
    stream(P,Data),
    cancelled(P,Data).

not valid_stream(A,B) :-
    not valid_stream1(A,B).

not false.

nmr_check.
```

C.2 s(CASP) output of stream.pl

The next figure shows the output for the query `?- valid_stream(Pr,Data)` when it is made to the program `stream.pl` (see Appendix C.1). The output to a query consists of: (i) a justification tree with the successful derivation (note that variables could be free, ground, or constrained); (ii) a model with the positive atoms defined by the program that support the successful derivation; and (iii) the bindings of variables in the query (in this example, the bindings of `Pr` and `Data`). The constraint store active at each call is shown close to each variable.

```
?- valid_stream(Pr, Data).

Answer 1        (in 18.907 ms):

valid_stream(1,p({A \= [a,b]})) :-
   stream(1,p({A \= [a,b]})),
   not cancelled(1,p({A \= [a,b]})) :-
      not o_cancelled1(1,p({A \= [a,b]})) :-
         forall(B,forall(C,not o_cancelled1(1,p({A \= [a,b]}),B,C))) :-
            forall(C,not o_cancelled1(1,p({A \= [a,b]}),{D #=< 1},C)) :-
               not o_cancelled1(1,p({A \= [a,b]}),{D #=< 1},C) :-
                  not higher_prio({D #=< 1},1) :-
                     not o_higher_prio1({D #=< 1},1) :-
                        {D #=< 1} #=< 1.
            forall(C,not o_cancelled1(1,p({A \= [a,b]}),{E #> 3},C)) :-
               not o_cancelled1(1,p({A \= [a,b]}),{E #> 3},F) :-
                  higher_prio({E #> 3},1) :-
                     {E #> 3} #> 1.
                  not stream({E #> 3},F) :-
                     not o_stream1({E #> 3},F) :-
                        forall(G,not o_stream1({E #> 3},F,G)) :-
                           not o_stream1({E #> 3},F,G) :-
                              {E #> 3} \= 1.
                     not o_stream2({E #> 3},F) :-
                        {E #> 3} \= 2.
                     not o_stream3({E #> 3},F) :-
                        {E #> 3} \= 2.
                     not o_stream4({E #> 3},F) :-
                        {E #> 3} \= 3.
            forall(C,not o_cancelled1(1,p({A \= [a,b]}),{H #> 2, H #< 3},C)) :-
               not o_cancelled1(1,p({A \= [a,b]}),{H #> 2, H #< 3},I) :-
                  higher_prio({H #> 2, H #< 3},1) :-
                     {H #> 2, H #< 3} #> 1.
                  not stream({H #> 2, H #< 3},I) :-
                     not o_stream1({H #> 2, H #< 3},I) :-
                        forall(J,not o_stream1({H #> 2, H #< 3},I,J)) :-
                           not o_stream1({H #> 2, H #< 3},I,J) :-
                              {H #> 2, H #< 3} \= 1.
                     not o_stream2({H #> 2, H #< 3},I) :-
                        {H #> 2, H #< 3} \= 2.
                     not o_stream3({H #> 2, H #< 3},I) :-
                        {H #> 2, H #< 3} \= 2.
                     not o_stream4({H #> 2, H #< 3},I) :-
                        {H #> 2, H #< 3} \= 3.
            forall(C,not o_cancelled1(1,p({A \= [a,b]}),{K #> 1, K #< 2},C)) :-
   not o_cancelled1(1,p({A \= [a,b]}),{K #> 1, K #< 2},L) :-
      higher_prio({K #> 1, K #< 2},1) :-
         {K #> 1, K #< 2} #> 1.
      not stream({K #> 1, K #< 2},L) :-
         not o_stream1({K #> 1, K #< 2},L) :-
            forall(M,not o_stream1({K #> 1, K #< 2},L,M)) :-
               not o_stream1({K #> 1, K #< 2},L,M) :-
                  {K #> 1, K #< 2} \= 1.
         not o_stream2({K #> 1, K #< 2},L) :-
            {K #> 1, K #< 2} \= 2.
         not o_stream3({K #> 1, K #< 2},L) :-
            {K #> 1, K #< 2} \= 2.
         not o_stream4({K #> 1, K #< 2},L) :-
            {K #> 1, K #< 2} \= 3.
forall(C,not o_cancelled1(1,p({A \= [a,b]}),2,C)) :-
   not o_cancelled1(1,p({A \= [a,b]}),2,{N \= [q(a),q(b)]}) :-
      higher_prio(2,1) :-
         2 #> 1.
      not stream(2,{N \= [q(a),q(b)]}) :-
         not o_stream1(2,{N \= [q(a),q(b)]}) :-
            forall(O,not o_stream1(2,{N \= [q(a),q(b)]},O)) :-
               not o_stream1(2,{N \= [q(a),q(b)]},O) :-
                  2 \= 1.
         not o_stream2(2,{N \= [q(a),q(b)]}) :-
            2 = 2,
           {N \= [q(a),q(b)]} \= q(a).
         not o_stream3(2,{N \= [q(a),q(b)]}) :-
            2 = 2,
           {N \= [q(a),q(b)]} \= q(b).
         not o_stream4(2,{N \= [q(a),q(b)]}) :-
            2 \= 3.
   not o_cancelled1(1,p({A \= [a,b]}),2,q(a)) :-
      proved(higher_prio(2,1)),
      stream(2,q(a)),
      not incompt(p({A \= [a,b]}),q(a)) :-
         not o_incompt1(p({A \= [a,b]}),q(a)) :-
            forall(P,not o_incompt1(p({A \= [a,b]}),q(a),P)) :-
               not o_incompt1(p({A \= [a,b]}),q(a),{A \= [a,b]}) :-
                  p({A \= [a,b]}) = p({A \= [a,b]}),
                  q(a) \= q({A \= [a,b]}).
               not o_incompt1(p({A \= [a,b]}),q(a),a) :-
                  p({A \= [a,b]}) \= p(a).
         not o_incompt2(p({A \= [a,b]}),q(a)) :-
            forall(P,not o_incompt2(p({A \= [a,b]}),q(a),P)) :-
               not o_incompt2(p({A \= [a,b]}),q(a),P) :-
                  p({A \= [a,b]}) \= q(P).
   not o_cancelled1(1,p({A \= [a,b]}),2,q(b)) :-
      proved(higher_prio(2,1)),
      stream(2,q(b)),
      not incompt(p({A \= [a,b]}),q(b)) :-
         not o_incompt1(p({A \= [a,b]}),q(b)) :-
            forall(Q,not o_incompt1(p({A \= [a,b]}),q(b),Q)) :-
               not o_incompt1(p({A \= [a,b]}),q(b),{A \= [a,b]}) :-
                  p({A \= [a,b]}) = p({A \= [a,b]}),
                          q(b) \= q({A \= [a,b]}).
                       not o_incompt1(p({A \= [a,b]}),q(b),a) :-
                          p({A \= [a,b]}) \= p(a).
                       not o_incompt1(p({A \= [a,b]}),q(b),b) :-
                          p({A \= [a,b]}) \= p(b).
                 not o_incompt2(p({A \= [a,b]}),q(b)) :-
                    forall(R,not o_incompt2(p({A \= [a,b]}),q(b),R)) :-
                       not o_incompt2(p({A \= [a,b]}),q(b),R) :-
                          p({A \= [a,b]}) \= q(R).
           forall(C,not o_cancelled1(1,p({A \= [a,b]}),3,C)) :-
              not o_cancelled1(1,p({A \= [a,b]}),3,{S \= [p(a)]}) :-
                 higher_prio(3,1) :-
                    3 #> 1.
                 not stream(3,{S \= [p(a)]}) :-
                    not o_stream1(3,{S \= [p(a)]}) :-
                       forall(T,not o_stream1(3,{S \= [p(a)]},T)) :-
                          not o_stream1(3,{S \= [p(a)]},T) :-
                             3 \= 1.
                    not o_stream2(3,{S \= [p(a)]}) :-
                       3 \= 2.
                    not o_stream3(3,{S \= [p(a)]}) :-
                       3 \= 2.
                    not o_stream4(3,{S \= [p(a)]}) :-
                       3 = 3,
                      {S \= [p(a)]} \= p(a).
              not o_cancelled1(1,p({A \= [a,b]}),3,p(a)) :-
                 proved(higher_prio(3,1)),
                 stream(3,p(a)),
                 not incompt(p({A \= [a,b]}),p(a)) :-
                    not o_incompt1(p({A \= [a,b]}),p(a)) :-
                       forall(U,not o_incompt1(p({A \= [a,b]}),p(a),U)) :-
                          not o_incompt1(p({A \= [a,b]}),p(a),{A \= [a,b]}) :-
                             p({A \= [a,b]}) = p({A \= [a,b]}),
                             p(a) \= q({A \= [a,b]}).
                          not o_incompt1(p({A \= [a,b]}),p(a),a) :-
                             p({A \= [a,b]}) \= p(a).
                          not o_incompt1(p({A \= [a,b]}),p(a),b) :-
                             p({A \= [a,b]}) \= p(b).
                    not o_incompt2(p({A \= [a,b]}),p(a)) :-
                       forall(V,not o_incompt2(p({A \= [a,b]}),p(a),V)) :-
                          not o_incompt2(p({A \= [a,b]}),p(a),V) :-
                             p({A \= [a,b]}) \= q(V).
add_to_query :- o_nmr_check.

[ valid_stream(1,p({A \= [a,b]})), stream(1,p({A \= [a,b]})), higher_prio({E #> 3},1),
  higher_prio({H #> 2, H #< 3},1), higher_prio({K #> 1, K #< 2},1), higher_prio(2,1),
  stream(2,q(a)), stream(2,q(b)), higher_prio(3,1), stream(3,p(a)), o_nmr_check ]

Pr  =  1,
Data  =  p({A \= [a,b]}) ? ;

Answer 2        (in 49.191 ms):

valid_stream(2,q(b)) :-
```

```
stream(2,q(b)),
not cancelled(2,q(b)) :-
   not o_cancelled1(2,q(b)) :-
      forall(B,forall(C,not o_cancelled1(2,q(b),B,C))) :-
         forall(C,not o_cancelled1(2,q(b),{A #=< 2},C)) :-
            not o_cancelled1(2,q(b),{A #=< 2},D) :-
               not higher_prio({A #=< 2},2) :-
                  not o_higher_prio1({A #=< 2},2) :-
                     {A #=< 2} #=< 2.
         forall(C,not o_cancelled1(2,q(b),{E #> 3},C)) :-
            not o_cancelled1(2,q(b),{E #> 3},F) :-
               higher_prio({E #> 3},2) :-
                  {E #> 3} #> 2.
               not stream({E #> 3},F) :-
                  not o_stream1({E #> 3},F) :-
                     forall(G,not o_stream1({E #> 3},F,G)) :-
                        not o_stream1({E #> 3},F,G) :-
                           {E #> 3} \= 1.
                  not o_stream2({E #> 3},F) :-
                     {E #> 3} \= 2.
                  not o_stream3({E #> 3},F) :-
                     {E #> 3} \= 2.
                  not o_stream4({E #> 3},F) :-
                     {E #> 3} \= 3.
         forall(C,not o_cancelled1(2,q(b),{H #> 2, H #< 3},C)) :-
            not o_cancelled1(2,q(b),{H #> 2, H #< 3},I) :-
               higher_prio({H #> 2, H #< 3},2) :-
                  {H #> 2, H #< 3} #> 2.
               not stream({H #> 2, H #< 3},I) :-
                  not o_stream1({H #> 2, H #< 3},I) :-
                     forall(J,not o_stream1({H #> 2, H #< 3},I,J)) :-
                        not o_stream1({H #> 2, H #< 3},I,J) :-
                           {H #> 2, H #< 3} \= 1.
                  not o_stream2({H #> 2, H #< 3},I) :-
                     {H #> 2, H #< 3} \= 2.
                  not o_stream3({H #> 2, H #< 3},I) :-
                     {H #> 2, H #< 3} \= 2.
                  not o_stream4({H #> 2, H #< 3},I) :-
                     {H #> 2, H #< 3} \= 3.
         forall(C,not o_cancelled1(2,q(b),3,C)) :-
            not o_cancelled1(2,q(b),3,{K \= [p(a)]}) :-
               higher_prio(3,2) :-
                  3 #> 2.
               not stream(3,{K \= [p(a)]}) :-
                  not o_stream1(3,{K \= [p(a)]}) :-
                     forall(L,not o_stream1(3,{K \= [p(a)]},L)) :-
                        not o_stream1(3,{K \= [p(a)]},L) :-
                           3 \= 1.
                  not o_stream2(3,{K \= [p(a)]}) :-
                     3 \= 2.
                  not o_stream3(3,{K \= [p(a)]}) :-
                     3 \= 2.
                  not o_stream4(3,{K \= [p(a)]}) :-
                     3 = 3,
                        {K \= [p(a)]} \= p(a).
                not o_cancelled1(2,q(b),3,p(a)) :-
                   proved(higher_prio(3,2)),
                   stream(3,p(a)),
                   not incompt(q(b),p(a)) :-
                      not o_incompt1(q(b),p(a)) :-
                         forall(M,not o_incompt1(q(b),p(a),M)) :-
                            not o_incompt1(q(b),p(a),M) :-
                               q(b) \= p(M).
                      not o_incompt2(q(b),p(a)) :-
                         forall(N,not o_incompt2(q(b),p(a),N)) :-
                            not o_incompt2(q(b),p(a),{O \= [b]}) :-
                               q(b) \= q({O \= [b]}).
                            not o_incompt2(q(b),p(a),b) :-
                               q(b) = q(b),
                               p(a) \= p(b).
add_to_query :- o_nmr_check.


[ valid_stream(2,q(b)), stream(2,q(b)), higher_prio({E #> 3},2), higher_
  prio({H #> 2, H #< 3},2), higher_prio(3,2), stream(3,p(a)), o_nmr_check ]

Pr  =  2,
Data  =  q(b) ? ;

Answer 3        (in 1.606 ms):

valid_stream(3,p(a)) :-
   stream(3,p(a)),
   not cancelled(3,p(a)) :-
      not o_cancelled1(3,p(a)) :-
         forall(B,forall(C,not o_cancelled1(3,p(a),B,C))) :-
            forall(C,not o_cancelled1(3,p(a),{A #=< 3},C)) :-
               not o_cancelled1(3,p(a),{A #=< 3},D) :-
                  not higher_prio({A #=< 3},3) :-
                     not o_higher_prio1({A #=< 3},3) :-
                        {A #=< 3} #=< 3.
            forall(C,not o_cancelled1(3,p(a),{E #> 3},C)) :-
               not o_cancelled1(3,p(a),{E #> 3},F) :-
                  higher_prio({E #> 3},3) :-
                     {E #> 3} #> 3.
                  not stream({E #> 3},F) :-
                     not o_stream1({E #> 3},F) :-
                        forall(G,not o_stream1({E #> 3},F,G)) :-
                           not o_stream1({E #> 3},F,G) :-
                              {E #> 3} \= 1.
                     not o_stream2({E #> 3},F) :-
                        {E #> 3} \= 2.
                     not o_stream3({E #> 3},F) :-
                        {E #> 3} \= 2.
                     not o_stream4({E #> 3},F) :-
                        {E #> 3} \= 3.
add_to_query :- o_nmr_check.
```

```
[ valid_stream(3,p(a)), stream(3,p(a)), higher_prio({E #> 3},3), o_nmr_check ]

Pr  =  3,
Data  =  p(a) ? ;

no
```

Appendix D Yale Scenario Example

D.1 ASP + constraint encoding of yale_shooting_asp.pl

Nest figure shows the spoiled Yale shooting scenario model written in clingo + constraints using multi-shot solving (Janhunen et al. 2017).

```
#include "incmode_lc.lp".
#program base.
action(load).
action(shoot).
action(wait).
duration(load,25).
duration(shoot,5).
duration(wait,36).
unloaded(0).
&sum { at(0) } = 0.
&sum { armed(0) } = 0.

#program step(n).
1 { do(X,n) : action(X) } 1.
&sum { at(n),-1*at(N') } = D :-
     do(X,n),
     duration(X,D),
     N' = n - 1.

loaded(n) :-
     loaded(n-1),
     not unloaded(n).
unloaded(n) :-
     unloaded(n-1),
     not loaded(n).
dead(n) :-
     dead(n-1).

&sum { armed(n) } = 0 :-
     unloaded(n-1).
&sum { armed(n),-1*armed(N') } = D :-
     do(X,n),
     duration(X,D),
     N' = n - 1,
     loaded(N').

loaded(n) :-
     do(load,n).
unloaded(n) :-
     do(shoot,n).
dead(n) :-
     do(shoot,n),
     &sum { armed(n) } <= 35.

:- do(shoot,n), unloaded(n-1).

#program check(n).
:- not dead(n), query(n).
:- not &sum {at(n)} <= 100, query(n).
:- do(shoot,n), not &sum {at(n)} > 35.
```

Appendix E The Traveling Salesman Problem Example

E.1 ASP encoding of hamicycle_asp.pl

The next figure shows an ASP program for the Travelling Salesman Problem described in section 4.3. The encoding for the Hamiltonian cycle part is from (Dovier et al. 2005) and the code of `#sum` is adapted to run using *clingo*. The bound on the total distance is one of the global constraints in the program.

```
1 {cycle(X,Y) : edge(X,Y)} 1 :- node(X).
1 {cycle(Z,X) : edge(Z,X)} 1 :- node(X).

```

```
reachable(X) :- node(X), cycle(b,X).
reachable(Y) :- node(X), node(Y), reachable(X), cycle(X,Y).

:- not reachable(X), node(X).

cycle_length(N) :- N = #sum {C: cycle(X,Y), distance(X, Y, C)}.
:- cycle_length(N), N >= 10.      % Cycles whose length is less than 10

edge(X,Y) :- distance(X,Y,D).
cycle_dist(U,V,D) :- cycle(U,V), distance(U,V,D).

node(a).         node(b).          node(c).           node(d).
distance(b,c,3).    %% ASP does not support rationals.
distance(c,d,8).    %% ASP does not support intervals.
distance(d,a,1).          distance(c,a,1).         distance(d,b,1).
distance(a,b,1).          distance(a,d,1).
```

E.2 CLP(FD) encoding of hamicycle_clpfd.pl

The next figure shows the program in CLP(*FD*) for the Hamiltonian cycle problem presented in (Dovier et al. 2005), using SICStus Prolog 3.11.2. Note that the library predicate `circuit/1` does the bulk of the work. Its implementation is non-trivial and shares a lot of code with the implementation of *all_different*, and it implicitly imposes that constraint. It does not calculate cycle lengths, but even in this (simplified) case, the code as a whole is much larger that either the ASP or s(CASP) code.

```
hamiltonian_path(Path) :-
   graph(Nodes,Edges),
   length(Nodes,N),
   length(Path,N),
   domain(Path,1,N),
   make_domains(Path,1,Edges,N),
   circuit(Path),
   labeling([ff],Path).

make_domains([],_,_,_).
make_domains([X|Y],Node,Edges,N) :-
   findall(Z,
     member([Node,Z],Edges),Succs),
   reduce_domains(N,Succs,X),
   Node1 is Node + 1,
   make_domains(Y,Node1,Edges,N).

reduce_domains(0,_,_) :- !.
reduce_domains(N,Succs,Var) :-
   N > 0,
   member(N,Succs), !,
   N1 is N - 1,
   reduce_domains(N1,Succs,Var).
reduce_domains(N,Succs,Var) :-
   Var #\=  N,
   N1 is N - 1,
   reduce_domains(N1,Succs,Var).
```

E.3 s(CASP) output of hamicycle_scasp.pl

The next figure shows the output to the query `?- D#<10,cycle(a,D,Cycle)` to the program hamicycle_scasp.pl (Figure 7 in Section 4.3).

```
?- D #< 10, travel_path(b, D, Cycle).

Answer 1        (in [2346.489] ms):

[ travel_path(b,61/10,[b,[31/10],c,[1],a,[1],d,[1],b]), path(b,b,b,61/10,[],
  [b,[31/10],c,[1],a,[1],d,[1],b]), cycle_dist(d,b,1), cycle(d,b), edge(d,b),
  distance(d,b,1), node(d), node(b), node(a), edge(d,a), distance(d,a,1),
  other(d,a), node(b), cycle(d,b), node(c), distance(d,b,1), path(b,b,d,51/10,
  [[1],b],[b,[31/10],c,[1],a,[1],d,[1],b]), cycle_dist(a,d,1), cycle(a,d),
  edge(a,d), distance(a,d,1), edge(a,b), distance(a,b,1), other(a,b), node(d),
  cycle(a,d), distance(a,d,1), path(b,b,a,41/10,[[1],d,[1],b],[b,[31/10],c,[1],
  a,[1],d,[1],b]), cycle_dist(c,a,1), cycle(c,a), edge(c,a), distance(c,a,1),
  edge(c,d), distance(c,d,{A #> 8, A #< 21rat2}), other(c,d), node(a), cycle(c,a),
  distance(c,a,1), path(b,b,c,31/10,[[1],a,[1],d,[1],b],[b,[31/10],c,[1],a,[1],
  d,[1],b]), cycle_dist(b,c,31/10), cycle(b,c), edge(b,c), distance(b,c,3.1),
  distance(b,c,31/10), o_nmr_check, reachable(a), cycle(c,a), edge(c,a),
  distance(c,a,1), reachable(b), cycle(d,b), edge(d,b), distance(d,b,1),
  reachable(d), cycle(a,d), edge(a,d), distance(a,d,1), reachable(c), cycle(b,c),
  edge(b,c), distance(b,c,3.1), other(a,a), node(d), other(a,c), node(d),
  other(b,a), node(c), other(b,b), node(c), other(b,d), node(c), other(c,b),
  node(a), other(c,c), node(a), other(d,c), node(b), other(d,d), node(b) ]

Cycle  =  [b,[31/10],c,[1],a,[1],d,[1],b],
D  =  61/10 ?
```

Appendix F Towers of Hanoi Example

F.1 ASP encoding of toh_asp.pl

The next program is part of (Gebser et al. 2008):

```
% Instance
peg(a;b;c).
disk(1..7).
init_on(1..7,a).
goal_on(1..7,b).
moves(127).
% Generate
1 { move(D,P,T) : disk(D) : peg(P) } 1 :- moves(M), T = 1..M.
% Define
move(D,T) :- move(D,_,T).
on(D,P,0) :- init_on(D,P).
on(D,P,T) :- move(D,P,T).
on(D,P,T+1) :- on(D,P,T), not move(D,T+1), not moves(T).
blocked(D-1,P,T+1) :- on(D,P,T), disk(D), not moves(T).
blocked(D-1,P,T) :- blocked(D,P,T), disk(D).
% Test
:- move(D,P,T), blocked(D-1,P,T).
:- move(D,T), on(D,P,T-1), blocked(D,P,T).
:- goal_on(D,P), not on(D,P,M), moves(M).
:- not 1 { on(D,P,T) : peg(P) } 1, disk(D), moves(M), T = 1..M.
#hide.
#show move/3.
```

F.2 ASP incremental encoding of toh_aspI.pl

The next program is part of the *clingo* distribution and is available at `https://github.com/potassco/clingo/tree/master/examples/gringo/toh`

```
#include <incmode>.

#program base.
peg(a;b;c).
disk(1..7).
init_on(1..7,a).
goal_on(1..7,b).

on(D,P,0) :- init_on(D,P).

#program step(t).
1 {move(D,P,t): disk(D),peg(P)} 1.

move(D,t) :- move(D,P,t).
on(D,P,t) :- move(D,P,t).
on(D,P,t) :- on(D,P,t-1),
             not move(D,t).
blocked(D-1,P,t) :- on(D,P,t-1).
blocked(D-1,P,t) :- blocked(D,P,t),
                    disk(D).
:- move(D,P,t), blocked(D-1,P,t).
:- move(D,t), on(D,P,t-1), blocked(D,P,t).
:- not 1 { on(D,P,t) } 1, disk(D).

#program check(t).
:- query(t), goal_on(D,P), not on(D,P,t).

#show move/3.
```

F.3 s(CASP) output of hanoi.pl

```
?- hanoi(7,T).

Answer 1        (in [420.343] ms):

[ hanoi(7,127), move(a,b,1), move(a,c,2), move(b,c,3), move(a,b,4),
  move(c,a,5), move(c,b,6), move(a,b,7), move(a,c,8), move(b,c,9),
  move(b,a,10), move(c,a,11), move(b,c,12), move(a,b,13), move(a,c,14),
  move(b,c,15), move(a,b,16), move(c,a,17), move(c,b,18), move(a,b,19),
  move(c,a,20), move(b,c,21), move(b,a,22), move(c,a,23), move(c,b,24),
  move(a,b,25), move(a,c,26), move(b,c,27), move(a,b,28), move(c,a,29),
  move(c,b,30), move(a,b,31), move(a,c,32), move(b,c,33), move(b,a,34),
  move(c,a,35), move(b,c,36), move(a,b,37), move(a,c,38), move(b,c,39),
  move(b,a,40), move(c,a,41), move(c,b,42), move(a,b,43), move(c,a,44),
  move(b,c,45), move(b,a,46), move(c,a,47), move(b,c,48), move(a,b,49),
  move(a,c,50), move(b,c,51), move(a,b,52), move(c,a,53), move(c,b,54),
  move(a,b,55), move(a,c,56), move(b,c,57), move(b,a,58), move(c,a,59),
  move(b,c,60), move(a,b,61), move(a,c,62), move(b,c,63), move(a,b,64),
  move(c,a,65), move(c,b,66), move(a,b,67), move(c,a,68), move(b,c,69),
  move(b,a,70), move(c,a,71), move(c,b,72), move(a,b,73), move(a,c,74),
  move(b,c,75), move(a,b,76), move(c,a,77), move(c,b,78), move(a,b,79),
  move(c,a,80), move(b,c,81), move(b,a,82), move(c,a,83), move(b,c,84),
  move(a,b,85), move(a,c,86), move(b,c,87), move(b,a,88), move(c,a,89),
  move(c,b,90), move(a,b,91), move(a,c,92), move(b,c,93), move(b,a,94),
  move(c,a,95), move(c,b,96), move(a,b,97), move(a,c,98), move(b,c,99),
  move(a,b,100), move(c,a,101), move(c,b,102), move(a,b,103), move(a,c,104),
  move(b,c,105), move(b,a,106), move(c,a,107), move(b,c,108), move(a,b,109),
  move(a,c,110), move(b,c,111), move(a,b,112), move(c,a,113), move(c,b,114),
  move(a,b,115), move(c,a,116), move(b,c,117), move(b,a,118), move(c,a,119),
  move(c,b,120), move(a,b,121), move(a,c,122), move(b,c,123), move(a,b,124),
  move(c,a,125), move(c,b,126), move(a,b,127) ]

T  =  127 ?
```