instituto madrileño de estudios avanzados

www.imdea.org

Executable Logic Workshop @ UDC

September 14, 2018

J. Arias[1,2] M. Carro[1,2] E. Salazar[3] K. Marple[3] G. Gupta[3]

[1]IMDEA Software Institute, [2]Universidad Politécnica de Madrid, [3]University of Texas at Dallas

institute imdea software

madrid institute for advanced studies in software development technologies

Last Activities

- Tabling: advantages of bottom-up computation using top-down execution.
  - Termination, performance.
  - Well known, old (but still challenging)

Last Activities

- Tabling: advantages of bottom-up computation using top-down execution.
  - Termination, performance.
  - Well known, old (but still challenging)
- Add constraints: enhance expressiveness, termination properties, also speed.
  - First implementation with *full* call / answer entailment checks.
  - Modular: constraint solvers as plug-ins.
  - Improved termination results w.r.t. [Toman 1997].
  - PPDP'16, TPLP'1? (submitted).

Last Activities

- Tabling: advantages of bottom-up computation using top-down execution.
  - Termination, performance.
  - Well known, old (but still challenging)
- Add constraints: enhance expressiveness, termination properties, also speed.
  - First implementation with *full* call / answer entailment checks.
  - Modular: constraint solvers as plug-ins.
  - Improved termination results w.r.t. [Toman 1997].
  - PPDP'16, TPLP'1? (submitted).
- Top-down execution of ASP with constraints — evolution of s(ASP).
  - Can execute non-grounded CASP programs.
  - Constraint system with arbitrary (e.g., unbound) variable domains.
  - Partial models: relevance.
  - Almost ASP semantics: *unsafe* goals allowed.
  - ICLP'18.

Tabling

- Solve issues with loops in SLD resolution:

```
p(b):- p(X).
p(a).

?- p(A).
```

- Variant calls *suspend*:
  - Branch freezes.
  - Execution switches to another clause.
  - Possible results feed and resume suspended calls. Execution continues.
- Termination for programs with bounded-depth property.
- Involved implementation!

Tabling + Constraints

- Add constraints: same, plus entailment instead of variance.

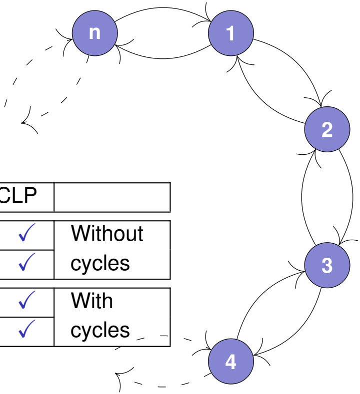
```
p(X) :- X #> Y, p(Y).
p(0).

?- p(A).
```

- More particular calls suspend.
  - `p(Y){Y < X}` more particular than `p(X)`.
  - Suspension, resumption driven by *constraint entailment*.
- Answers checked for entailment: only more general answers kept.
- Less resumptions: speedup.
- Termination guaranteed for compact constraint domains ([Toman 1997])
- Also, for programs that generate compact subset of constraint domain ([Arias & Carro]).
- **Very** involved implementation.

Termination Comparison

Example: Find nodes in a weighted graph within a distance K from each other (using comparable –very similar– programs).



	Prolog	CLP	Tabling	TCLP	
Left recursion	x	x	✓	✓	Without
Right recursion	✓	✓	✓	✓	cycles
Left recursion	x	x	x	✓	With
Right recursion	x	✓	x	✓	cycles

Design: Flexibility

- Constraint solver implementations pluggable.
- In general, amounts to writing an interface file to access projection and constraint store extraction.
- Validated with several cases.

CLP($D_{\leq}$) Connection of existing constraint solver for difference constraints, written in C.

CLP(Q) and CLP(R) Constraint solvers for linear equations over rationals (CLP(Q)) and over reals (CLP(R)) ([Holzbaur 1995]).

CLP(Lat) New constraint solver over finite lattices.

Performance evaluation (Time)

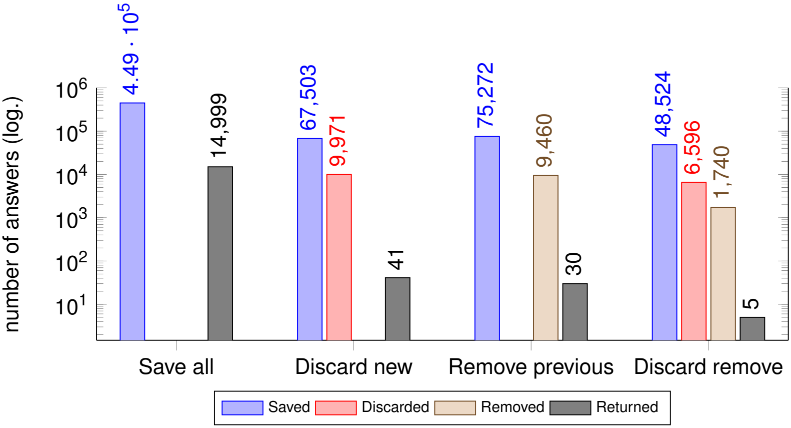
Different answer management strategies

	$\mathrm{CLP}(D_{\leq})$	Modular $\mathrm{TCLP}(D_{\leq})$
truckload(300)	40452	7268
truckload(200)	4179	2239
truckload(100)	145	259

	Save all	Discard new answer	Remove previous	Discard and remove
truckload(300)	742039	7806	7780	7268
truckload(200)	11785	2314	2354	2239
truckload(100)	300	263	263	259
step_bound(30)	–	8450	–	1469
step_bound(20)	–	6859	38107	1267
step_bound(10)	–	2846	8879	845

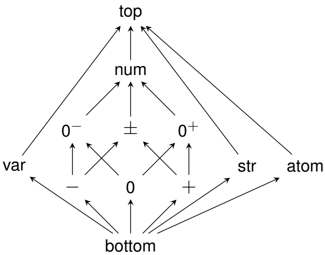
Performance Evaluation (Number of Answers)

Different answer management strategies



Performance evaluation – Tabling vs TCLP(Lat)

Simple Abstract Interpreter using sign abstract domain.



Two versions of Abstract Interpreter:

- **Tabling** uses variant tabling.
- **TCLP(Lat)** Uses entailment check to suspend. Computation saved by reusing results from previous, more general, call.

	Tabling	TLCP(Lat)
analyze(takeuchi/9)	31.44	8.09
analyze(takeuchi/7)	13.75	5.85
analyze(takeuchi/4)	2.42	3.12

Constraint ASP Without Grounding

Motivation

ASP + constraints: *grounding phase* an issue since ranges of (constrained) variables may be infinite.

- Unbound range: `X #> 0` in $\mathbb{N}$
- Bound range, but dense domain: `X #> 0` $\land$ `X #< 1` in $\mathbb{Q}$

Current CASP systems (e.g., EZCSP [Balduccini and Lierler 2017] and clingo[DL/LP] [Janhunen et al. 2017]) limit (some of):

- Admissible constraint domains.
- Where constraints can appear.
- Type / number of models computed.

s(CASP): Main Points

- Adds constraints to s(ASP) [Marple et al. 2017], a top-down execution model that avoids the *grounding* phase.
- Is implemented with a goal-driven interpreter written in Ciao Prolog.
  - The execution of a program starts with a *query*.
  - Each answer provides the *mgu* of a successful derivation, its justification, and the *relevant* (partial) stable model.
- Retains variables and constraints during the execution and in the model.

`https://ciao-lang.org`

`https://gitlab.software.imdea.org/joaquin.arias/sCASP`

Background: s(ASP) [Marple et al. 2017]

- s(ASP) computes constructive negation: `not p(X)` returns in `X` the values for which `p(X)` fails.
  - Negated atoms are resolved against dual rules synthesized applying De Morgan's laws to Clark's completion of the original program.
- The construction of dual rules need two new operators:
  - Disequality (negation of the unification).
  - Universal quantifier (in the body of the clauses).
- To ensure that global constraints and consistency rules hold, NMR-check rules are synthesized and executed.
- The resulting program is executed by the s(ASP) interpreter which:
  - Carries around explicitly unification and disequality constraints.
  - Detects and handles different types of loops.

Background: Compilation of the Dual (Example)

- The *Dual* of a predicate *P* is another predicate that succeeds for the cases where *P* would have failed:
  1. Clark completion.
  2. Negation of $\leftrightarrow$.
  3. Rearrangement of atoms.
  4. Introduction of $\neq$ and $\forall$.

Given the predicate:

```
p(0).
p(X) :- q(X), not t(X,Y).
```

Its dual rules are:

```
not p(X) :- not p1(X), not p2(X).
not p1(X) :- X\=0.
not p2(X) :-
    forall(Y, not p2_(X,Y)).
not p2_(X,Y) :- not q(X).
not p2_(X,Y) :- q(X), t(X,Y).
```

Background: Compilation of the Dual (Example)

- The *Dual* of a predicate *P* is another predicate that succeeds for the cases where *P* would have failed:
  1. Clark completion.
  2. Negation of $\leftrightarrow$.
  3. Rearrangement of atoms.
  4. Introduction of $\neq$ and $\forall$.

Given the predicate:

```
p(0).
p(X) :- q(X), not t(X,Y).
```

Its dual rules are:

Constructive disequality

```
not p(X) :- not p1(X), not p2(X).
not p1(X) :- X\=0.
not p2(X) :-
    forall(Y, not p2_(X,Y)).
not p2_(X,Y) :- not q(X).
not p2_(X,Y) :- q(X), t(X,Y).
```

Background: Compilation of the Dual (Example)

- The *Dual* of a predicate *P* is another predicate that succeeds for the cases where *P* would have failed:
  1. Clark completion.
  2. Negation of $\leftrightarrow$.
  3. Rearrangement of atoms.
  4. Introduction of $\neq$ and $\forall$.

Given the predicate:

```
p(0).
p(X) :- q(X), not t(X,Y).
```

Its dual rules are:

Forall

```
1(X), not p2(X).
not p      X \= 0.
not p2(X)  -
   forall(Y, not p2_(X,Y)).
not p2_(X,Y) :- not q(X).
not p2_(X,Y) :- q(X), t(X,Y).
```

Background: Compilation of the Dual (Example)

- The *Dual* of a predicate *P* is another predicate that succeeds for the cases where *P* would have failed:
  1. Clark completion.
  2. Negation of $\leftrightarrow$.
  3. Rearrangement of atoms.
  4. Introduction of $\neq$ and $\forall$.

Given the predicate:

```
p(0).
p(X) :- q(X), not t(X,Y).
```

Its dual rules are:

```
not p(X) :- not p1(X), not p2(X).
not p1(X) :- X \= 0.
not p2(X) :-
    forall(Y, not p2_(X,Y)).
not p2_(X,Y) :- not q(X).
not p2_(X,Y) :- q(X), t(X,Y).
```

For efficiency

Background: Compilation of the NMR-check (Example)

Given the consistency rule:

$$\forall \vec{x}\ (p(\vec{x}) \leftarrow \exists \vec{y}\ B \wedge \neg p(\vec{x}))$$

Any model should satisfy:

$$\forall \vec{x} \forall \vec{y}\ (\neg B \vee p(\vec{x})))$$

Background: Compilation of the NMR-check (Example)

Given the consistency rule:

$$\forall \vec{x}\ (p(\vec{x}) \leftarrow \exists \vec{y}\ B \land \neg p(\vec{x}))$$

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

Any model should satisfy:

$$\forall \vec{x} \forall \vec{y}\ (\neg B \lor p(\vec{x})))$$

```
chk_1(X) :- forall(Y, not chk_1_(X,Y)).
not chk_1_(X,Y) :- not q(X,Y).
...
not chk_1_(X,Y) :- q(X,Y), ..., p(X).
```

Background: Compilation of the NMR-check (Example)

Given the consistency rule:

$$\forall \vec{x}\ (p(\vec{x}) \leftarrow \exists \vec{y}\ B \wedge \neg p(\vec{x}))$$

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

Any model should satisfy:

$$\forall \vec{x} \forall \vec{y}\ (\neg B \vee p(\vec{x})))$$

```
chk_1(X) :- forall(Y, not chk_1_(X,Y)).
not chk_1_(X,Y) :- not q(X,Y).
...
not chk_1_(X,Y) :- q(X,Y), ..., p(X).
```

$$\perp \leftarrow \exists \vec{x}\ \neg r(\vec{x})$$

```
:- not r(X).
```

$$\forall \vec{x}\ r(\vec{x})$$

```
chk_2 :- forall(X, r(X)).
```

Background: Compilation of the NMR-check (Example)

Given the consistency rule:

$$\forall \vec{x}\ (p(\vec{x}) \leftarrow \exists \vec{y}\ B \wedge \neg p(\vec{x}))$$

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

Any model should satisfy:

$$\forall \vec{x} \forall \vec{y}\ (\neg B \vee p(\vec{x})))$$

```
chk_1(X) :- forall(Y, not chk_1_(X,Y)).
not chk_1_(X,Y) :- not q(X,Y).
...
not chk_1_(X,Y) :- q(X,Y), ..., p(X).
```

$$\perp \leftarrow \exists \vec{x}\ \neg r(\vec{x})$$

```
:- not r(X).
```

$$\forall \vec{x}\ r(\vec{x})$$

```
chk_2 :- forall(X, r(X)).
```

To ensure that each NMR-check rule is satisfied, the compiler adds the rule:

```
nmr_check :- forall(X,chk_1(X)), chk_2, ...
```

Background: $\neq$ and `forall(X,Goal)`

Explanation delayed.

Background: Handling Loops (Goal eventually Calling Itself)

s(ASP) interpreter checks existence of different types of loops:

Background: Handling Loops (Goal eventually Calling Itself)

s(ASP) interpreter checks existence of different types of loops:

- Odd loop over negation: recursion with an odd number of intervening negations: fails (avoid inconsistencies).

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

```
?- p(a).
no
```

Background: Handling Loops (Goal eventually Calling Itself)

s(ASP) interpreter checks existence of different types of loops:

- Odd loop over negation: recursion with an odd number of intervening negations: fails (avoid inconsistencies).

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

```
q(a).
```

```
?- p(a).
no
```

- Even loop over negation: Id. with even (non zero) number of intervening negations. It generates multiple models.

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

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

```
?- p(a).
{p(a),not q(a)}
```

Background: Handling Loops (Goal eventually Calling Itself)

s(ASP) interpreter checks existence of different types of loops:

- Odd loop over negation: recursion with an odd number of intervening negations: fails (avoid inconsistencies).

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

```
?- p(a).
no
```

- Even loop over negation: Id. with even (non zero) number of intervening negations. It generates multiple models.

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

```
?- p(a).
{p(a),not q(a)}
```

- Positive loops: No intervening negations. Fail if calls match (as in tabling).

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

```
?- p(a).
no
```

s(CASP): Design and Implementation

Main contributions w.r.t. s(ASP) are:

- Re-implemented interpreter: Prologs takes care of environment (e.g., the behavior of variables).

s(CASP): Design and Implementation

Main contributions w.r.t. s(ASP) are:

- Re-implemented interpreter: Prologs takes care of environment (e.g., the behavior of variables).
- (Simple) constraint solver for disequality, CLP($\neq$), using attributed variables.

s(CASP): Design and Implementation

Main contributions w.r.t. s(ASP) are:

- Re-implemented interpreter: Prologs takes care of environment (e.g., the behavior of variables).
- (Simple) constraint solver for disequality, CLP($\neq$), using attributed variables.
- Generic CLP interface and an extended compiler to plug in constraint solvers.

s(CASP): Design and Implementation

Main contributions w.r.t. s(ASP) are:

- Re-implemented interpreter: Prologs takes care of environment (e.g., the behavior of variables).
- (Simple) constraint solver for disequality, CLP($\neq$), using attributed variables.
- Generic CLP interface and an extended compiler to plug in constraint solvers.
- Design and implementation of *C-forall* — generalizes original *forall* algorithm, to support constraints in arbitrary domains.

s(CASP): Design and Implementation

Main contributions w.r.t. s(ASP) are:

- Re-implemented interpreter: Prologs takes care of environment (e.g., the behavior of variables).
- (Simple) constraint solver for disequality, CLP($\neq$), using attributed variables.
- Generic CLP interface and an extended compiler to plug in constraint solvers.
- Design and implementation of *C-forall* — generalizes original *forall* algorithm, to support constraints in arbitrary domains.

	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

Run time (ms) comparison s(CASP) vs. s(ASP).

s(CASP): The Interpreter

```
??(Query) :-
  solve(Query,[],Mid),
  solve_goal(nmr_check,Mid,Out),
  output_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),!,
  check_loops(Goal,In,Out).
solve_goal(Goal,In,Out) :-
  Goal = forall(V,FGoal),!,
  c_forall(V,FGoal,In,Out).
solve_goal(Goal,In,Out) :-
  call(Goal),
  Out = ['$success',Goal|In].
```

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

s(CASP): The Interpreter

Check consistency

```
??(Query) :-
  solve(Query,[],Mid),
  solve_goal(nmr_check,Mid,Out),
  output_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),!,
  check_loops(Goal,In,Out).
solve_goal(Goal,In,Out) :-
  Goal = forall(V,FGoal),!,
  c_forall(V,FGoal,In,Out).
solve_goal(Goal,In,Out) :-
  call(Goal),
  Out = ['$success',Goal|In].
```

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

s(CASP): The Interpreter

```
??(Query) :-
  solve(Query,[],Mid),
  solve_goal(nmr_check,Mid,Out),
  output_just_model(Out).

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

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

Call path /
derivation tree

(very abridged) Code of the s(CASP) interpreter.

s(CASP): The Interpreter

Detect and handle loops

```
??(Query) :-
  solve(Query,[],Mid),
  solve_goal(nmr_check,Mid,Out),
  output_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),!,
  check_loops(Goal,In,Out).
solve_goal(Goal,In,Out) :-
  Goal = forall(V,FGoal),!,
  c_forall(V,FGoal,In,Out).
solve_goal(Goal,In,Out) :-
  call(Goal),
  Out = ['$success',Goal|In].
```

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

```
...
check_loops(Goal,In,Out) :-
    pr_rule(Goal,Body),
    solve(Body,[Goal|In],Out).
```

s(CASP): The Interpreter

```
??(Query) :-
  solve(Query,[],Mid),
  solve_goal(nmr_check,Mid,O
  output_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),!,
  k_loops(Goal,In,Out).
solve_goal(Goal,In,Out) :-
  Goal = forall(V,FGoal),!,
  c_forall(V,FGoal,In,Out).
solve_goal(Goal,In,Out) :-
  call(Goal),
  Out = ['$success',Goal|In].
```

C-forall

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

s(CASP): The Interpreter

```
??(Query) :-
  solve(Query,[],Mid),
  solve_goal(nmr_check,Mid,Out),
  output_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),!,
  check_loops(Goal,In,Out).
solve_goal(Goal,In,Out) :-
  Goal = forall(V,FGoal),!,
  c_forall(V,FGoal,In,Out).
solve_goal(Goal,In,Out) :-
  call(Goal),
  Out = ['$success',Goal|In].
```

Figure: (Very abridged) Code of t

Builtins, CLP($\neq$), CLP($\mathbb{Q}$), ...

s(CASP): `c_forall(X,Goal)`

Check if `Goal` is true for all values in the constraint domain of `X`.

Intuition: Narrow the constraint store C_i under which `Goal` is executed by selecting an answer A_i and removing from C_i the values of `X` covered by A_i.

$A_{\mathtt{X}}$ is the projection of A onto `X`.

$A_{\overline{\mathtt{X}}}$ Id. onto the set of variables in `Goal` that are not `X`.

s(CASP): `c_forall(X,Goal)`

Check if `Goal` is true for all values in the constraint domain of `X`.

Intuition: Narrow the constraint store C_i under which `Goal` is executed by selecting an answer A_i and removing from C_i the values of `X` covered by A_i.

$A_{\mathtt{X}}$ is the projection of A onto `X`.

$A_{\overline{\mathtt{X}}}$ Id. onto the set of variables in `Goal` that are not `X`.

Algorithm (simplified):

- If `Goal` succeeds with answer A_i under C_i, there are two possibilities:
  - $A_{i.\mathtt{X}} \equiv C_{i.\mathtt{X}}$ then succeed.
  - $A_{i.\mathtt{X}} \sqsubset C_{i.\mathtt{X}}$ then re-execute `Goal` under $C_{i+1} = C_i \wedge A_{i.\overline{\mathtt{X}}} \wedge \neg A_{i.\mathtt{X}}$.
- If `Goal` fails, then fail.

Note 1: `c_forall/2` takes care of disjunctions generated by $\neg A_{i.\mathtt{X}}$
(Constraints solvers usually cannot handle them natively.)

s(CASP): `c_forall(X,Goal)`

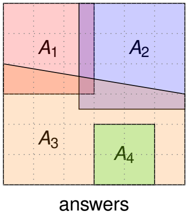


Figure: A *C-forall* evaluation that succeeds.

s(CASP): `c_forall(X,Goal)`

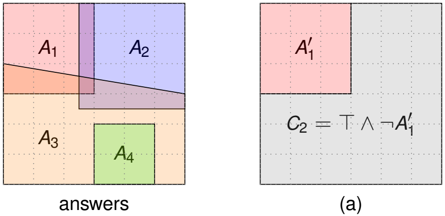


Figure: A *C-forall* evaluation that succeeds.

s(CASP): `c_forall(X,Goal)`

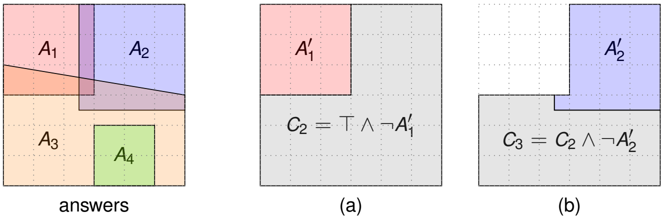


Figure: A *C-forall* evaluation that succeeds.

s(CASP): `c_forall(X,Goal)`

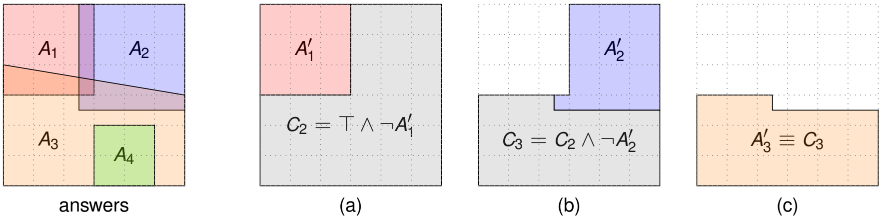


Figure: A *C-forall* evaluation that succeeds.

s(CASP): `c_forall(X,Goal)`

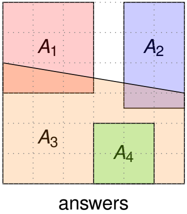


Figure: A *C-forall* evaluation that fails.

s(CASP): c_forall(X,Goal)

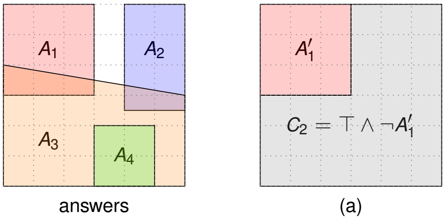


Figure: A *C-forall* evaluation that fails.

s(CASP): c_forall(X,Goal)

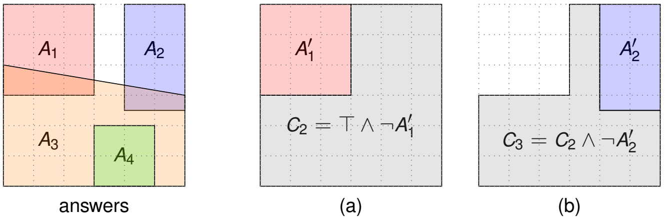


Figure: A *C-forall* evaluation that fails.

s(CASP): `c_forall(X,Goal)`

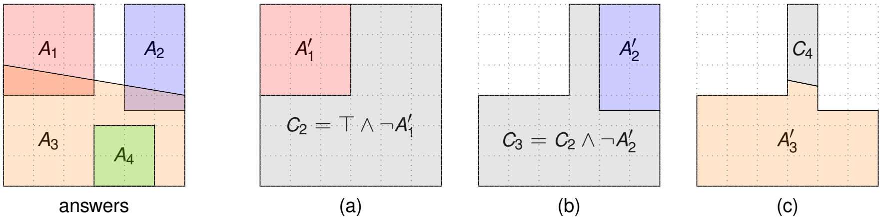
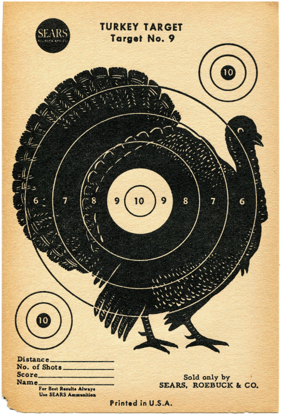


Figure: A *C-forall* evaluation that fails.

s(CASP): Yale Shooting Scenario [Janhunen et al. 2017]



- There is a turkey, a gun, and three possible actions: wait, load, shoot.
- Initially: turkey alive, gun unloaded.
- The turkey will die if we load and shoot within 35 minutes. Otherwise, the gun powder is spoiled.
- We are not allowed to shoot in the first 35 minutes.
- We want a plan to kill the turkey within 100 minutes.

s(CASP): Yale Shooting Scenario [Janhunen et al. 2017]

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

holds(0, State, []):- init(State).
holds(F_Time, F_State, [Action|As]):-
   F_Time #> 0,
   F_Time #= P_Time + Duration,
   duration(Action, Duration),
   not prohibited(Action, F_Time),
   trans(Action, P_State, F_State),
   holds(P_Time, P_State, As).
init(st(alive,unloaded,0)).

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

s(CASP) code for the Yale Shooting problem

s(CASP): Yale Shooting Scenario [Janhunen et al. 2017]

Restrictions
as constraints

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

holds(0, State, []):- init(State).
holds(F_Time, F_State, [Action|As]):-
   F_Time #> 0,
   F_Time #= P_Time + Duration,
   duration(Action, Duration),
   not prohibited(Action, F_Time),
   trans(Action, P_State, F_State),
   holds(P_Time, P_State, As).
```

```
ve,unloaded,0)).

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

s(CASP) code for the Yale Shooting problem

s(CASP): Yale Shooting Scenario [Janhunen et al. 2017]

Restrictions as constraints

Negation

Negation

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

holds(0, State, []):- init(State).
holds(F_Time, F_State, [Action|As]):-
        0,
    F_  me #= P_Time + Duration,
    duration(Action, Duration),
    not prohibited(Action, F_Time),
    trans(Action, P_State, F_State),
    holds(P_Time, P_State, As).
```

```
   ve,unloaded,0)).

trans(load, st(alive,_,_),
            st(alive,loaded,0)).
trans(wait, st(alive,Gun,P_Armed),
            st(alive,Gun,F_Armed)):-
    F_Ar        + Duration,
    du          on).
trans(shoo   st(alive,loaded,T_Armed),
             st(dead,unloaded,0)):-
    not spoiled(T_Armed).
trans(shoot, st(alive,loaded,T_Armed),
             st(alive,unloaded,0)):-
    spoiled(T_Armed).
```

s(CASP) code for the Yale Shooting problem

s(CASP): Yale Shooting Scenario [Janhunen et al. 2017]

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

holds(0, State, []):- init(State).
holds(F_Time, F_State, [Action|As]):-
   F_Time #> 0,
   F_Time #= P_Time + Duration,
   duration(Action, Duration),
   not prohibited(Action, F_Time),
   trans(Action, P_State, F_State),
   holds(P_Time, P_State, As).
init(st(alive,unloaded,0)).

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

s(CASP) code for the Yale Shooting problem

```
% holds(Time, st(Turkey,Gun,Time_Armed), Plan)
```

s(CASP): Yale Shooting Scenario [Janhunen et al. 2017]

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

holds(0, State, []):- init(State).
holds(F_Time, F_State, [Action|As]):-
    F_Time #> 0,
    F_Time #= P_Time + Duration,
    duration(Action, Duration),
    not prohibited(Action, F_Time),
    trans(Action, P_State, F_State),
    holds(P_Time, P_State, As).
init(st(alive,unloaded,0)).

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

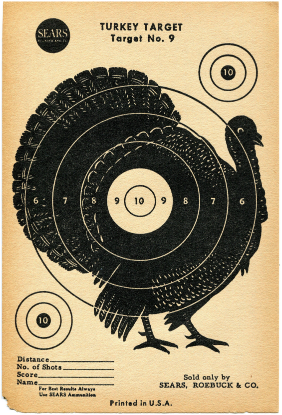
s(CASP) code for the Yale Shooting problem

```
?- Time #< 100, holds(Time, st(dead,_,_), Plan).
```

s(CASP): Yale Shooting Scenario [Janhunen et al. 2017]

```
?- ?? [Time #< 100,holds(Time,st(dead,_,_),Plan)].
Time=55, Plan=[shoot,load, load]
Time=66, Plan=[shoot,load, wait]
Time=80, Plan=[shoot,load, load, load]
Time=91, Plan=[shoot,load, load, wait]
Time=91, Plan=[shoot,load, wait, load]
Time=96, Plan=[shoot,load, shoot, wait, load]
```

s(CASP): Yale Shooting Scenario extended



Extensions:

- Time is dense $\rightarrow$ intervals have infinite # of elements.
- There is a second gun and initially only one of them is loaded.
- We cannot shoot in the first 35 minutes only if our gun is initially unloaded.

s(CASP): Yale Shooting Scenario extended

```
duration(load,25).
duration(shoot,D):- D #> 5, D #< 15/2.
duration(wait,36).
spoiled(T_Armed):- T_Armed #> 35.
prohibited(shoot,Time):-
    Time #< 35, gun(unloaded).

holds(0, State, []):- init(State).
holds(F_Time, [illegible]):-
    F_Time #> 0,
    F_Time #= P_Time + Duration,
    duration(Action, Duration),
    not prohibited(Action, F_Time),
    trans([illegible]e),
    holds(P_Time, P_Sta[illegible] As).

init(st(alive,Gun,0)) :- gun(Gun).
```

```
trans(load, st(alive,_,_),
            st(alive,loaded,0)).
[illegible],Gun,P_Armed),
[illegible]Gun,F_Armed)):-
[illegible]med + Duration,
    duration(wait,Duration).
trans(shoot, st(alive,loaded,T_Armed),
             st(dead,unloaded,0)):-
    not spoiled(T_Armed).
[illegible](alive,loaded,T_Armed),
[illegible]live,unloaded,0)):-
[illegible]rmed).

gun(loaded)  :- not s_gun(loaded).
s_gun(loaded)  :- not gun(loaded).
gun(unloaded)  :- not gun(loaded).
s_gun(unloaded)  :- not s_gun(loaded).
```

Interval in a dense domain

Restriction

Two possible worlds

Initial state

s(CASP) code for the extended and updated Yale Shooting problem.

s(CASP): Other Examples

Stream Data Reasoning: constraints and goal-directed strategy make it possible to answer queries without evaluating the complete stream database.

```
valid_stream(Pr,Data) :-
      stream(Pr,Data),
      not cancelled(Pr,Data).

cancelled(PrLo,DataLo) :-
      PrHi #> PrLo,
      stream(PrHi,DataHi),
      incompt(DataLo,DataHi).
```

s(CASP): Other Examples

Stream Data Reasoning: constraints and goal-directed strategy make it possible to answer queries without evaluating the complete stream database.

```
valid_stream(Pr,Data) :-
        stream(Pr,Data),
        not cancelled(Pr,Data).

cancelled(PrLo,DataLo) :-
        PrHi #> PrLo,
        stream(PrHi,DataHi),
        incompt(DataLo,DataHi).
```

Traveling Salesman Problem: s(CASP) encoding is more compact than CLP and constraints (over dense domains) can appear as part of the model.

```
?- travel_path(b,Length,Cycle).

{ cycle_dist(b,c,31/10), cycle_dist(c,d,A) {A #> 8, A #< 21/2},
  cycle_dist(d,a,1), cycle_dist(a,b,1) }
```

General Thoughts

- Constraint + ASP subject of a lot of research.
- Constraints work better with the notion of variables!
  - E.g., intensional description of sets.
- Traditional ASP evaluation shares some points with bottom-up.
  - I.e., both do not in principle use variables.
- But smart top-down evaluation (tabling) achieves results similar to bottom up.
  - And variables and relationships can be used.
- Can the case for ASP be similar?

Bibliography I

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.

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

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.

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

Toman, D. (1997). Memoing Evaluation for Constraint Extensions of Datalog. *Constraints*, 2(3/4):337–359.