Compilation of tight ASP programs

Carmine Dodaro, Giuseppe Mazzotta, Francesco Ricca

UNIVERSITÀ DELLA CALABRIA

CILC 2024 – Rome, Italy

Outline

Answer Set Programming

- Well-know declarative AI formalism for KR & R
- Employed in several industrial AI application
  - planning, scheduling, decision support
  - natural language understanding and more
- ASP solvers implement the stable model semantics
  - Follow a Ground&Solve approach
  - Grounding: Variable elimination
  - Solving: Propositional search for stable models

Ground&Solve Approach (1)

Example (K-Coloring Problem)

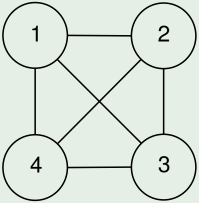
$$asgn(X, C) \leftarrow node(X),\ color(C),\ not\ nAsgn(X, C)$$
$$nAsgn(X, C) \leftarrow node(X),\ color(C),\ not\ asgn(X, C)$$
$$colored(X) \leftarrow asgn(X, C)$$
$$\leftarrow node(X), not\ colored(X)$$
$$\leftarrow asgn(X, C1),\ asgn(X, C2), C1 \neq C2$$
$$\leftarrow edge(X, Y),\ asgn(X, C),\ asgn(Y, C)$$

Ground&Solve Approach (1)

Example (K-Coloring Problem)

$$asgn(X, C) \leftarrow node(X),\ color(C),\ not\ nAsgn(X, C)$$
$$nAsgn(X, C) \leftarrow node(X),\ color(C),\ not\ asgn(X, C)$$
$$colored(X) \leftarrow asgn(X, C)$$
$$\leftarrow node(X), not\ colored(X)$$
$$\leftarrow asgn(X, C1),\ asgn(X, C2), C1 \neq C2$$
$$\leftarrow edge(X, Y),\ asgn(X, C),\ asgn(Y, C)$$

Example (Problem instance: node/1, edge/2)



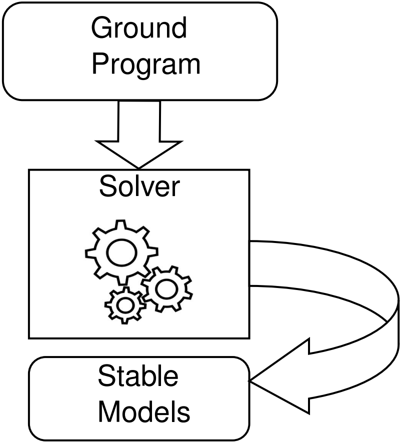
$node(1).\ node(2).\ node(3).\ node(4).$
$edge(1, 2).\ edge(1, 3).\ edge(1, 4).$
$edge(2, 1).\ edge(2, 3).\ edge(2, 4).$
$edge(3, 1).\ edge(3, 2).\ edge(3, 4).$
$edge(4, 1).\ edge(4, 2).\ edge(4, 3).$

Grounding (2)

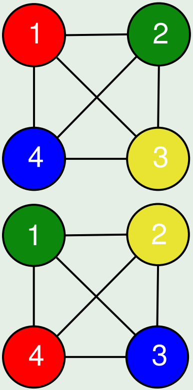
Example (4-Coloring Grounded)

$asgn(1, c_1) \leftarrow node(1),\ color(c_1),\ not\ nAsgn(1, c_1)$
$nAsgn(1, c_1) \leftarrow node(1),\ color(c_1),\ not\ asgn(1, c_1)$
...
$asgn(1, c_4) \leftarrow node(1),\ color(c_4),\ not\ nAsgn(1, c_4)$
$nAsgn(1, c_4) \leftarrow node(1),\ color(c_4),\ not\ asgn(1, c_4)$
...
$\leftarrow edge(1, 2),\ asgn(1, c_1),\ asgn(2, c_1)$
...
$\leftarrow edge(1, 2),\ asgn(1, c_4),\ asgn(2, c_4)$
$\leftarrow edge(1, 3),\ asgn(1, c_1),\ asgn(3, c_1)$
...
$\leftarrow edge(1, 3),\ asgn(1, c_4),\ asgn(3, c_4)$
...

Solving (3)



Example (Stable Models)



asgn(1,r)
asgn(2,g)
asgn(4,b)
asgn(4,y)

asgn(1,g)
asgn(2,y)
asgn(4,r)
asgn(4,b)

... ...

Motivation

What about larger instances?

Motivation

What about larger instances?

Grounding Bottlenck

Compilation-Based ASP-Solving (1)

General Idea

Translate a ground intensive sub-program into a dedicate procedure, named **propagator**, that simulates it into the solver during model computation process

- Two possible strategies
  - **Lazy propagators** when a candidate stable model is found, check whether it satisfied compiled constraints
  - **Eager Propagators** as soon as a literal is assigned, the propagator is notified to simulates the inferences of compiled constraint

Compilation-based ASP-Solving (2)

Example (Ground normal rules)

$asgn(1, c_1) \leftarrow node(1),\ color(c_1),\ not\ nAsgn(1, c_1)$
$nAsgn(1, c_1) \leftarrow node(1),\ color(c_1),\ not\ asgn(1, c_1)$
...
$asgn(1, c_4) \leftarrow node(1),\ color(c_4),\ not\ nAsgn(1, c_4)$
$nAsgn(1, c_4) \leftarrow node(1),\ color(c_4),\ not\ asgn(1, c_4)$
...
$\leftarrow edge(1, 2),\ asgn(1, c_1),\ asgn(2, c_1)$
...
$\leftarrow edge(1, 2),\ asgn(1, c_4),\ asgn(2, c_4)$
$\leftarrow edge(1, 3),\ asgn(1, c_1),\ asgn(3, c_1)$
...

Compilation-based ASP-Solving (3)

Propagator for "$\leftarrow edge(X, Y),\ asgn(X, C),\ asgn(Y, C)$"

Input : A literal l, an interpretation M
Output: A set of literals M_l
begin
    $M_l := \emptyset$;
    if $\underline{pred(l) = \text{“}asgn\text{”}}$ **and** $\underline{l \in M^+}$ **then**
        $x := l[0]$; $c := l[1]$;
        for $\underline{l_2 \in \{edge(x, y) \in M^+\}}$ **do**
            $y := l_2[2]$;
            $M_l := M_l \cup \{\overline{asgn(y, c)}\}$
        end
        $y := l[0]$; $c := l[1]$;
        for $\underline{l_2 \in \{edge(x, y) \in M^+\}}$ **do**
            $x := l_2[2]$;
            $M_l := M_l \cup \{\overline{asgn(x, c)}\}$
        end
    end
    return $\underline{M_l}$
end

Main limitations

- Compiled rules must act like constraints [MRD22]

Main limitations

- Compiled rules must act like constraints [MRD22]
  - Compiled propagators can model only deterministic inferences

Main limitations

- Compiled rules must act like constraints [MRD22]
  - Compiled propagators can model only deterministic inferences
- Atoms defined in propagators are unknown to the solver

Main limitations

- Compiled rules must act like constraints [MRD22]
  - Compiled propagators can model only deterministic inferences
- Atoms defined in propagators are unknown to the solver
  - They cannot appear in any rule in the solver
  - Solver cannot use them as branching literal

The Idea

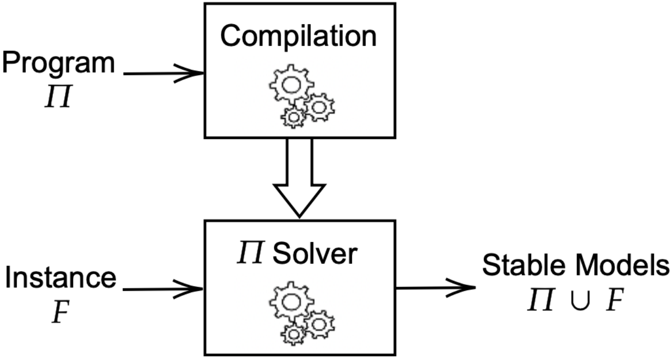
Can we overcome such limitations and compiles an entire program?

The Idea

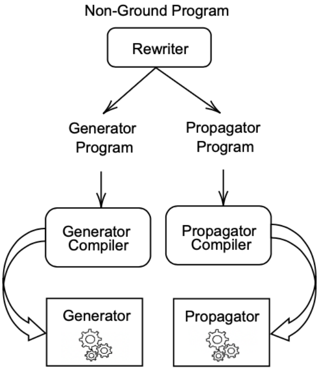
Can we overcome such limitations and compiles an entire program?

Yes, the ProASP system does

The ProASP System



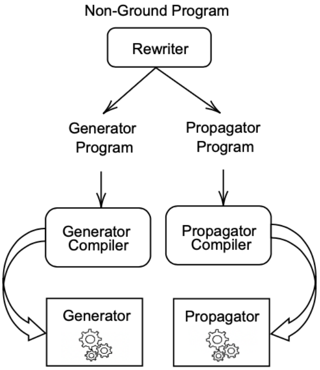
The ProASP System: Compilation Phase



Given a non-ground program Π:

1. The Rewriter generates two programs: Π^{Prop} and Π^{Gen}

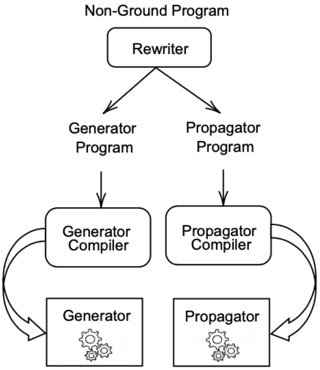
The ProASP System: Compilation Phase



Given a non-ground program Π:

1. The Rewriter generates two programs: Π^{Prop} and Π^{Gen}
   - Π^{Prop} simulates the propagation of the program Π

The ProASP System: Compilation Phase



Given a non-ground program Π:

1. The Rewriter generates two programs: Π^{Prop} and Π^{Gen}
   - Π^{Prop} simulates the propagation of the program Π
   - Π^{Gen} defines the domain of predicates in Π^{Prop}

Example: Rewriting Output

Example (K-Coloring Propagator Program: Π^{Prop})

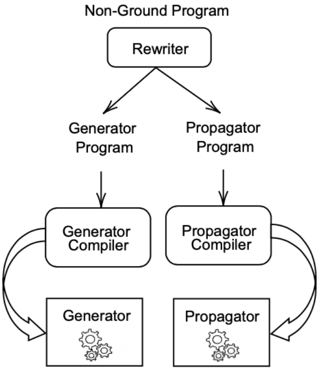
$$\begin{array}{l}
\leftarrow asgn(X,C),\ not\ node(X) \\
\leftarrow asgn(X,C),\ not\ color(C) \\
\leftarrow asgn(X,C),\ nAsgn(X,C) \\
\leftarrow node(X),\ color(C),\ not\ nAsgn(X,C),\ not\ asgn(X,C) \\
\leftarrow nAsgn(X,C),\ not\ node(X) \\
\leftarrow nAsgn(X,C),\ not\ color(C) \\
\leftarrow nAsgn(X,C),\ asgn(X,C) \\
\leftarrow node(X),\ color(C),\ not\ asgn(X,C),\ not\ nAsgn(X,C) \\
\leftarrow edge(X,Y), asgn(X,C), asgn(Y,C) \\
colored(X) \leftarrow asgn(X,C) \\
\leftarrow node(X), not\ colored(X) \\
\leftarrow asgn(X,C1),\ asgn(X,C2), C1 \neq C2
\end{array}$$

Example: Rewriting Output

Example (K-Coloring Generator Program: Π^{Gen})

$$asgn(X, C) \leftarrow node(X),\ color(C),\ not\ nAsgn(X, C)$$
$$nAsgn(X, C) \leftarrow node(X),\ color(C),\ not\ asgn(X, C)$$
$$colored(X) \leftarrow asgn(X, C)$$

The ProASP System: Compilation Phase



Given a non-ground program Π:

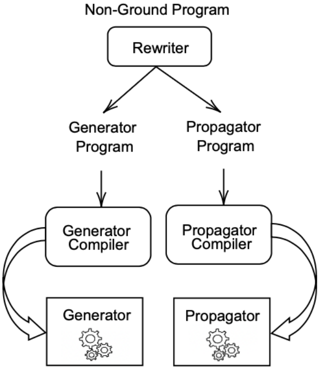
1. The Rewriter generates two programs: Π^{Prop} and Π^{Gen}
   - Π^{Prop} simulates the propagation of Π
   - Π^{Gen} defines the domain of predicates in Π^{Gen}
2. Π^{Gen} is compiled into custom bottom-up evaluation procedures

Example: Compiled Generator Module

```
Input  : set of facts F, set of atoms B
Output: set of atoms M
begin
    M := ∅;
    T1 := {node(X) ∈ B ∪ F};
    for l1 ∈ T1 do
        x := l1[0]
        T2 := {color(C) ∈ B ∪ F};
        for l2 ∈ T2 do
            c := l2[0]
            if nAsgn(x, c) ∉ F then
                M :=
                  M ∪ {asgn(x, c)}
            end
        end
    end
    return M
end
```

```
Input  : set of facts F, set of atoms B
Output: set of atoms M
begin
    M := ∅;
    T1 := {node(X) ∈ B ∪ F};
    for l1 ∈ T1 do
        x := l1[0]
        T2 := {color(C) ∈ B ∪ F};
        for l2 ∈ T2 do
            c := l2[0]
            if asgn(x, c) ∉ F then
                M :=
                  M ∪ {nAsgn(x, c)};
            end
        end
    end
    return M
end
```

The ProASP System: Compilation Phase



Given a non-ground program Π:

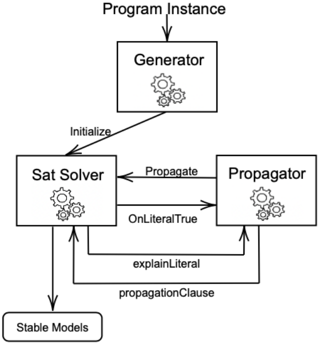
1. The Rewriter generates two programs: Π^{Prop} and Π^{Gen}
   - Π^{Prop} simulates the propagation of Π
   - Π^{Gen} defines the domain of predicates in Π^{Prop}
2. Π^{Gen} is compiled into custom bottom-up evaluation procedures
3. Π^{Prop} is compiled into custom propagators

Example: Compiled Propagator Module

Input : A literal l, an interpretation M
Output: A set of literals M_l
begin
    $M_l := \emptyset$;
    if $\underline{pred(l) = \text{“}asgn\text{”}}$ **and** $\underline{l \in M^+}$
      then
        $x := l[0]$; $c := l[1]$;
        for $\underline{l_2 \in \{edge(x, y) \in M^+\}}$
          do
            $y := l_2[2]$;
            $M_l := M_l \cup \{\overline{asgn(y, c)}\}$
        end
        $y := l[0]$; $c := l[1]$;
        for $\underline{l_2 \in \{edge(x, y) \in M^+\}}$
          do
            $x := l_2[2]$;
            $M_l := M_l \cup \{\overline{asgn(x, c)}\}$
        end
    end
    return $\underline{M_l}$
end

Input : A literal l, an interpretation M
Output: A set of literals M_l
begin
    $M_l := \emptyset$;
    if $\underline{pred(l) = \text{“}asgn\text{”}}$ **and** $\underline{l \in M^+}$
      then
        $x := l[0]$; $\underline{c := l[1]}$;
        $M_l := M_l \cup \{\overline{nAsgn(x, c)}\}$
    end
    if $\underline{pred(l) = \text{“}nAsgn\text{”}}$ **and** $\underline{l \in M^+}$
      then
        $x := l[0]$; $\underline{c := l[1]}$;
        $M_l := M_l \cup \{\overline{asgn(x, c)}\}$
    end
    return $\underline{M_l}$
end

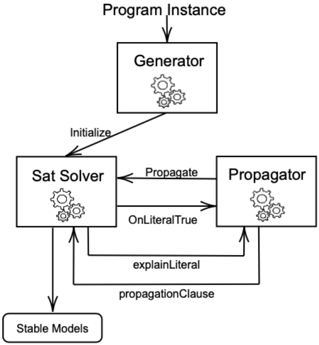
The ProASP System: Solving Phase



Given a program instance F:

1. The Generator module computes the domain of each predicate

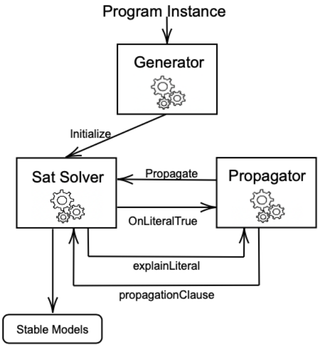
The ProASP System: Solving Phase



Given a program instance *F*:

1. The Generator module computes the domain of each predicate
2. Generated atoms are fed into the Sat Solver and CDCL starts

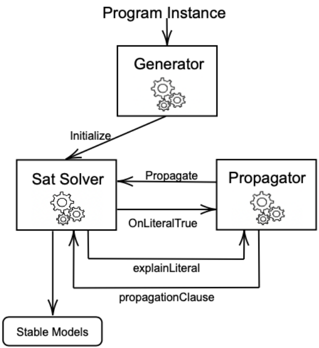
The ProASP System: Solving Phase



Given a program instance *F*:

1. The Generator module computes the domain of each predicate
2. Generated atoms are fed into the Sat Solver and CDCL starts
3. Each assigned literal activates the Propagator module, and rule inferences are propagated

The ProASP System: Solving Phase



Given a program instance *F*:

1. The Generator module computes the domain of each predicate
2. Generated atoms are fed into the Sat Solver and CDCL starts
3. Each assigned literal activates the Propagator module, and rule inferences are propagated
4. Conflicts are analyzed in the Sat Solver asking the Propagator to reconstruct propagation clauses

Experiment Goals

1. Demonstrate empirically the strengths and limitation of the PROASP system

2. Compare PROASP with existing implementation:
   (i) WASPPROP v. cb67c17 [MRD22] where propagators are nested into the solver WASP [ADLR15] and GRINGO [GKKS11] is used as grounder.
   (ii) plain version of WASP v. d87f3f0 using GRINGO as grounder;
   (iii) CLINGO [GKK+16] v. 5.6.2;
   (iv) ALPHA [Wei17] v. 0.7.0.

Experiments Setting

Considered benchmarks:

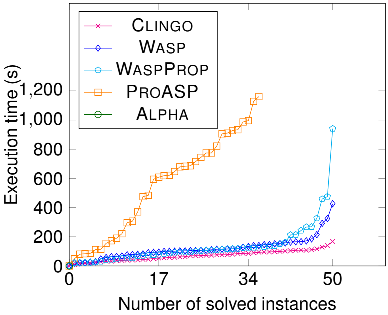
- Non Partition Removal Colouring **(NPRC)**
- Packing Problem **(P)**
- Quasi Group **(QG)**
- Stable Marriage **(SM)**
- Weight Assignment Tree **(WAT)**

All experiments were with memory and CPU time (i.e. user+system) limited of 12GB and 1200 seconds

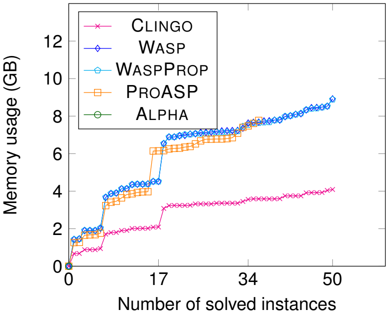
Obtained results

Benchmark	#	ProASP			WaspProp			Wasp			Clingo			Alpha		
		SO	TO	MO	SO	TO	MO	SO	TO	MO	SO	TO	MO	SO	TO	MO
(NPRC)	110	**110**	0	0	**110**	0	0	**110**	0	0	**110**	0	0	**110**	0	0
(P)	50	**23**	27	0	12	38	0	0	50	0	0	48	2	0	45	5
(QG)	100	**20**	0	80	15	0	85	12	3	85	5	0	95	5	40	55
(SM)	314	**230**	84	0	225	89	0	197	117	0	213	4	97	28	286	0
(WAT)	62	36	14	12	**50**	0	12	**50**	0	12	**50**	0	12	0	62	0

WAT: Time and Memory consumption

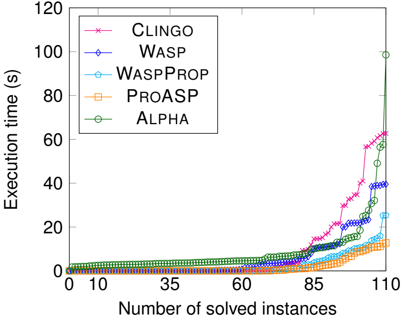


(a) (WAT) – Solving time.

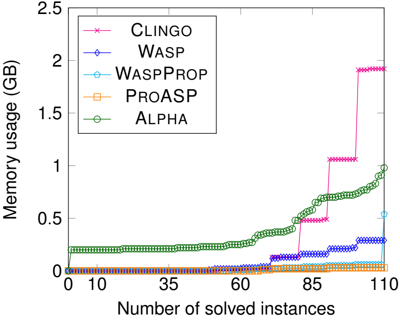


(b) (WAT) – Memory usage.

NPRC: Time and Memory consumption

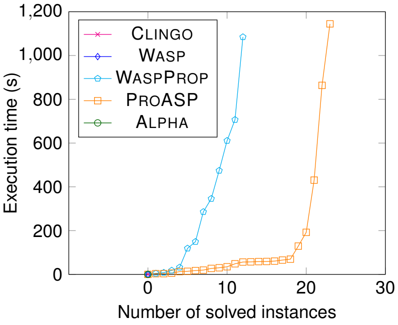


(a) (NPRC) – Solving time.

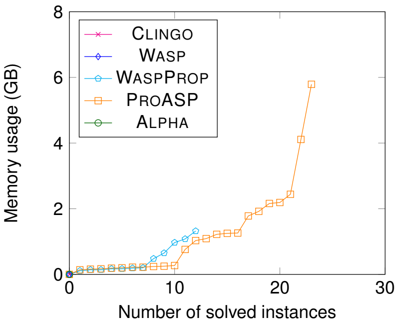


(b) (NPRC) – Memory usage.

P: Time and Memory consumption

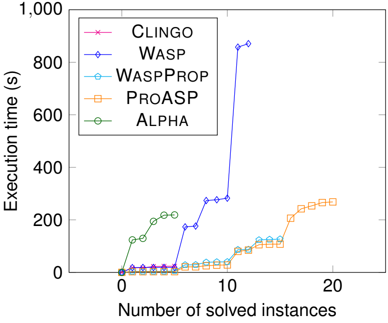


(a) (P) – Solving time.

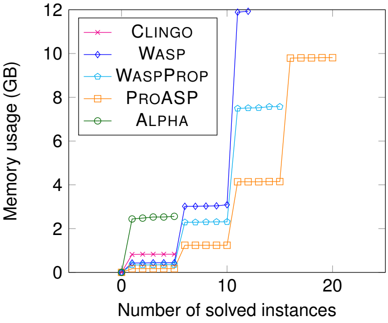


(b) (P) – Memory usage.

QG: Time and Memory consumption

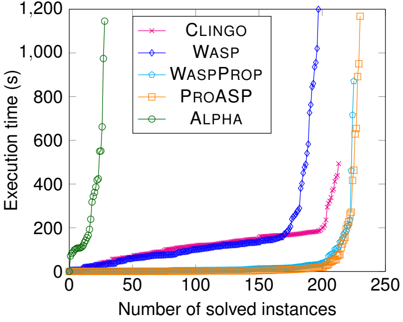


(a) (QG) – Solving time.

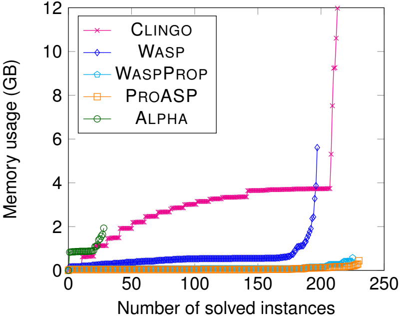


(b) (QG) – Memory usage.

SM: Time and Memory consumption



(a) (SM) – Solving time.



(b) (SM) – Memory usage.

Conclusion & Future works

1. **PROASP: Grounding-less Compilation-based system**
   - Pushed compilation boundaries beyond constraints
   - Non-ground tight programs are compiled into ad-hoc solver
     - Generated solvers extends GLUCOSE with custom propagators
   - Very effective on grounding-intensive domains
2. **Next directions**
   - Support the entire ASP-Core 2 standard
   - Enhancing PROASP by means of:
     - Compilation of support propagation
     - Lazy generation of derived symbols

Acknowledgments

Thanks for your attention!

Questions?

References

[ADLR15] Mario Alviano, Carmine Dodaro, Nicola Leone, and Francesco Ricca. Advances in WASP. In LPNMR, volume 9345 of Lecture Notes in Computer Science, pages 40–54. Springer, 2015.

[GKK+16] Martin Gebser, Roland Kaminski, Benjamin Kaufmann, Max Ostrowski, Torsten Schaub, and Philipp Wanko. Theory solving made easy with clingo 5. In ICLP (Technical Communications), volume 52 of OASICS, pages 2:1–2:15. Schloss Dagstuhl, 2016.

[GKKS11] Martin Gebser, Roland Kaminski, Arne König, and Torsten Schaub. Advances in gringo series 3. In LPNMR, volume 6645 of Lecture Notes in Computer Science, pages 345–351. Springer, 2011.

[MRD22] Giuseppe Mazzotta, Francesco Ricca, and Carmine Dodaro. Compilation of aggregates in ASP systems. In AAAI, pages 5834–5841. AAAI Press, 2022.

[Wei17] Antonius Weinzierl. Blending lazy-grounding and CDNL search for answer-set solving. In LPNMR, volume 10377 of Lecture Notes in Computer Science, pages 191–204. Springer, 2017.