A User's Guide to `gringo`, `clasp`, `clingo`, and `iclingo` *

Martin Gebser Roland Kaminski Benjamin Kaufmann
Max Ostrowski Torsten Schaub Sven Thiele **

November 9, 2008

Abstract

This document provides an introduction to the Answer Set Programming (ASP) tools `gringo`, `clasp`, `clingo`, and `iclingo`, developed at the University of Potsdam. The first tool, `gringo`, is a *grounder* capable of translating logic programs provided by users into equivalent propositional logic programs. The answer sets of such programs can be computed by `clasp`, which is a *solver*. The third tool, `clingo`, integrates the functionalities of `gringo` and `clasp`, thus, acting as a *monolithic* solver for user programs. Finally, `iclingo` extends `clingo` by an *incremental* mode that incorporates both grounding and solving. For one, this document aims at enabling ASP novices to make use of the aforementioned tools. For another, it provides a reference of their features that ASP adepts might be tempted to exploit.

*Tools `gringo`, `clasp`, `clingo`, and `iclingo` are available at [61].

**{`gebser,kaminski,kaufmann,ostrowsk,torsten,sthiele`}`@cs.uni-potsdam.de`

Contents

List of Figures

Listings

1 Introduction

The "Potsdam Answer Set Solving Collection" (Potassco) [61] by now gathers a variety of tools for Answer Set Programming. Among them, we find grounder `gringo`, solver `clasp`, and combinations thereof within integrated systems `clingo` and `iclingo`. All these tools are written in `C++` and published under GNU General Public License(s) [39]. Source packages as well as precompiled binaries for Linux and Windows are available at [61]. Note that there currently are two source packages: one containing `clasp`, and another one grouping `gringo`, `clingo`, and `iclingo`. For building one of the tools from sources, please download the most recent source package and consult the included `README` or `INSTALL` text file, respectively. Please make sure that the platform to build on has the required software installed. If you nonetheless encounter problems in the building process, please do not hesitate to contact the authors of this guide.

After downloading (and possibly building) a tool, one can check whether everything works fine by invoking the tool with flag `--version` (to get version information) or with flag `--help` (to see the available command line options). For instance, assuming that a binary called `gringo` is in the path (similarly, with the other tools), the following command line calls should be responded by `gringo`:

```
gringo --version
gringo --help
```

If grounder `gringo`, solver `clasp`, as well as integrated systems `clingo` and `iclingo` are all available, one usually provides the filenames of input text files to either `gringo`, `clingo`, or `iclingo`, while the output of `gringo` is typically piped into `clasp`. Thus, the standard invocation schemes are as follows:

```
gringo  [ options | filenames ] | clasp [ options | number ]
clingo  [ options | filenames | number ]
iclingo [ options | filenames | number ]
```

Note that a numerical argument provided to either `clasp`, `clingo`, or `iclingo` determines the maximum number of answer sets to be computed, where `0` stands for "compute all answer sets." By default, only one answer set is computed (if it exists).

This guide introduces the fundamentals of using `gringo`, `clasp`, `clingo`, and `iclingo`. In particular, it tries to enable the reader to benefit from them by significantly reducing the "time to solution" on difficult problems. The outline is as follows. In Section 2, we describe the basics of Answer Set Programming, and we formally introduce the syntax and semantics of logic programs. Section 3 details restrictedness notions, important when dealing with logic programs containing first-order variables. The probably most important part for a user, Section 4, is dedicated to the input languages of our tools, where the joint input language of `gringo` and `clingo` claims the main share (later on, it is extended by `iclingo`). For illustrating the application of our tools, three well-known example problems are solved in Section 5. Practical aspects are also in the focus of Section 6 and 7, where we elaborate and give some hints on the available command line options as well as input-related errors and warnings that may be reported. Finally, we conclude with some remarks on future work in Section 8.

For readers familiar with `lparse` [68] (a grounder that constitutes the traditional front end of solver `smodels` [66]), Appendix A lists the most prominent differences to our tools. Otherwise, `gringo`, `clingo`, and `iclingo` should accept most inputs recognized by `lparse`, while the input of solver `clasp` can also be generated by

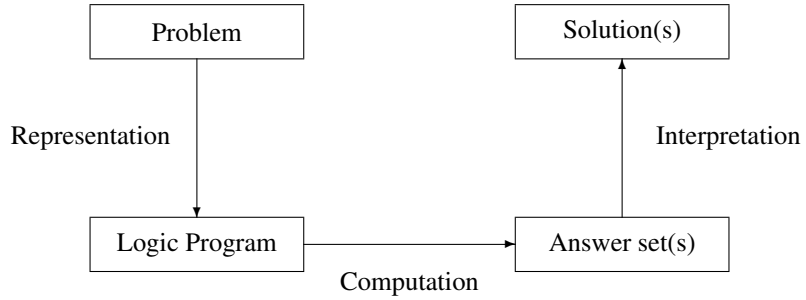

Figure 1: Declarative Problem Solving in ASP.

`lparse` instead of `gringo`. Throughout this guide, we will provide quite a number of examples. Many of them can actually be run, and instructions how to accomplish this (or sometimes meta-remarks) are provided in margin boxes, where an occurrence of "\" usually means that a text line broken for space reasons is actually continuous. After all these preliminaries, it is time to start our guided tour through Potassco [61]. We sincerely hope that you will find it enjoyable and helpful!

2 Background

In Section 2.1, we give a brief introduction to Answer Set Programming, which constitutes the basic framework for the tools we describe here. This framework deals with logic programs, whose syntax and semantics are formally introduced in Section 2.2 and 2.3, respectively. We invite the reader to merely skim or even skip these two sections upon the first reading, and to rather look them up later on if formal details are requested. However, we want to stress that the syntax we use for logic programs admits (uninterpreted) function symbols with non-zero arity, whose full support by our tools is rather exceptional and thus innovative.

2.1 Answer Set Programming

Answer Set Programming (ASP) [2, 5, 34, 45, 50, 56] emerged in the late 1990s as a declarative paradigm for modeling and solving search problems. As illustrated in Figure 1, the basic approach of ASP is to represent a problem as a logic program Π such that particular Herbrand models of Π, called answer sets, correspond to problem solutions. In contrast to traditional logic programming languages, e.g., Prolog [58], ASP programs are sets (not lists) of rules, and computations are oriented towards models (not proofs). Let us illustrate this on an example.

Example 2.1. Consider a logic program comprising the following facts:

```
1 bird(tux).     penguin(tux).
2 bird(tweety).  chicken(tweety).
```

These facts express that constants `tux` and `tweety` stand for birds. Furthermore, `tux` is a penguin, and `tweety` is a chicken. Note that the *unique names assumption* applies, that is, `tux` $\neq$ `tweety` holds. Besides the two constants, the language of the

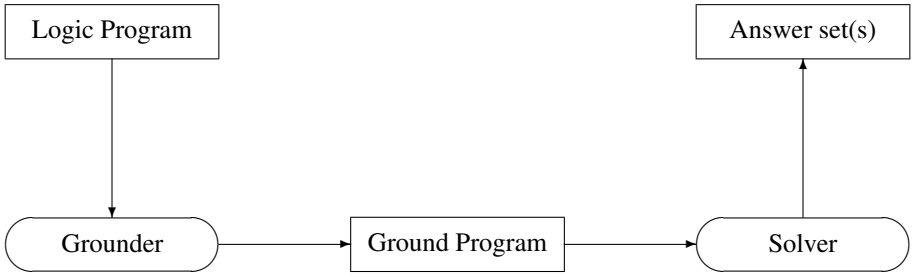

Figure 2: Basic Architecture of ASP Systems.

above facts contains predicates `bird`/1, `penguin`/1, and `chicken`/1 (each having arity 1). We next augment the above facts with the following rules:

```
3      flies(X) :- bird(X), not neg_flies(X).
4  neg_flies(X) :- bird(X), not     flies(X).
5  neg_flies(X) :- penguin(X).
```

Instead of `neg_flies(X)`, one could write `-flies(X)` to make explicit that instances (obtained for `X`) amount to the classical (or strong) negation of `flies(X)`.

Informally, such rules correspond to implications where the left hand side of connective "`:-`" (called head) is derived if all premises on the right hand side (their conjunction called body) hold. Connective "`not`" stands for *default negation*, and uppercase letter `X` is the name of a first-order variable. Variables are local to rules, that is, a variable name refers to different variables in distinct rules. Finally, every variable is universally quantified and thus applies to all terms in the language of a logic program.

The rule in Line 3 formalizes that any instance of `X` that is a bird `flies`, as long as `neg_flies` does not hold. The converse relationship between `neg_flies` and `flies` is expressed in Line 4. At this point, observe that, though syntactically correct, the above program "makes no sense" in Prolog because a query

```
?- flies(tweety).
```

can be answered neither positively nor negatively. In ASP, however, the rules in Line 3 and 4 admit multiple belief states, depending on whether `flies` or `neg_flies` is assumed for a bird. In addition, the rule in Line 5 requires `neg_flies` to hold for every penguin. Combining this knowledge with the facts in Line 1 and 2 makes us derive `neg_flies(tux)`, and it provides us with two alternatives for `tweety`: either `flies(tweety)` or `neg_flies(tweety)` holds. These alternatives cannot jointly hold because, under such an assumption, neither of them is derived, thus, violating the justification principle of ASP (cf. Section 2.3). Without already going into formal details, we conclude that the above program has two answer sets: one according to which `tweety` flies and one where `tweety` does not fly. □

The computation of answer sets usually consists of two phases. First, a *grounder* substitutes the variables in a logic program Π by variable-free terms, resulting in a propositional program $\mathbf{\Pi}$. Second, the answer sets of $\mathbf{\Pi}$ are computed by a *solver*. This prototypical architecture is visualized in Figure 2. Of course, a ground program $\mathbf{\Pi}$ must be finite and equivalent to input program Π. To this end, grounders like `lparse` [68], the one embedded into `dlv` [43], and `gringo`[1] [33] impose certain restrictions on Π (cf. Section 3.1). An obtained ground program $\mathbf{\Pi}$ can be piped into a solver, such as `assat` [47], `clasp`[1] [29], `cmodels` [37], `nomore++` [1], `smodels` [66], or

smodels$_{cc}$ [71], in order to compute answer sets of **Π**, matching the ones of Π. All of the aforementioned solvers deal with `lparse`'s output format, a numerical representation of **Π**, while `dlv` couples its grounding and its solving component directly via an internal interface, an approach also realized by `clingo`[1]. Finally, `iclingo`[1] [28] integrates and interleaves grounding and solving within incremental computations.

The next example illustrates the computation of answer sets for the logic program given in Example 2.1.

Example 2.2. Reconsidering the facts in Line 1 and 2 of Example 2.1, we identify two constants: `tux` and `tweety`. They can be substituted for variables `X` in the rules in Line 3–5. In fact, Line 1–5 of Example 2.1 are a shorthand for the ground instantiation:

```
1 bird(tux).     penguin(tux).
2 bird(tweety).  chicken(tweety).

3 flies(tux)         :- bird(tux),    not neg_flies(tux).
4 flies(tweety)      :- bird(tweety), not neg_flies(tweety).
5 neg_flies(tux)     :- bird(tux),    not     flies(tux).
6 neg_flies(tweety)  :- bird(tweety), not     flies(tweety).
7 neg_flies(tux)     :- penguin(tux).
8 neg_flies(tweety)  :- penguin(tweety).
```

Note that the facts in Line 1 and 2 are unaltered from Example 2.1, while the ground rules in Line 3–8 are obtained by instantiating variables `X`. We have that the answer sets of the above ground program match those of the original input program in Example 2.1.

In practice, grounders try to avoid producing the full ground instantiation of an input program by applying answer-set preserving simplifications. In particular, facts can (recursively) be eliminated. Thus, `gringo` computes the following ground program:

The ground program in text format is obtained via call:
```
gringo -t \
examples/bird.lp \
examples/fly.lp
```

```
1 bird(tux).     penguin(tux).
2 bird(tweety).  chicken(tweety).

      flies(tweety)  :- not neg_flies(tweety).
  neg_flies(tweety)  :- not     flies(tweety).
  neg_flies(tux).
```

Observe that `neg_flies(tux)` has become a fact because `penguin(tux)` is known. Similarly, `bird(tux)` and `bird(tweety)` have been removed from bodies of rules, making use of the fact that "*true* & something" is equivalent to "something." Applying the complementary principle that "*false* & something" evaluates to "*false*," the rule with `not neg_flies(tux)` in the body (Line 3) has been dropped completely. After performing these simplifications, the ground program computed by `gringo` is still equivalent to the input program in Example 2.1, viz., there is one answer set such that `tweety` flies and one where `tweety` does not fly. In fact, running `clasp` yields:

The two answer sets are computed by piping the output of `gringo` into `clasp`:
```
gringo \
examples/bird.lp \
examples/fly.lp | \
clasp -n 0
```
Note that `gringo` is called without option `-t`, while option `-n 0` makes `clasp` compute all answer sets (rather than just one, as done by default). Alternatively, one may use `clingo` to compute the answer sets. To this end, invoke:
```
clingo -n 0 \
examples/bird.lp \
examples/fly.lp
```

```
Answer: 1
bird(tux) penguin(tux) bird(tweety) chicken(tweety) \
neg_flies(tux) flies(tweety)
Answer: 2
bird(tux) penguin(tux) bird(tweety) chicken(tweety) \
neg_flies(tux) neg_flies(tweety)
```

[1]Described in this guide.

Note that the order of the answer sets is incidental and not obliged to the order of rules and bodies in the input, as the semantics of ASP programs is purely declarative. □

We have seen how an ASP system computes answer sets of a search problem represented by a set of facts (Line 1 and 2 in Example 2.1) and a set of schematic rules with first-order variables (Line 3–5 in Example 2.1). This separation of a problem into a knowledge part, called *encoding*, and a data part, called *instance*, is not a coincidence but a common methodology in ASP [50, 56, 65], sometimes called *uniform* definition. In fact, the possibility to describe problems in a uniform way is a major advantage of ASP, as it promotes simplicity and flexibility in knowledge representation. Together with the availability of efficient reasoning engines, this makes ASP an attractive and powerful paradigm for declarative problem solving. By explaining and illustrating the functionalities of grounder `gringo`, solver `clasp`, and their bondings in `clingo` and `iclingo`, this guide aims at enabling the reader to make (better) use of ASP.

2.2 Syntax of Logic Programs

This section gives a brief account of the formal syntax of logic programs, needed for defining their semantics. Consult, e.g., [48] for a detailed introduction. The language of a logic program is composed from sets

- $\mathcal{P} = \{p_1/i_1, \ldots, p_m/i_m\}$ of *predicates* (with arities $i_1, \ldots, i_m$),
- $\mathcal{F} = \{f_1/j_1, \ldots, f_n/j_n\}$ of *functions* (with arities $j_1, \ldots, j_n$), and
- $\mathcal{V} = \{V_1, V_2, V_3, \ldots\}$ of *variables*

along with connectives ("`:-`," "`,`" "`not`," etc.) and punctuation symbols ("`(`," "`)`," "`.`," etc.). A function f/j is called a *constant* if $j = 0$, that is, if f has no arguments. The set $\mathcal{T}$ of *terms* is the smallest set containing $\mathcal{V}$ and all expressions $f(t_1, \ldots, t_k)$ such that f/k is a function from $\mathcal{F}$ and $t_1, \ldots, t_k$ are terms in $\mathcal{T}$. Note that every variable and every constant is an elementary term, while functions with non-zero arity induce compound terms. By $ground(\mathcal{T})$, we denote the set of all variable-free terms in $\mathcal{T}$. Note that $ground(\mathcal{T})$ is infinite as soon as $\mathcal{F}$ contains at least one constant and one function with non-zero arity. The set $\mathcal{A}$ of *atoms* consists of $\top$ ("*true*"), $\bot$ ("*false*"), and all expressions $p(t_1, \ldots, t_k)$ such that p/k is a predicate from $\mathcal{P}$ and $t_1, \ldots, t_k$ are terms in $\mathcal{T}$. As with terms, $ground(\mathcal{A})$ denotes the set of all variable-free atoms in $\mathcal{A}$.

A *rule* r is an expression of the form $Head$ `:-` $Body$`.`, where $Head$ and $Body$ are composed from atoms in $\mathcal{A}$ (other than $\bot$ and $\top$), punctuation symbols, and connectives other than "`:-`." The intuitive reading of a rule is that $Head$ follows from $Body$, and either $Body$ or $Head$ may be omitted, in which case the rule is called a *fact* or an *integrity constraint*, respectively. A fact is written in the form $Head$`.` and understood as $Head$ `:-` $\top$`.`, and an integrity constraint `:-` $Body$`.` is a shorthand for $\bot$ `:-` $Body$`.` By $var(r)$, we denote the set of all variables in rule r, and r is called *ground* if $var(r) = \emptyset$. A *ground substitution* is a mapping $\sigma : var(r) \rightarrow ground(\mathcal{T})$, and $r\sigma$ is the ground rule obtained by replacing every occurrence of a variable $V \in var(r)$ with $\sigma(V)$. By $ground(r)$, we denote the set of ground rules $r\sigma$ obtained from r by applying all possible ground substitutions σ. A *logic program* Π is a set of rules, and the *ground instantiation* of Π is $ground(\Pi) = \bigcup_{r \in \Pi} ground(r)$. By convention, we assume that $\mathcal{P}$ and $\mathcal{F}$ are the sets of all predicates and functions, respectively, that occur in Π. Note that $ground(\Pi)$ is infinite if there is at least one rule $r \in \Pi$ such that $var(r) \neq \emptyset$ and if $\mathcal{F}$ contains at least one constant and one function with non-zero arity.

For a well-known class of logic programs, called *normal*, rules are of the form:

$$A_0 \texttt{ :- } A_1\texttt{, } \ldots\texttt{, } A_m\texttt{, not } A_{m+1}\texttt{, } \ldots\texttt{, not } A_n\texttt{.} \tag{1}$$

For $0 \leq k \leq n$, every A_k is an atom, that is, the head is an atom and the body is a (possibly empty) conjunction of atoms that may be preceded by "`not`." As mentioned in Section 2.1, the order of elements in the body is not essential, so that they can be regrouped to match the form in (1). For a rule r as in (1), we define $\mathit{head}(r) = A_0$, $\mathit{body}^+(r) = \{A_1, \ldots, A_m\}$, and $\mathit{body}^-(r) = \{A_{m+1}, \ldots, A_n\}$. If $r = A_0$`.` is a fact, then $\mathit{body}^+(r) = \mathit{body}^-(r) = \emptyset$. The particular structure of normal programs admits a rather intuitive characterization of their semantics in the next section.

Example 2.3. The language of the normal program in Example 2.1 includes predicates $\mathcal{P} = \{\texttt{bird}/1, \texttt{penguin}/1, \texttt{chicken}/1, \texttt{flies}/1, \texttt{neg\_flies}/1\}$ and functions $\mathcal{F} = \{\texttt{tux}/0, \texttt{tweety}/0\}$. As the arity of the latter is 0, terms `tux` and `tweety` are constants. By substituting them for variables, we obtain the ground instantiation given in Example 2.2. For instance, the ground rules

```
3  flies(tux)    :- bird(tux),    not neg_flies(tux).
4  flies(tweety) :- bird(tweety), not neg_flies(tweety).
```

are obtained from:

```
flies(X) :- bird(X), not neg_flies(X).
```

Observe that the three occurrences of variable name `X` stand for a single variable, so that substitutions $\{\texttt{X} \mapsto \texttt{tux}\}$ and $\{\texttt{X} \mapsto \texttt{tweety}\}$ induce the above ground rules. Notably, sets $\mathcal{P}$ and $\mathcal{F}$ of predicates and functions need not be disjoint because atoms and terms are also distinguished by the context they occur in, and a predicate or function symbol may be used with different arities. For instance, we could add a rule

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

over predicates `flies`/0 and `flies`/1. In fact, such rules are not unusual to express that there is some object with certain properties, while it does not matter which object it is. Finally, note that an infinite ground instantiation would result from adding rule

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

in view of function `parent`/1 with non-zero arity. □

2.3 Semantics of Logic Programs

The semantics of logic programs is given by answer sets [36], also called stable models [35]. In view of the rich input language of `gringo` (cf. Section 4.1), we below recall a definition of answer sets for propositional theories [24] and explain the semantics of logic programs by translation into propositional logic. We will also provide an alternative direct definition of answer sets for the class of normal programs.

We consider propositional formulas composed from atoms, $\bot$, and connectives $\wedge$, $\vee$, and $\rightarrow$. Furthermore, $\top$ and $\neg F$ are used as shorthands for $(\bot \rightarrow \bot)$ and $(F \rightarrow \bot)$, respectively. The *reduct* F^X of a propositional formula F relative to a set X of atoms is defined recursively as follows:

- $F^X = \bot$ if $X \not\models F$,
- $F^X = F$ if $F \in X$, and
- $F^X = (G^X \circ H^X)$ if $X \models F$ and $F = (G \circ H)$ for $\circ \in \{\wedge, \vee, \rightarrow\}$.[2]

[2] $\models$ is the standard satisfaction relation of propositional logic between interpretations X and formulas F.

Essentially, the reduct relative to X is obtained by replacing all maximal unsatisfied subformulas of F with $\bot$. As a matter fact, this implies that any maximal negative subformula of the form $(G \rightarrow \bot)$ is replaced by either $\bot$ (if $X \models G$) or $(\bot \rightarrow \bot) = \top$ (if $X \not\models G$). Furthermore, any atom occurring in F^X belongs to X.

A propositional theory Φ is a set of propositional formulas, and the reduct of Φ relative to a set X of atoms is $\Phi^X = \{F^X \mid F \in \Phi\}$. Furthermore, X is a *model* of Φ if $X \models F$ for every $F \in \Phi$. Finally, X is an *answer set* of Φ if X is a $\subseteq$-minimal model of Φ^X, that is, if X is a model of Φ such that no $Y \subset X$ is a model of Φ^X. The latter condition uses the fact that X is a model of Φ^X iff X is a model of Φ.[3] Also note that, if X is an answer set of Φ, it is the unique $\subseteq$-minimal model of Φ^X because all atoms occurring in Φ^X belong to X (which excludes incomparable $\subseteq$-minimal models). This must not be confused with uniqueness of an answer set of Φ. In fact, Φ may have zero, one, or multiple answer sets X (being $\subseteq$-minimal models of distinct reducts Φ^X). In a sense, the $\subseteq$-minimality of an answer set X as a model of Φ^X stipulates the atoms in X to be justified by Φ (or necessarily true) under the assumption of X.

For a logic program Π, answer sets of Π can now be explained by translation to a propositional theory $\Phi[\Pi]$. To this end, let

- $\Phi[\Pi] = \bigcup_{r \in ground(\Pi)} \phi[r]$,
- $\phi[Head \texttt{ :- } Body\texttt{ .}] = (\phi[Body] \rightarrow \psi[Head])$,
- $\phi[A] = \psi[A] = A$ if $A \in \mathcal{A}$,
- $\phi[\top] = \top$,[4]
- $\phi[\texttt{not } F] = \neg\phi[F]$, and
- $\phi[G\texttt{ , }H] = (\phi[G] \wedge \phi[H])$.

The translation ϕ of a rule primarily consists of straightforward syntactic conversions by replacing "`:-`," "`,`" and "`not`" with $\rightarrow$, $\wedge$, and $\neg$, respectively. However, it also contains a separate translation ψ for heads of rules, which coincides with ϕ on atoms by simply keeping them unchanged. In Section 4.1.10, we will customize ψ in order to reflect the "choice semantics" for aggregates in heads of rules [25, 66]. By virtue of the above translation, we can simply define the answer sets of a logic program Π to be the answer sets of $\Phi[\Pi]$. We have thus specified answer sets for normal programs.

In order to provide more intuitions, we now reproduce a direct definition of answer sets for normal programs. This definition builds on the fact that the reduct for a normal program is a set of Horn clauses, for which the $\subseteq$-minimal model (if it exists) can be constructed systematically in a bottom-up manner [14]. To this end, for a normal program Π and a set $X \subseteq ground(\mathcal{A})$, we define:

- $T^0_{\Pi,X} = \emptyset$ and
- $T^{i+1}_{\Pi,X} = \{head(r) \mid r \in ground(\Pi), body^+(r) \subseteq T^i_{\Pi,X}, body^-(r) \cap X = \emptyset\}$.

It is not hard to check that operator $T_{\Pi,X}$ is monotonic, as it adds head atoms of ground rules to its previous result if the positive body atoms have been derived, while negative bodies are merely evaluated w.r.t. X. Using this operator, we can equivalently define $X \subseteq ground(\mathcal{A})$ as an answer set of a normal program Π if $X = \bigcup_{0 \leq i} T^i_{\Pi,X}$. This equilibrium requirement exhibits two fundamental aspects of answer set semantics:

[3] If X is not a model of Φ, then there is some $F \in \Phi$ such that $X \not\models F$, which in turn implies $\bot \in \Phi^X$.

[4] Recall that a fact $Head\texttt{ .}$ is shorthand for $Head\texttt{ :- }\top\texttt{ .}$

1. Negative conditions are first evaluated w.r.t. an answer set candidate.

2. The candidate must be justified by the result of the preceding evaluation.

Provided that X is a model of Π (that is, of $\Phi[\Pi]$), the second item can be understood in the sense that all atoms of X must have a proof from normal program Π w.r.t. X.

Example 2.4. The normal program in Example 2.1 (or its ground instantiation given in Example 2.2) translates into the following propositional theory:

```
1  bird(tux)      penguin(tux)
2  bird(tweety)   chicken(tweety)

3  bird(tux)    ∧¬neg_flies(tux)    → flies(tux)
4  bird(tweety) ∧¬neg_flies(tweety) → flies(tweety)
5  bird(tux)    ∧¬flies(tux)        → neg_flies(tux)
6  bird(tweety) ∧¬flies(tweety)     → neg_flies(tweety)
7                   penguin(tux)    → neg_flies(tux)
8                   penguin(tweety) → neg_flies(tweety)
```

Let $X = \{$`bird(tux)`, `penguin(tux)`, `bird(tweety)`, `chicken(tweety)`, `neg_flies(tux)`, `flies(tweety)`$\}$. Relative to X, we obtain the reduct:

```
1  bird(tux)      penguin(tux)
2  bird(tweety)   chicken(tweety)

3                 ⊥ → ⊥
4  bird(tweety) ∧⊤ → flies(tweety)
5  bird(tux)    ∧⊤ → neg_flies(tux)
6                 ⊥ → ⊥
7     penguin(tux) → neg_flies(tux)
8                 ⊥ → ⊥
```

We have that X is a model of the reduct (and the original program) because the reduct does not contain formula $\bot$ as an element. In addition, observe that no proper subset of X is a model of the reduct. From this, we conclude that X is an answer set.

The latter can also be verified by determining $\bigcup_{0 \leq i} T^i_{\Pi,X}$ (for Π as in Example 2.1):

i	$T^i_{\Pi,X}$
0	—
1	`bird(tux)`, `penguin(tux)`, `bird(tweety)`, `chicken(tweety)`
2	`bird(tux)`, `penguin(tux)`, `bird(tweety)`, `chicken(tweety)`, `neg_flies(tux)`, `flies(tweety)`
>2	*do.*

For $i = 1$, we simply derive the head atoms of facts. They allow us to derive the remaining atoms of X, viz., `neg_flies(tux)` and `flies(tweety)`, in step $i = 2$. In fact, two rules produce `neg_flies(tux)`, while

```
flies(tweety) :- bird(tweety), not neg_flies(tweety).
```

is the unique rule with head `flies(tweety)` that "fires." No further rules apply in step $i = 3$, so that the set of atoms derived for $i = 2$ equals $\bigcup_{0 \leq i} T^i_{\Pi,X}$. We have thus reproduced X by bottom-up derivation, which again shows that X is an answer set. □

We have above seen two definitions of answer sets that start from an answer set candidate $X \subseteq ground(\mathcal{A})$, which is then verified in some way. Such definitions must not be confused with the computation of answer sets, as (explicitly) enumerating all subsets of $ground(\mathcal{A})$ is infeasible in practice. Rather, the computational schemes of solvers are similar to the "Davis-Putnam-Logemann-Loveland" procedure (DPLL) [11, 12] or to "Conflict-Driven Clause Learning" (CDCL) [51, 54, 55].

3 Restrictedness Notions

In view of function symbols with non-zero arity, we may be confronted with logic programs over an infinite Herbrand base. In order to maintain decidability of reasoning tasks, it is thus important to identify language fragments for which finite equivalent ground programs are guaranteed to exist. Level-restricted logic programs constitute such a fragment, where finiteness is manifested in the requirement that any variable in a rule must be bound to a finite set of ground terms via a predicate not subject to positive recursion through that rule. The notion of level-restrictedness is complemented by stratification, which by disallowing negative recursion among predicates describes a class of logic programs having unique answer sets. The formal definition of level-restricted programs, which can be grounded by `gringo` (stand-alone or embedded in `clingo` and `iclingo`), is provided in Section 3.1. Section 3.2 introduces stratified programs and the related concept of domain predicates, which can serve particular purposes during grounding. Both sections may be skimmed or even skipped upon the first reading, and rather be looked up later on to find out details.

3.1 Level-Restricted Logic Programs

The main task of a grounder is to substitute the variables in a logic program Π by terms such that the result is a finite equivalent ground program $\mathbf{\Pi}$.[5] Of course, a necessary condition for this is that Π possesses only finitely many finite answer sets. Unfortunately, such a property is undecidable in general [10]. Instead of semantic properties, grounders do thus impose rather simple syntactic conditions to guarantee the existence of finite equivalent ground programs (which are then also computed). For instance, the `dlv` system [43] requires programs to be *safe* for establishing finiteness in the absence of functions with non-zero arity and under limited arithmetic. Grounder `lparse` [68] deals with *ω-restricted* programs [69], which are more restrictive than safe programs but guarantee finiteness also in the presence of functions with non-zero arity. Finally, `gringo` [33] requires programs to be *λ-restricted*, a property more general than ω-restrictedness but likewise applicable to programs over the same languages. We below reproduce the concept of λ-restrictedness, called level-restrictedness here.

The idea underlying level-restrictedness is that the structure of a logic program Π must be such that, for each rule $r \in \Pi$, the set of potentially applicable ground instances of r is known in advance. This is the case if, for each variable $V \in var(r)$, we find an atom A in the body of r with $V \in var(A)$ (where $var(A)$ is that set of variables occurring in A) such that the potentially derivable ground instances of A are limited (see below). In fact, with such an atom A at hand, the set of ground terms to which V needs to be instantiated is known a priori, and no further ground terms need to be considered. The sketched approach is justified by the $\subseteq$-minimality of answer sets in the sense of

[5] Π and $\mathbf{\Pi}$ are equivalent if they have the same answer sets.

Section 2.3, as it allows us to restrict the attention to rules whose bodies are derivable w.r.t. some answer set. Beyond outputting only "relevant rules," grounders can apply answer-set preserving simplifications, some of which are discussed in Section 3.2.

We next provide a more precise account of level-restrictedness, where we start by considering normal programs Π. For a rule $r \in \Pi$ and $V \in var(r)$, we let $bind(V)$ contain all elements of $body^+(r)$ that may be used to limit V to a certain subset of $ground(\mathcal{T})$. Although some exceptions will be explained later on, it is convenient to assume that $bind(V)$ consists of all atoms $A \in body^+(r)$ such that $V \in var(A)$. We then define Π as *level-restricted* if there is some level mapping $\lambda : \mathcal{P} \to \mathbb{N}$ such that,[6] for every rule $r \in \Pi$ and each $V \in var(r)$, there is some $A \in bind(V)$ such that $\lambda(p/i) > \lambda(q/j)$ for predicate p/i of $head(r)$ and predicate q/j of A. Note that an appropriate λ (if it exists) is easy to determine, and `gringo` exploits it to schedule the order in which rules will be instantiated. In fact, if rules are processed in the order of levels of their head predicates, ground terms to be substituted for variables can be restricted on the basis of atoms identified as (un)derivable before.

Example 3.1. Consider the following logic program:

```
1  zig(0) :- not zag(0).  zig(1) :- not zag(1).
2  zag(0) :- not zig(0).  zag(1) :- not zig(1).
3  zigzag(X,Y) :- zig(X), zag(Y).
4  zagzig(Y,X) :- zigzag(X,Y).
```

The set $\mathcal{P}$ of predicates contains `zig`/1, `zag`/1, `zigzag`/2, and `zagzig`/2. The rules r with head predicates `zig`/1 or `zag`/1 in Line 1 and 2 are ground ($var(r) = \emptyset$). Hence, the level-restrictedness condition is trivially satisfied for these rules, and we may map their head predicates by λ as follows: $\texttt{zig}/1 \mapsto 0$ and $\texttt{zag}/1 \mapsto 0$. For the rule in Line 3, we have to respect that $\lambda(\texttt{zigzag}/2) > \lambda(\texttt{zig}/1) = \lambda(\texttt{zag}/1) = 0$. We can thus choose $\texttt{zigzag}/2 \mapsto 1$. Also note that `zig(X)` and `zag(Y)` are the only atoms in $bind(\texttt{X})$ and $bind(\texttt{Y})$, respectively. Finally, we let $\texttt{zagzig}/2 \mapsto 2$ in view of the rule in Line 4, where `zigzag(X,Y)` belongs to $bind(\texttt{X})$ and $bind(\texttt{Y})$. The total mapping $\lambda = \{\texttt{zig}/1 \mapsto 0, \texttt{zag}/1 \mapsto 0, \texttt{zigzag}/2 \mapsto 1, \texttt{zagzig}/2 \mapsto 2\}$ witnesses that the above program is level-restricted.

Note that the given mapping λ would no longer be appropriate if we add rule

```
5  zigzag(X,Y) :- zagzig(Y,X).
```

In fact, the program consisting of all rules in Line 1–5 is not level-restricted because Line 4 and 5 require $\lambda(\texttt{zagzig}/2) > \lambda(\texttt{zigzag}/2)$ and $\lambda(\texttt{zigzag}/2) > \lambda(\texttt{zagzig}/2)$, respectively. Clearly, there cannot be any level mapping λ jointly satisfying these two conditions. However, replacing Line 4 with

```
4  zagzig(Y,X) :- zigzag(X,Y), zig(X), zag(Y).
```

would restore level-restrictedness because `zig(X)` and `zag(Y)` belong to $bind(\texttt{X})$ and $bind(\texttt{Y})$, respectively. Instead of the previous λ, the following level mapping could then be used: $\{\texttt{zig}/1 \mapsto 0, \texttt{zag}/1 \mapsto 0, \texttt{zagzig}/2 \mapsto 1, \texttt{zigzag}/2 \mapsto 2\}$.□

The fundamental property of a level-restricted program is that it admits a finite equivalent ground program, not to be confused with the (full) ground instantiation because $ground(\mathcal{T})$ may still be infinite. Furthermore, level-restrictedness provides a syntactic criterion that is not difficult to check. In fact, `gringo` performs such a

[6] Recall that $\mathcal{P}$ is the set of all predicates that occur in Π.

check and accepts an input program if it is level-restricted. Generalizations of level-restrictedness to the rich input language of `gringo` will be discussed in Section 4.

3.2 Stratified Logic Programs

Another relevant restriction is stratification (cf. [53]), as every stratified (normal) program has a unique answer. Grounder `gringo` exploits stratification in two respects: first, it fully evaluates stratified (sub)programs during grounding, and second, predicates that are subject to stratification can serve as domain predicates (see below). As with level-restrictedness, stratification can be characterized in terms of level mappings, here, witnessing the absence of recursion through "`not`." We start by introducing stratification on normal programs, and in a second step, we consider stratified subprograms.

A normal program Π is *stratified* if there is some level mapping $\xi : \mathcal{P} \to \mathbb{N}$ such that, for every rule $r \in \Pi$ and predicate p/i of $head(r)$, we have

- $\xi(p/i) \geq \max\{\xi(q/j) \mid q(t_1, \ldots, t_j) \in body^+(r)\}$ and
- $\xi(p/i) > \max\{\xi(q/j) \mid q(t_1, \ldots, t_j) \in body^-(r)\}$.

Observe that the levels of predicates in the positive body of a rule can be equal to the level of the head predicate, while the levels of predicates in the negative body must be strictly less. That is, positive recursion is allowed among predicates of the same level, but negative dependencies must obey a strict order.

Example 3.2. We construct a level mapping ξ for the predicates in the logic program:

```
1 fruit(apple).  fruit(peach).  foul(peach).
2 natural(X) :- fruit(X).
3 healthy(X) :- natural(X), not foul(X).
4 tasty(X)   :- healthy(X), not bitter(X).
```

Observe that predicate `bitter`/1 does not occur in the head on any rule, thus, we may map $\texttt{bitter}/1 \mapsto 0$. Furthermore, the facts in Line 1 and the rule in Line 2 do not have a negative body, so that their head predicates can also be mapped to the lowest level: $\texttt{fruit}/1 \mapsto 0$, $\texttt{foul}/1 \mapsto 0$, and $\texttt{natural}/1 \mapsto 0$. We still have to map predicates `healthy`/1 and `tasty`/1. As `foul`/1 occurs in the negative body of the rule in Line 3, we require $\xi(\texttt{healthy}/1) > \xi(\texttt{foul}/1) = 0$. We can thus choose $\texttt{healthy}/1 \mapsto 1$. Finally, the occurrence of `healthy`/1 in the positive body of the rule in Line 4 necessitates $\xi(\texttt{tasty}/1) \geq \xi(\texttt{healthy}/1) = 1$, realized by $\texttt{tasty}/1 \mapsto 1$. In this way, we also satisfy $\xi(\texttt{tasty}/1) > \xi(\texttt{bitter}/1) = 0$. The obtained total mapping $\xi = \{\texttt{bitter}/1 \mapsto 0, \texttt{fruit}/1 \mapsto 0, \texttt{foul}/1 \mapsto 0, \texttt{natural}/1 \mapsto 0, \texttt{healthy}/1 \mapsto 1, \texttt{tasty}/1 \mapsto 1\}$ witnesses that the above program is stratified. As mentioned before, every stratified normal program has a unique answer set X. Here, we get $X = \{\texttt{fruit(apple)}, \texttt{fruit(peach)}, \texttt{foul(peach)}, \texttt{natural(apple)}, \texttt{natural(peach)}, \texttt{healthy(apple)}, \texttt{tasty(apple)}\}$.

The program would no longer be stratified if we add rule

```
5 bitter(X) :- healthy(X), not tasty(X).
```

In fact, the conditions $\xi(\texttt{tasty}/1) > \xi(\texttt{bitter}/1)$ (because of Line 4) and $\xi(\texttt{bitter}/1) > \xi(\texttt{tasty}/1)$ (because of Line 5) cannot jointly be satisfied by any level mapping ξ. Note that the non-stratified program containing all rules in Line 1–5 has two answer sets: X and $(X \setminus \{\texttt{tasty(apple)}\}) \cup \{\texttt{bitter(apple)}\}$. □

By invoking
```
gringo -t \
examples/strat.lp
```
the reader can observe that `gringo` computes this unique answer set X and represents it in terms of facts.

After dealing with stratified programs, we consider subprograms. For a given logic program Π, some $\pi \subseteq \Pi$ is a *stratified subprogram* of Π if π is stratified and if no predicate p/i occurring in π belongs to the head of any rule in $\Pi \setminus \pi$. As a matter of fact, a stratified normal subprogram π of Π has a unique answer set Y such that $Y \subseteq X$ for any answer set X of Π [46]. A stratified subprogram π of Π is *maximal* if every stratified subprogram of Π is contained in π. Note that every logic program has a maximal stratified subprogram, which is also easy to determine. In fact, `gringo` identifies the maximal stratified subprogram π of a logic program Π, and it fully evaluates the predicates p/i not belonging to the head of any rule in $\Pi \setminus \pi$, which we call *domain predicates*. In Section 4.1.8, we will see that domain predicates can serve particular purposes during grounding.

Example 3.3. As the program in Line 1–4 of Example 3.2 is stratified, the maximal stratified subprogram consists of all rules. After adding the rule in Line 5, only the rules in Line 1–3 belong to the maximal stratified subprogram. The corresponding domain predicates are `fruit`/1, `foul`/1, `natural`/1, and `healthy`/1, and the unique answer set for them is $Y = \{$`fruit(apple)`, `fruit(peach)`, `foul(peach)`, `natural(apple)`, `natural(peach)`, `healthy(apple)`$\}$. □

One can check that the program consisting of Line 1–5 in Example 3.2 is level-restricted, but not stratified. Conversely, program

```
nat(0).  nat(s(X)) :- nat(X).
```

is stratified (take $\xi = \{\texttt{nat}/1 \mapsto 0\}$), but not level-restricted ($\lambda(\texttt{nat}/1) > \lambda(\texttt{nat}/1)$ needed because of the second rule is impossible). In fact, the program has a unique but infinite answer set. Thus, the program cannot be translated to a finite equivalent ground program, which is why `gringo` does not accept it. Rather, in the first place, a program must be level-restricted in order to ground it with `gringo`. If level-restrictedness is granted, the maximal stratified subprogram is exploited in a second stage to optimize grounding and to make use of domain predicates.

4 Input Languages

This section provides an overview of the input languages of grounder `gringo`, combined grounder and solver `clingo`, incremental grounder and solver `iclingo`, and of solver `clasp`. The joint input language of `gringo` and `clingo` is detailed in Section 4.1. It is extended by `iclingo` with a few directives described in Section 4.2. Finally, Section 4.3 is dedicated to the inputs handled by `clasp`.

4.1 Input Language of `gringo` and `clingo`

The tool `gringo` [33] is a grounder capable of translating logic programs provided by users into equivalent ground programs. The output of `gringo` can be piped into solver `clasp` [29], which then computes answer sets. System `clingo` internally couples `gringo` and `clasp`, thus, it takes care of both grounding and solving. In contrast to `gringo` outputting ground programs, `clingo` returns answer sets. Both `gringo` and `clingo` can handle level-restricted input programs (cf. Section 3.1), usually specified in one or more text files whose names are passed via the command line in an invocation of either `gringo` or `clingo`. We below provide a description of constructs belonging to the input language of `gringo` and `clingo`.

4.1.1 Normal Programs and Integrity Constraints

In the previous sections, we have already seen a number of normal programs. Now also considering integrity constraints, we get the following basic rule types:

Fact: A_0 `.`
Rule: A_0 `:-` L_1`,` $\ldots$`,` L_n `.`
Integrity Constraint: `:-` L_1`,` $\ldots$`,` L_n `.`

The head A_0 of a fact or a rule is an *atom* of the form p`(`t_1`,` $\ldots$`,` t_m`)`, where p is the name of some predicate, that is, a sequence of letters and digits starting with a lowercase letter, and any t_i is a term.[7] A *term* t starting with an uppercase letter followed by a sequence of letters and digits (e.g., `X08x15`) is a variable name, and integers are written as sequences of digits possibly preceded by "–." In addition, a term can have the same syntactic structure as an atom (e.g., `p(a,1,f(X))` can be either an atom or a term, depending on where it occurs in a rule), and functions can be nested within a term. There are also built-in constructs (cf. Section 4.1.4 and 4.1.5) having particular representations. Finally, any L_j is a *literal* of the form A or `not` A for an atom A.

In Section 2.3, we have already provided a translation ψ from the head of a rule to a propositional formula. By viewing an integrity constraint `:-` $Body$ `.` as a shorthand for $\bot$ `:-` $Body$ `.`, we can now explain the semantics of integrity constraints by adding the following case:

- $\psi[\bot] = \bot$.

Note the role of integrity constraints is to eliminate answer set candidates. In fact, given an integrity constraint `:-` $Body$ `.`, any answer set X is such that $X \not\models \phi[Body]$, or in other words, some literal L in $Body$ must be unsatisfied w.r.t. X. Elaborate examples on the usage of facts, rules, and integrity constraints will be provided in Section 5.

4.1.2 Classical Negation

In logic programs, connective `not` expresses default negation, that is, a literal `not` A is assumed to hold unless A is derived. In contrast, the classical (or strong) negation of some proposition holds if the complement of the proposition is objectively derived [36]. Classical negation, indicated by symbol "–," is permitted in front of atoms. That is, if A is an atom, then $-A$ is the complement of A. Semantically, $-A$ is simply a new atom, with the additional condition that A and $-A$ must not jointly hold. A logic program Π containing complementary atoms is thus translated to propositional theory $\Phi[\Pi \cup \{$`:-` A`,` $-A \mid A \in ground(\mathcal{A})\}]$.[8] Observe that classical negation is merely a syntactic feature that can be implemented via integrity constraints whose effect is to eliminate any answer set candidate containing complementary atoms.

Example 4.1. The following logic program reformulates the one in Example 2.1:

```
1  bird(tux).      penguin(tux).
2  bird(tweety).   chicken(tweety).

3   flies(X) :- bird(X), not -flies(X).
4  -flies(X) :- bird(X), not  flies(X).
5  -flies(X) :- penguin(X).
```

[7] An atom p without arguments is simply a sequence of letters and digits (starting with a lowercase letter). For such an atom p, parentheses after the name are skipped, e.g., `p42X` could be an atom without arguments. Also note that lowercase letters include "_," which can thus be used at any position within a predicate name.

[8] Recall that $ground(\mathcal{A})$ consists of all variable-free atoms built from predicates and functions in Π.

Logically, classical negation is reflected by (implicit) integrity constraints as follows:

```
6  :- flies(tux),     -flies(tux).
7  :- flies(tweety), -flies(tweety).
```

By invoking
`gringo -t \ examples/bird.lp \ examples/flycn.lp`
the reader can observe that `gringo` indeed produces the integrity constraint in Line 7.

The additional integrity constraints do not yet change the semantics of the program, which still has the answer sets already provided in Example 2.2 (of course, identifying `neg_flies(tux)` with `-flies(tux)` and `neg_flies(tweety)` with `-flies(tweety)`). This situation changes if we add the following fact:

```
8  flies(tux).
```

While the program from Example 2.1 still admits two answer sets, both containing `flies(tux)` and `neg_flies(tux)`, there no longer is any answer set for our new program using classical negation. In fact, answer set candidates that contain both `flies(tux)` and `-flies(tux)` violate the integrity constraint in Line 6. □

4.1.3 Disjunction

Disjunctive logic programs permit connective "`|`" between atoms in rule heads. An additional case of translation ψ from Section 2.3 reflects the semantics of disjunction:

- $\psi[G \mid H] = \psi[G] \vee \psi[H]$.

The concept of level-restrictedness (cf. Section 3.1) is extended to disjunctive programs by, for a disjunctive rule r, every predicate p/i of some atom in the head of r, and each $V \in var(r)$, requiring that there is some $A \in bind(V)$ such that $\lambda(p/i) > \lambda(q/j)$ for predicate q/j of A. Furthermore, a (maximal) stratified subprogram (cf. Section 3.2) must not contain any rule with a (non-trivial) disjunctive head.

The following rule combines the ones in Line 3 and 4 from Example 4.1:

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

In general, the use of disjunction however increases computational complexity [19]. This is why `clingo`[9] and solvers like `assat` [47], `clasp` [29], `nomore++` [1], `smodels` [66], and `smodels`$_{cc}$ [71] do not work on disjunctive programs. Rather, `claspD` [15], `cmodels` [37, 44], or `gnt` [42] needs to be used for solving a disjunctive program.[10] We thus suggest to use "choice constructs" (cf. Section 4.1.10) instead of disjunction, unless the latter is required for complexity reasons (see [20] for an implementation methodology in disjunctive ASP).

4.1.4 Built-In Arithmetic Functions

`gringo` and `clingo` support a number of arithmetic functions that are evaluated during grounding. The following symbols are used for these functions: + (addition), − (subtraction), `*` (multiplication), `/` or `div` (integer division), `mod` (modulo function), `abs` (absolute value), `~` (bitwise complement), `&` (bitwise AND), `?` (bitwise OR), and `^` (bitwise exclusive OR).

Example 4.2. The usage of arithmetic functions is illustrated by the logic program:

The unique answer set of the program, obtained after evaluating all arithmetic functions, can be inspected by invoking:
`gringo -t \ examples/arithf.lp`

[9] Run as a monolithic system performing both grounding and solving.

[10] System `dlv` [43] also deals with disjunctive programs, but it uses a different syntax than presented here.

```
1  left        (7).
2  right             (2).
3  plus        (L  +  R)  :- left(L), right(R).
4  minus       (L  -  R)  :- left(L), right(R).
5  times       (L  *  R)  :- left(L), right(R).
6  divide      (L  /  R)  :- left(L), right(R).
7  divide      (R div L)  :- left(L), right(R).
8  modulo      (L mod R)  :- left(L), right(R).
9  absolute    (abs(- R)) :-          right(R).
10 complement       (~ R)  :-          right(R).
11 and         (L  &  R)  :- left(L), right(R).
12 or          (L  ?  R)  :- left(L), right(R).
13 xor         (L  ^  R)  :- left(L), right(R).
```

Note that variables `L` and `R` are instantiated to `7` and `2`, respectively, before arithmetic evaluations. Consecutive and non-separative (e.g., before "`(`") spaces can also be dropped, while spaces around tokens `div` and `mod` are mandatory. Furthermore, the argument of function `abs` must be enclosed in parentheses. In Line 9, observe that there is a unary version of `-`, "`- R`" standing for "`0 - R`." The four bitwise functions apply to signed integers, using the complement on two of a negative integer. □

It is important to note that variables in the scope of an arithmetic function are not necessarily bound (in the sense of Section 3.1) by a corresponding atom. For instance, atom `p(X+1,X)` belongs to $bind($`X`$)$, while atom `p(X+1,Y)` does not belong to $bind($`X`$)$ because it contains `X` only in the scope of an arithmetic function.

4.1.5 Built-In Comparison Predicates

The following built-in predicates permit term comparisons within the bodies of rules and on the right-hand side of conditions (cf. Section 4.1.8): == (equal), `!=` (not equal), < (less than), <= (less than or equal), > (greater than), >= (greater than or equal).

Example 4.3. The usage of comparison predicates is illustrated by the logic program:

The unique answer set of the program is obtained via call:
`gringo -t \ examples/arithc.lp`

```
1  num(1).  num(2).
2  eq (X,Y) :- X   ==  Y, num(X), num(Y).
3  neq(X,Y) :- X   !=  Y, num(X), num(Y).
4  lt (X,Y) :- X   <   Y, num(X), num(Y).
5  leq(X,Y) :- X   <=  Y, num(X), num(Y).
6  gt (X,Y) :- X   >   Y, num(X), num(Y).
7  geq(X,Y) :- X   >=  Y, num(X), num(Y).
8  all(X,Y) :- X-1 < X+Y, num(X), num(Y).
9  non(X,Y) :- X/X > Y*Y, num(X), num(Y).
```

The last two lines hint at the fact that arithmetic functions are evaluated before comparison predicates, so that the latter actually compare integers.

All comparison predicates can also be used with constants, as in the next program:

As above, invoking:
`gringo -t \ examples/symbc.lp`
yields the unique answer set of the program in terms of facts.

```
1  sym(a).  sym(b).
2  eq (X,Y) :- X == Y, sym(X), sym(Y).
3  neq(X,Y) :- X != Y, sym(X), sym(Y).
4  lt (X,Y) :- X <  Y, sym(X), sym(Y).
5  leq(X,Y) :- X <= Y, sym(X), sym(Y).
```

```
6 gt (X,Y) :- X >  Y, sym(X), sym(Y).
7 geq(X,Y) :- X >= Y, sym(X), sym(Y).
```

Finally, note that == and != also apply to compound terms over functions with non-zero arity as well as to mixed integer and non-integer arguments.[11] □

Importantly, a built-in comparison predicate cannot bind variables, that is, it does not belong to $bind(\texttt{X})$ for any variable X. Also note that built-in predicates are not considered by level mappings ξ in the context of stratification (cf. Section 3.2), or equivalently, one may assume that built-in predicates are mapped to level 0 and all other predicates to levels greater than 0.

4.1.6 Assignments

Built-in predicate = can be used in the body of a rule (or on right-hand sides of conditions introduced in Section 4.1.8) to assign a term on its right-hand side to a variable on its left-hand side. Note that "X = t" belongs to $bind(\texttt{X})$, provided that t is variable-free or that its variables are bound by atoms other than assignments with X on the right-hand side. Cyclic assignments over otherwise unbound variables are thus excluded.

Example 4.4. The next program demonstrates how terms can be assigned to variables:

The unique answer set of the program is obtained via call:
`gringo -t \ examples/assign.lp`

```
1 num(1).  num(2).  num(3).  num(4).  num(5).
2 squares(XX,YY,Z) :-
3         XX = X*X, YY = Y*Y, Z = XX+YY, Y1 = Y+1,
4         Y1*Y1 == Z, num(X), num(Y), X < Y.
```

Line 3 contains four assignments, where the right-hand sides directly or indirectly depend on X and Y. These two variables are bound in Line 4 via atoms of predicate num/1. Also observe the different usage and role of built-in comparison predicate ==. □

4.1.7 Intervals

In Line 1 of Example 4.4, there are five facts num(k) over consecutive integers k. For a more compact representation, gringo and clingo support integer intervals of the form i..j, where i and j are integers. Such an interval represents each integer k such that $i \leq k \leq j$, and intervals are expanded during grounding.

Example 4.5. The next program makes use of integer intervals:

```
1 num(1..5).
2 top5(5..9).
3 top(9).
4 top5num(1..X-4,5..X) :- num(X-4..X), top5(1..5), top(X).
```

The facts in Line 1 and 2 are expanded as follows:

```
num(1).   num(2).   num(3).   num(4).   num(5).
top5(5).  top5(6).  top5(7).  top5(8).  top5(9).
```

By instantiating X to 9, the rule in Line 4 becomes:

```
top5num(1..5,5..9) :- num(5..9), top5(1..5), top(9).
```

[11] Such functionalities are currently not supported for the other four comparison predicates, where (after instantiation and arithmetic evaluation) both arguments must be either integers or (symbolic) constants.

It is expanded to the cross product $(1..5) \times (5..9) \times (5..9) \times (1..5)$ of intervals:

```
top5num(1,5) :- num(5), top5(1), top(9).
top5num(2,5) :- num(5), top5(1), top(9).
        ⋮
top5num(5,5) :- num(5), top5(1), top(9).
top5num(1,6) :- num(5), top5(1), top(9).
top5num(2,6) :- num(5), top5(1), top(9).
        ⋮ ⋮
top5num(5,9) :- num(5), top5(1), top(9).
top5num(1,5) :- num(6), top5(1), top(9).
top5num(2,5) :- num(6), top5(1), top(9).
        ⋮ ⋮         ⋮
top5num(5,9) :- num(9), top5(1), top(9).
top5num(1,5) :- num(5), top5(2), top(9).
top5num(2,5) :- num(5), top5(2), top(9).
        ⋮ ⋮         ⋮       ⋮
top5num(5,9) :- num(9), top5(4), top(9).
top5num(1,5) :- num(5), top5(5), top(9).
top5num(2,5) :- num(5), top5(5), top(9).
        ⋮ ⋮
top5num(5,9) :- num(5), top5(5), top(9).
top5num(1,5) :- num(6), top5(5), top(9).
top5num(2,5) :- num(6), top5(5), top(9).
        ⋮ ⋮         ⋮
top5num(5,9) :- num(9), top5(5), top(9).
```

Note that only the rules with `num(5)` and `top5(5)` in the body actually contribute to the unique answer set of the above program by deriving all atoms `top5num(`m`,`n`)` for $1 \leq m \leq 5$ and $5 \leq n \leq 9$. □

Again the unique answer set is obtained via call:

```
gringo -t \
examples/int.lp
```

As with built-in arithmetic functions, an integer interval mentioning some variable (like `X` in Line 4 of Example 4.5) cannot be used to bind the variable.

4.1.8 Conditions

Conditions allow for instantiating variables to collections of terms within a single rule. This is particularly useful for encoding conjunctions or disjunctions over arbitrarily many ground atoms as well as for the compact representation of aggregates (cf. Section 4.1.10). The symbol "`:`" is used to formulate conditions.

Example 4.6. The following program uses conditions in a rule body and in a rule head:

```
1 person(jane).  person(john).
2 day(mon). day(tues). day(wed). day(thurs). day(fri).
3 available(jane) :- not on(fri).
4 available(john) :- not on(mon), not on(wed).
5 meet :- available(X) : person(X).
6 on(X) : day(X) :- meet.
```

We are particularly interested in the rules in Line 5 and 6, instantiated as follows:

```
5 meet :- available(jane), available(john).
6 on(mon) | on(tues) | on(wed) | on(thurs) | on(fri) :- meet.
```

The conjunction in Line 5 is obtained by replacing `X` in `available(X)` with all ground terms t such that `person(`t`)` holds, namely, t = `jane` and t = `john`. Furthermore, the condition in the head of the rule in Line 6 turns into a disjunction over all ground instances of `on(X)` where `X` is substituted by some term t such that `day(`t`)` holds. That is, conditions in the body and in the head of a rule are expanded to different basic language constructs.

The reader can reproduce these ground rules by invoking:
```
gringo -t \
examples/cond.lp
```

Composite conditions can also be constructed via "`:`," as in the additional rules:

```
7 weekend(sat).  weekend(sun).
8 weekdays :- day(X) : day(X) : not weekend(X).
```

Observe that we may use the same atom, viz., `day(X)`, both on the left-hand and on the right-hand side of "`:`." Furthermore, negative literals like `not weekend(X)` can occur on both sides of a condition. Note that literals on the right-hand side of a condition are connected conjunctively, that is, all of them must hold for ground instances of an atom in front of the condition. Thus, the instantiated rule in Line 8 looks as follows:

```
8 weekdays :- day(mon), day(tues), day(wed), day(thurs), day(fri).
```

The atoms in the body of this rule follow from facts, so that the rule can be simplified to a fact `weekdays.` (as done by `gringo`). □

There are three important issues about the correct usage of conditions. First, all predicates of atoms on the right-hand side of a condition must be either domain predicates (cf. Section 3.2) or built-in, which is due to the fact that conditions are evaluated during grounding.[12] Second, any variable occurring within a condition is considered as *local*, that is, a condition cannot be used to bind variables outside itself. In turn, variables outside conditions are *global*, and each variable within an atom in front of a condition must occur on the right-hand side or be global. Third, global variables take priority over local ones, that is, they are instantiated first. As a consequence, a local variable that also occurs globally is substituted by a term before the ground instances of a condition are determined. Hence, the names of local variables must be chosen with care, making sure that they do not accidentally match the names of global variables.

4.1.9 Pooling

Symbol "`;`" allows for pooling alternative terms to be used as argument within an atom, thus, specifying rules more compactly. An atom written in the form `p(...,X;Y,...)` abbreviates two options: `p(...,X,...)` and `p(...,Y,...)`. Pooled arguments in an atom of a rule body (or on the right-hand side of a condition) are expanded to a conjunction of the options within the same body (or within the same condition), while they are expanded to multiple rules (or multiple literals connected via "`,`") when occurring in the head (or in front of a condition).

Example 4.7. The following logic program makes use of pooling:

[12] The bodies of rules in a stratified subprogram (cf. Section 3.2) may contain conditions. For a rule r and $L_0 : L_1 : \ldots : L_n$ in the body of r, let $A_0 \in \mathit{body}^-(r)$ if $L_0 =$ `not` A_0 and $L_0 \in \mathit{body}^+(r)$ otherwise, and for $1 \leq i \leq n$, assume $A_i \in \mathit{body}^-(r)$ for atom A_i such that $L_i = A_i$ or $L_i =$ `not` A_i, respectively.

```
1   sym(a).  sym(b).
2   num(1).  num(2).
3   mix(A;B,M;N) :- sym(A;B), num(M;N), not -mix(M;N,A;B).
4  -mix(M;N,A;B) :- sym(A;B), num(M;N), not  mix(A;B,M;N).
```

Let us consider instantiations of the rule in Line 3 obtained with substitution $\{\texttt{A} \mapsto \texttt{a}, \texttt{B} \mapsto \texttt{b}, \texttt{M} \mapsto \texttt{1}, \texttt{N} \mapsto \texttt{2}\}$. Note that `mix`/2 and `-mix`/2 each admit four options, corresponding to the cross product of $\{\texttt{a}, \texttt{b}\}$ substituted for `A` and `B`, respectively, together with $\{\texttt{1}, \texttt{2}\}$ substituted for `M` and `N`. While the instances obtained for `mix`/2 give rise to four rules, the instances for `-mix`/2 jointly belong to the body. The (repeated) body also contains two instances each of `sym`/1 and of `num`/1. We thus get the rules:

Simplified versions of these rules are produced via call:
`gringo -t \ examples/pool.lp`

```
mix(a,1) :- sym(a),sym(b), num(1),num(2), not -mix(1,a),
  not -mix(1,b), not -mix(2,a), not -mix(2,b).
mix(a,2) :- sym(a),sym(b), num(1),num(2), not -mix(1,a),
  not -mix(1,b), not -mix(2,a), not -mix(2,b).
mix(b,1) :- sym(a),sym(b), num(1),num(2), not -mix(1,a),
  not -mix(1,b), not -mix(2,a), not -mix(2,b).
mix(b,2) :- sym(a),sym(b), num(1),num(2), not -mix(1,a),
  not -mix(1,b), not -mix(2,a), not -mix(2,b).
```

Finally, we note that pooling is also possible with arguments of built-in predicates. □

4.1.10 Aggregates

An aggregate is an operation on a multiset of weighted literals that evaluates to an integer. In combination with arithmetic comparisons, we can extract a truth value from an aggregate's evaluation, thus, obtaining an aggregate atom. Customizing the notation in [24], we consider ground *aggregate atoms* of the form:

$$l \ \mathrm{op}\ [\, L_1{=}w_1\texttt{,}\ \ldots\texttt{,}\ L_n{=}w_n \,]\ u \tag{2}$$

where l and u are integers each of which can possibly be omitted, op is a function from multisets of integers to an integer, each L_i is a literal of the form A or `not` A for some atom $A \in \mathit{ground}(\mathcal{A})$, and each w_i is an integer. The intuitive reading of an aggregate atom is that the value returned by op applied to all weights w_i such that L_i holds should be in-between lower bound l and upper bound u (inclusively). If l or u is omitted, it means that the aggregate's value is not constrained from below or above, respectively.

For an aggregate atom Agg as in (2), we let $\mathit{atom}(\mathit{Agg}) = \{L_i \in \mathit{ground}(\mathcal{A}) \mid 1 \leq i \leq n\}$ denote the set of atoms occurring positively in Agg. Currently, `gringo` and `clingo` support aggregate operations `sum`, `max`, and `min`.[13] We can now extend translations ϕ and ψ from Section 2.3 to aggregate atoms, using the approach in [24] but applying the "choice semantics" for aggregates in heads of rules [25, 66]:[14]

1. $\phi[\mathrm{op}\,[\, L_1{=}w_1\texttt{,}\ \ldots\texttt{,}\ L_n{=}w_n \,]] = \top$,
2. $\phi[l \ \mathrm{op}\,[\, L_1{=}w_1\texttt{,}\ \ldots\texttt{,}\ L_n{=}w_n \,]\ u] =$
   $\phi[l \ \mathrm{op}\,[\, L_1{=}w_1\texttt{,}\ \ldots\texttt{,}\ L_n{=}w_n \,]] \wedge \phi[\mathrm{op}\,[\, L_1{=}w_1\texttt{,}\ \ldots\texttt{,}\ L_n{=}w_n \,]\ u]$,

[13] The additional operation `count` is a shorthand for a `sum` aggregate in which all weights are `1`.

[14] The translation ϕ we provide for `sum` aggregates assumes that there are no negative weights for literals, while the more sophisticated translation in [24] also captures negative weights. All current solvers work with positive weights only, and thus `gringo` and `lparse` use a compilation technique [66] to eliminate negative weights, which are not permitted in `lparse`'s output format [68]. As this technique may lead to counterintuitive results [24], we strongly recommend not to use negative weights within `sum` aggregates.

3. $\phi[l \texttt{ sum [} L_1\texttt{=}w_1\texttt{ , ... , }L_n\texttt{=}w_n\texttt{]} =$
$\big(\bigvee_{\{i_1,\ldots,i_j\}\subseteq\{1,\ldots,n\},l\leq w_{i_1}+\cdots+w_{i_j}}(\phi[L_{i_1}]\wedge\cdots\wedge\phi[L_{i_j}])\big)$,

4. $\phi[\texttt{sum [} L_1\texttt{=}w_1\texttt{ , ... , }L_n\texttt{=}w_n\texttt{]}\, u] =$
$\neg\big(\bigvee_{\{i_1,\ldots,i_j\}\subseteq\{1,\ldots,n\},u<w_{i_1}+\cdots+w_{i_j}}(\phi[L_{i_1}]\wedge\cdots\wedge\phi[L_{i_j}])\big)$,

5. $\phi[l \texttt{ max [} L_1\texttt{=}w_1\texttt{ , ... , }L_n\texttt{=}w_n\texttt{]} = \big(\bigvee_{i\in\{1,\ldots,n\},l\leq w_i}\phi[L_i]\big)$,

6. $\phi[\texttt{max [} L_1\texttt{=}w_1\texttt{ , ... , }L_n\texttt{=}w_n\texttt{]}\, u] = \neg\big(\bigvee_{i\in\{1,\ldots,n\},u<w_i}\phi[L_i]\big)$,

7. $\phi[l \texttt{ min [} L_1\texttt{=}w_1\texttt{ , ... , }L_n\texttt{=}w_n\texttt{]} = \neg\big(\bigvee_{i\in\{1,\ldots,n\},w_i<l}\phi[L_i]\big)$,

8. $\phi[\texttt{min [} L_1\texttt{=}w_1\texttt{ , ... , }L_n\texttt{=}w_n\texttt{]}\, u] = \big(\bigvee_{i\in\{1,\ldots,n\},w_i\leq u}\phi[L_i]\big)$, and

9. $\psi[\lhd \texttt{ op [} L_1\texttt{=}w_1\texttt{ , ... , }L_n\texttt{=}w_n\texttt{]} \rhd] = \phi[\lhd \texttt{ op [} L_1\texttt{=}w_1\texttt{ , ... , }L_n\texttt{=}w_n\texttt{]} \rhd] \wedge$
$\big(\bigwedge_{A\in atom(\lhd \texttt{ op [} L_1\texttt{=}w_1\texttt{ , ... , }L_n\texttt{=}w_n\texttt{]} \rhd)}(A\vee\neg A)\big)$.

The first item reflects that an unconstrained aggregate atom always holds, while aggregate atoms with a lower and an upper bound are split up into two conjunctively connected parts (Item 2). Item 3–8 specify these parts individually for `sum`, `max`, and `min` aggregates, respectively. Provided that only non-negative weights are used within `sum`s, all three aggregates are subject to one comparison that is monotone, meaning that by default it evaluates to *false* so that truth must be established. For `sum` and `max` aggregates, these monotone comparisons are related to lower bounds (Item 3 and 5), while an upper bound behaves monotonically for `min` (Item 8). In turn, the opposite comparison is antimonotone, that is, by default it evaluates to *true* unless too many or improper literals hold. This is the case for comparisons to upper bounds with aggregates `sum` and `max` (Item 4 and 6), while it concerns the lower bound for `min` (Item 7). Finally, translation ψ of an aggregate atom in the head of a rule consists of two parts: first, ϕ is used like in the body of a rule (in Item 9, $\lhd$ and $\rhd$ are used as "wildcards" for provided or omitted lower and upper bounds), second, a disjunction $(A \vee \neg A)$ is added for each atom A occurring positively. Even though such disjunctions are tautological, they are important under answer set semantics (cf. Section 2.3), as they permit deriving an arbitrary subset of atoms from an aggregate in the head (which explains the name "choice semantics"). Beyond this special treatment in rule heads, the truth values of aggregate atoms result from their literals as may be intuitively expected, and Item 1–8 basically do nothing but formalizing the standard meanings of aggregate operations. Note that the above translations merely define the semantics of aggregates, which does not imply that solvers unfold them in this way. In fact, `clasp` natively supports aggregates without translating them.

As regards syntactic representation, weight `1` is considered a default, so that L_i`=1` can simply be written as L_i. For instance, the following (multi)sets of (weighted) literals are the same when combined with any kind of aggregate operation and bounds:

```
[a=1, not b=1, c=2]   and
[a,   not b,   c=2].
```

Furthermore, keyword `sum` may be omitted, which in a sense makes `sum` the default aggregate operation. In fact, the following aggregate atoms are synonyms:

```
2 sum [a, not b, c=2] 3   and
2     [a, not b, c=2] 3.
```

By omitting keyword `sum`, we obtain the same notation as the one of so-called "weight constraints" [66, 68], which are actually aggregate atoms whose operation is addition.

It is important to note that the (weighted) literals within an aggregate belong to a multiset. In particular, if there are multiple occurrences L=$w_1, \ldots, L$=w_k of a literal L, in combination with `min` and `max`, it is not the same like having L=$w_1 + \cdots + w_k$. To see this, note that the program consisting of the facts:

```
2 max [a=2].  2 min [a=2].
```

has {`a`} as its unique answer set, while there is no answer set for:

```
2 max [a,a].  2 min [a,a].
```

If literals ought not to be repeated, we can use `count` instead of `sum`. Syntactically, `count` requires curly instead of square brackets, and there must not be any weights within a `count` aggregate. Regarding semantics, (l `count{`L_1`, ..., `L_n`}`u) reduces to (l `sum[`L_1`=1, ..., `L_m`=1]`u), where $\{L_1, \ldots, L_m\} = \{L_i \mid 1 \leq i \leq n\}$ is obtained by dropping repeated literals. Of course, the use of l and u is optional also with `count`. As an example, note that the next aggregate atoms express the same:

```
1 sum   [a=1, not b=1]     1   and
1 count {a,a, not b,not b} 1.
```

Keyword `count` can be omitted (like `sum`), so that the following are synonyms:

```
1 count {a, not b} 1   and
1       {a, not b} 1.
```

The last notation is similar to the one of so-called "cardinality constraints" [66, 68], which are aggregate atoms using counting as their operation.

After considering the syntax and semantics of ground aggregate atoms, we now turn our attention to non-ground aggregates. Regarding contained variables, an atom occurring in an aggregate behaves similar to an atom on the left-hand side of a condition (cf. Section 4.1.8). That is, any variable occurring within an aggregate is a priori local, and it must be bound via a variable of the same name that is global or that occurs on the right-hand side of a condition (with the atom containing the variable in front). As with local variables of conditions, global variables take priority during grounding, so that the names of local variables must be chosen with care to avoid accidental clashes. Beyond conditions (which are more or less the natural construct to use for instantiating variables within an aggregate), classical negation (cf. Section 4.1.2), built-in arithmetic functions (cf. Section 4.1.4), intervals (cf. Section 4.1.7), and pooling (cf. Section 4.1.9) can be incorporated as usual within aggregates, where intervals and pooling are expanded locally.[15] That is, an interval gives rise to multiple literals connected via "`,`" within the same aggregate. The same applies to pooling in front of a condition, while it turns into a composite condition chained by "`:`" on the right-hand side. The notions of level-restrictedness and stratification (cf. Section 3.1 and 3.2) are extended to programs with aggregates by considering the predicates of all atoms (except for those on the right-hand side of a condition) in an aggregate atom occurring as the head of a rule r,[16] and by assuming $A \in \mathit{body}^-(r)$ for all atoms A of an aggregate in the body of r. Finally, note that aggregates without bounds are also permitted on the

[15] Assignments (cf. Section 4.1.6) and (currently) also built-in comparison predicates (cf. Section 4.1.5) are permitted on the right-hand sides of conditions only.

[16] The rules in a (maximal) stratified subprogram (cf. Section 3.2) cannot have aggregate atoms in the head.

right-hand sides of assignments, but using this feature is only recommended for aggregates whose atoms belong to domain predicates because space blow-up can become a bottleneck otherwise. The following example, making exhaustive use of aggregates, nonetheless demonstrates this and other features.

Example 4.8. Consider a situation where an informatics student wants to enroll for a number of courses at the beginning of a new term. In the university calendar, eight courses are found eligible, and they are represented by the following facts:

```
1  course(1,1,5). course(1,2,5).
2  course(2,1,4). course(2,2,4).
3  course(3,1,6).                 course(3,3,6).
4  course(4,1,3).                 course(4,3,3). course(4,4,3).
5  course(5,1,4).                                course(5,4,4).
6                 course(6,2,2). course(6,3,2).
7                 course(7,2,4). course(7,3,4). course(7,4,4).
8                                course(8,3,5). course(8,4,5).
```

In an instance of `course`/3, the first argument is a number identifying one of the eight courses, and the third argument provides the course's contact hours per week. The second argument stands for a subject area: `1` corresponding to "theoretical informatics," `2` to "practical informatics," `3` to "technical informatics," and `4` to "applied informatics." For instance, atom `course(1,2,5)` expresses that course `1` accounts for `5` contact hours per week that may be credited to subject area `2` ("practical informatics"). Observe that a single course is usually eligible for multiple subject areas.

After specifying the above facts, the student starts to provide personal constraints on the courses to enroll. The first condition is that `3` to `6` courses should be enrolled:

```
9  3 { enroll(C) : course(C,S,H) } 6.
```

Instantiating the above `count` aggregate yields the following ground rule:

```
9  3 { enroll(1), enroll(2), enroll(3), enroll(4),
       enroll(5), enroll(6), enroll(7), enroll(8) } 6.
```

The full ground program is obtained by invoking:

```
gringo -t \
examples/aggr.lp
```

Observe that an instance of atom `enroll(C)` is included for each instantiation of `C` such that `course(C,S,H)` holds for some values of `S` and `H`. Duplicates resulting from distinct values for `S` are removed, thus, obtaining the above set of ground atoms.

The next constraints of the student regard the subject areas of enrolled courses:

```
10 :- [ enroll(C) : course(C,S,H) ] 10.
11 :- 2 [ not enroll(C) : course(C,2,H) ].
12 :- 6 [ enroll(C) : course(C,3,H), enroll(C) : course(C,4,H) ].
```

Each of the three integrity constraints above contains a `sum` aggregate, using default weight `1` for literals. Recalling that `sum` aggregates operate on multisets, duplicates are not removed. Thus, the integrity constraint in Line 10 is instantiated as follows:

```
10 :- [ enroll(1) = 1, enroll(1) = 1,
        enroll(2) = 1, enroll(2) = 1,
        enroll(3) = 1, enroll(3) = 1,
        enroll(4) = 1, enroll(4) = 1, enroll(4) = 1,
        enroll(5) = 1, enroll(5) = 1,
        enroll(6) = 1, enroll(6) = 1,
        enroll(7) = 1, enroll(7) = 1, enroll(7) = 1,
        enroll(8) = 1, enroll(8) = 1 ] 10.
```

Note that courses `4` and `7` count three times because they are eligible for three subject areas, viz., there are three distinct instantiations for `S` in `course(4,S,3)` and `course(7,S,4)`, respectively. Comparing the above ground instance, the meaning of the integrity constraint in Line 10 is that the number of eligible subject areas over all enrolled courses must be more than `10`. Similarly, the integrity constraint in Line 11 expresses the requirement that at most one course of subject area `2` ("practical informatics") is not enrolled, while Line 12 stipulates that the enrolled courses amount to less than six nominations of subject area `3` ("technical informatics") or `4` ("applied informatics"). Also note that, given the facts in Line 1–8, we could equivalently have used `count` rather than `sum` in Line 11, but not in Line 10 and 12.

The remaining constraints of the student deal with contact hours. To express them, we first introduce an auxiliary rule and a fact:

```
13  hours(C,H) :- course(C,S,H).
14  max_hours(20).
```

The rule in Line 13 projects instances of `course`/3 to `hours`/2, thereby, dropping courses' subject areas. This is used to not consider the same course multiple times within the following integrity constraints:

```
15  :- not M-2 [ enroll(C) : hours(C,H) = H ] M, max_hours(M).
16  :- min [ enroll(C) : hours(C,H) = H ] 2.
17  :- 6 max [ enroll(C) : hours(C,H) = H ].
```

As Line 15 shows, we may use default negation via "`not`" in front of aggregate atoms, and bounds may be specified in terms of variables. In fact, by instantiating `M` to `20`, we obtain the following ground instance of the integrity constraint in Line 15:

```
15  :- not 18 [ enroll(1) = 5, enroll(2) = 4,
                enroll(3) = 6, enroll(4) = 3,
                enroll(5) = 4, enroll(6) = 2,
                enroll(7) = 4, enroll(8) = 5 ] 20.
```

The above integrity constraint states that the `sum` of contact hours per week must lie in-between `18` and `20`. Note that the `min` and `max` aggregates in Line 16 and 17, respectively, work on the same (multi)set of weighted literals as in Line 15. While the integrity constraint in Line 16 stipulates that any course to enroll must include more than `2` contact hours, the one in Line 17 prohibits enrolling for courses of `6` or more contact hours. Of course, the last two requirements could also be formulated as follows:

```
16  :- enroll(C), hours(C,H), H <= 2.
17  :- enroll(C), hours(C,H), H >= 6.
```

Finally, the following rules illustrate the use of aggregates within assignments:

```
18  courses(N) :- N = count { enroll(C) : course(C,S,H) }.
19  hours(N)   :- N = sum [ enroll(C) : hours(C,H) = H ].
```

Note that the above aggregates have already been used in Line 9 and 15, respectively, where keywords `count` and `sum` have been omitted for convenience. These keywords can be dropped here too, and we merely include them to show the more verbose notations of `count` and `sum` aggregates. However, the usage of aggregates in the last two lines is different from before, as they now serve to assign an integer to a variable `N`. In this context, bounds are not permitted, and so none are provided in Line 18 and 19. The effect of these two lines is that the student can read off the number of courses to

enroll and the amount of contact hours per week from instances of `courses`/1 and `hours`/1 belonging to an answer set. In fact, running `clasp` shows the student that a unique collection of `5` courses to enroll satisfies all requirements: the courses `1`, `2`, `4`, `5`, and `7`, amounting to `20` contact hours per week.

Although the above program does not reflect this possibility, it should be noted that (as has been mentioned in Section 4.1.8) multiple literals may be connected via "`:`" in order to construct composite conditions within an aggregate. As before, the predicates of atoms on the right-hand side of such conditions must be either domain predicates or built-in. Furthermore, the usage of non-domain predicates within an aggregate on the right-hand side of an assignment (like `enroll`/1 in Line 18 and 19 above) is not recommended in general because the space blow-up may be significant. □

To compute the unique answer set of the program, invoke:

```
gringo \
examples/aggr.lp | \
clasp -n 0
```

or alternatively:

```
clingo -n 0 \
examples/aggr.lp
```

4.1.11 Optimization

Optimization statements extend the basic question of whether a set of atoms is an answer set to whether it is an optimal answer set. To support this reasoning mode, `gringo` and `clingo` adopt the optimization statements of `lparse` [68], indicated via keywords `maximize` and `minimize`. Syntactically, a `maximize` or `minimize` statement is similar to a fact whose head is a `count` or a `sum` aggregate (without bounds), viz., "opt`{ ... } .`" or "opt`[ ... ] .`" where opt $\in$ {`maximize`, `minimize`}. As an optimization statement does not admit a body, any (local) variable in it must also occur in an atom (over a domain or built-in predicate) on the right-hand side of a condition (cf. Section 4.1.8) within the optimization statement. In multiset notation (square brackets), weights may be provided as with `sum` aggregates. In set notation (curly brackets), duplicates of literals are removed as with `count` aggregates.

The semantics of an optimization statement is intuitive: an answer set is *optimal* if the sum of weights (using `1` for unsupplied weights) of literals that hold is maximal or minimal, as required by the statement, among all answer sets of the given program. This definition is sufficient if a single optimization statement is specified along with a logic program. Unfortunately, the syntax used by `lparse` and adopted by `gringo` and `clingo` does not provide any means to explicitly declare priorities among multiple optimization statements. A technique how to compile multiple optimization statements into a single one is discussed in [66], but it can lead to large integers (probably rather unnatural for a user to assign). Thus, rather than restricting to a single optimization statement, a sequence of them is permitted, where a statement provided later on takes priority over previous ones. That is, the last optimization statement in program is the most significant one, and previous ones are only considered if answer sets agree on the sum of weights of its literals. In this way, multiple optimization statements are arranged into a total order, which is the inverse order of occurrence. In contrast to the declarative answer set semantics, the semantics of optimization statements is thus order-dependent, requiring special care when providing more than one of them.

Example 4.9. To illustrate optimization, we consider a hotel booking situation where we want to choose one among five available hotels. The hotels are identified via numbers assigned in descending order of stars. Of course, the more stars a hotel has the more it costs per night. As an ancillary information, we know that hotel `4` is located on a main street, which is why we expect its rooms to be noisy. This knowledge is specified in Line 1–5 of the following program:

```
1 1 { hotel(1..5) } 1.
2 star(1,5).   star(2,4).   star(3,3).  star(4,3).  star(5,2).
```

```
3 cost(1,170). cost(2,140). cost(3,90). cost(4,75). cost(5,60).
4 main_street(4).
5 noisy :- hotel(X), main_street(X).
6 maximize [ hotel(X) : star(X,Y) = Y ].
7 minimize [ hotel(X) : cost(X,Y) : star(X,Z) = Y/Z ].
8 minimize { noisy }.
```

Line 6–8 contribute optimization statements in inverse order of significance, according to which we want to choose the best hotel to book. The most significant optimization statement in Line 8 states that avoiding noise is our main priority. The secondary optimization criterion in Line 7 consists of minimizing the cost per star. Finally, the third optimization statement in Line 6 specifies that we want to maximize the number of stars among hotels that are otherwise indistinguishable. The optimization statements in Line 6–8 are instantiated as follows:

The full ground program is obtained by invoking:
```
gringo -t \
examples/opt.lp
```

```
6 maximize [ hotel(1) = 5, hotel(2) = 4, hotel(3) = 3,
             hotel(4) = 3, hotel(5) = 2 ].
7 minimize [ hotel(1) = 34, hotel(2) = 35, hotel(3) = 30,
             hotel(4) = 25, hotel(5) = 30 ].
8 minimize [ noisy = 1 ].
```

If we now use `clasp` to compute an optimal answer set, we find that hotel `4` is not eligible because it implies `noisy`. Thus, hotel `3` and `5` remain as optimal w.r.t. the second most significant optimization statement in Line 7. This tie is broken via the least significant optimization statement in Line 6 because hotel `3` has one star more than hotel `5`. We thus decide to book hotel `3` offering `3` stars to cost `90` per night. □

To compute the unique optimal answer set, invoke:
```
gringo \
examples/opt.lp | \
clasp -n 0
```
or alternatively:
```
clingo -n 0 \
examples/opt.lp
```

4.1.12 Meta-Statements

After considering the language of logic programs, we now introduce features going beyond the contents of a program.

Comments. To keep records of the contents of a logic program, a logic program file may include comments. A comment until the end of a line is initiated by symbol "`%`," and a comment within one or over multiple lines is enclosed by "`%*`" and "`*%`." As an abstract example, consider:

```
logic program  %* enclosed comment *%  logic program
logic program  % comment till end of line
logic program
%*
comment over multiple lines
*%
logic program
```

Hiding Predicates. Sometimes, one may be interested only in a subset of the atoms belonging to an answer set. In order to suppress the atoms of "irrelevant" predicates from the output, the `#hide` declarative (in which `#` is optional) can be used. The meanings of the following statements are indicated via accompanying comments:

```
#hide.          % Suppress all atoms in output
 hide.          % Same as "#hide."
```

```
#hide p/3.      % Suppress all atoms of predicate p/3 in output
 hide p/3.      % Same as "#hide p/3."
#hide p(X,Y,Z). % Same as "#hide p/3."
 hide p(X,Y,Z). % Same as "#hide p/3."
```

In order to selectively include the atoms of a certain predicate in the output, one may use the `#show` declarative (in which `#` is again optional). Here are some examples:

```
#show p/3.      % Include all atoms of predicate p/3 in output
 show p/3.      % Same as "#show p/3."
#show p(X,Y,Z). % Same as "#show p/3."
 show p(X,Y,Z). % Same as "#show p/3."
```

A typical usage of `#hide` and `#show` is to hide all predicates via "`#hide.`" and to selectively re-add atoms of certain predicates `p/`n to the output via "`#show p/`n`.`"

Constant Replacement. Constants appearing in a logic program may actually be placeholders for concrete values to be provided by a user. An example of this will be given in Section 5.1. Via the `#const` declarative (`#` is optional), one may define a default value to be inserted for a constant. Such a default value can still be overridden via command line option `--const` (cf. Section 6.1). Syntactically, `#const` must be followed by an assignment having a (symbolic) constant on the left-hand side and a term on the right-hand side. Some exemplary `#const` declarations are:

```
#const x = 42.
 const y = f(g,h).
```

Domain Declarations. Usually, variable names are local to a rule, where they must be bound via appropriate atoms (cf. Section 3.1). This locality can be undermined by using `#domain` declarations (`#` is optional) that globally associate variable names to atoms. An associated atom is then simply added to the body of a rule in which such a predefined variable name occurs in. The following is a made-up example:

```
p(1,1). p(1,2).
#domain p(X,Y).
 domain p(Y,Z).
q(Z,X) :- not p(Z,X).
```

The above program is a priori not level-restricted because variables `X` and `Z` are unbound in the last rule. However, as they belong to `#domain` declarations, `gringo` and `clingo` expand the last rule to:

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

Observe that the resulting program is level-restricted (and stratified).

Compute Statements. These statements are artefacts supported for backward compatibility. Although we strongly recommend to avoid compute statements, we now describe their syntax. A compute statement is of the form "`#compute`n{ ... }`.`" (`#` and non-negative integer n are optional), where the "{ ... }" part is similar to a `count` aggregate. The meaning is that all literals contained in "{ ... }" must hold w.r.t. answer sets that are to be computed, while n specifies a number of answer sets to compute. As

`clasp`, `clingo`, and `iclingo` provide command line option `--number` (cf. Section 6.4) to specify how many answer sets are to be computed, they simply ignore n. Furthermore, the "`{ ... }`" part can equivalently be expressed in terms of integrity constraints, as indicated in the comments provided along with the following example:

```
q(1;2).
{ p(1..5) }.
#compute 0 { p(X) : q(X) }.       % :- 1 { not p(X) : q(X) }.
 compute   { not p(X) : X=4..5 }. % :- 1 { p(X) : X=4..5 }.
```

4.2 Input Language of `iclingo`

System `iclingo` [28] extends `clingo` by an *incremental* computation mode that incorporates both grounding and solving. Hence, its input language includes all constructs described in Section 4.1.[17] In addition, `iclingo` deals with statements of the following form:

```
#base.
#cumulative constant.
#volatile   constant.
```

Via "`#base.`," the subsequent part of a logic program is declared as static, that is, it is processed only once at the beginning of an incremental computation. In contrast, "`#cumulative constant.`" and "`#volatile constant.`" are used to declare a (symbolic) *`constant`* as a placeholder for incremental step numbers. In the parts of a logic program below a `#cumulative` statement, *`constant`* is in each step replaced with the current step number, and the resulting rules, facts, and integrity constraints are accumulated over a whole incremental computation. While the replacement of *`constant`* is similar, a logic program part below a `#volatile` statement is local to steps, that is, all rules, facts, and integrity constraints computed in one step are dismissed before the next incremental step. Note that the type of a logic program part (static, cumulative, or volatile) is determined by the last `#base`, `#cumulative`, or `#volatile` statement preceding it.

During an incremental computation, all static program parts are grounded first, while cumulative and volatile parts are grounded step-wise, replacing *`constant`*s with successive step numbers starting from `1`. After a grounding step, `clasp` is usually invoked via an internal interface (like with `clingo`), and the incremental computation stops after a step in which at least one answer set has been found by `clasp`. This default behavior can be readapted via command line options (cf. Section 6.3). For obtaining a well-defined incremental computation result, it is important that (ground) head atoms within static, cumulative, and volatile program parts are distinct from each other, and they must also be different from step to step (see [28] for details). In Section 5.3, we will provide a typical example in which these conditions naturally hold.

4.3 Input Language of `clasp`

Solver `clasp` [29] works on logic programs in `lparse`'s output format [68]. This numerical format, which is not supposed to be human-readable, is output by `gringo` and can be piped into `clasp`. Such an invocation of `clasp` looks as follows:

[17] In its current version, `iclingo` has no built-in support for classical negation (cf. Section 4.1.2). That is, complementary atoms may jointly belong to answer sets, unless explicitly prohibited by integrity constraints.

```
gringo [ options | filenames ] | clasp [ number | options ]
```

Note that `number` may be provided to specify a maximum number of answer sets to be computed, where `0` makes `clasp` compute all answer sets. This maximum number can also be set via option `--number` or its abbreviation `-n` (cf. Section 6.4). By default, `clasp` computes one answer set (if it exists). If a logic program in `lparse`'s output format has been stored in a `file`, it can be redirected into `clasp` as follows:[18]

```
clasp [ number | options ] < file
```

Via option `--dimacs`, `clasp` can also be instructed to compute models of a propositional formula in DIMACS/CNF format [13]. If such a formula is contained in `file`, then `clasp` can be invoked in the following way:

```
clasp [ number | options ] --dimacs < file
```

Finally, `clasp` may be used as a library, as done within `clingo` and `iclingo`.

5 Examples

We exemplarily solve the following problems in ASP: N-Coloring (Section 5.1), Traveling Salesperson (Section 5.2), and Blocks-World Planning (Section 5.3). While the first problem could likewise be solved within neighboring paradigms, the second one requires checking reachability, something that is rather hard to encode in either Boolean Satisfiability [9] or Constraint Programming [62]. The third problem coming from the area of planning illustrates incremental solving with `iclingo`.

5.1 N-Coloring

As already mentioned in Section 2.1, it is custom in ASP to provide a *uniform* problem definition [50, 56, 65]. We follow this methodology and separate the encoding from an instance of the following problem: given a (directed) graph, decide whether each node can be assigned one of N colors such that any pair of adjacent nodes is colored differently. Note that this problem is NP-complete for $N \geq 3$ (see, e.g., [59]), and thus it seems unlikely that a worst-case polynomial algorithm can be found. In view of this, it is convenient to reduce the particular problem to a declarative problem solving paradigm like ASP, where efficient off-the-shelf tools like `gringo` and `clasp` are ready to solve the problem reasonably well. Such a reduction is now exemplified.

5.1.1 Problem Instance

We consider directed graphs specified via facts over predicates `node`/1 and `edge`/2.[19] The graph shown in Figure 3 is represented by the following set of facts:

```
1 % Nodes
2 node(1..6).
3 % (Directed) Edges
4 edge(1,2;3;4).  edge(2,4;5;6).  edge(3,1;4;5).
5 edge(4,1;2).    edge(5,3;4;6).  edge(6,2;3;5).
```

Recall from Section 4.1 that "`..`" and "`;`" in the head expand to multiple rules, which are facts here. Thus, the instance contains six nodes and 17 directed edges.

[18] The same is achieved by using option `--file` or its short form `-f` (cf. Section 6.4).

[19] Directedness is not an issue in N-Coloring, but we will reuse our directed example graph in Section 5.2.

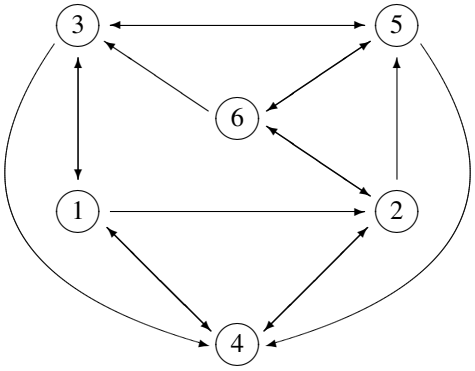

Figure 3: A Directed Graph with Six Nodes and 17 Edges.

5.1.2 Problem Encoding

We now proceed by encoding N-coloring via non-ground rules that are independent of particular instances. Typically, an encoding consists of a *Generate*, a *Define*, and a *Test* part [45]. As N-Coloring has a rather simple pattern, the following encoding does not contain any Define part:

```
1 % Default
2 #const n = 3.
3 % Generate
4 1 { color(X,1..n) } 1 :- node(X).
5 % Test
6 :- edge(X,Y), color(X,C), color(Y,C).
```

In Line 2, we use the `#const` declarative, described in Section 4.1.12, to install `3` as default value for constant `n` that is to be replaced with the number N of colors. (The default value can be overridden by invoking `gringo` with option `--const n=`N.) The Generate rule in Line 4 makes use of the `count` aggregate (cf. Section 4.1.10). For our example graph and `1` substituted for `X`, we obtain the following ground rule:

```
1 { color(1,1), color(1,2), color(1,3) } 1.
```

The full ground program is obtained by invoking:

```
gringo -t \
examples/color.lp \
examples/graph.lp
```

Note that `node(1)` has been removed from the body, as it is derived via a corresponding fact, and similar ground instances are obtained for the other nodes `2` to `6`. Furthermore, for each instance of `edge`/2, we obtain `n` ground instances of the integrity constraint in Line 6, prohibiting that the same color `C` is assigned to the adjacent nodes. Given `n=3`, we get the following ground instances due to `edge(1,2)`:

```
:- color(1,1), color(2,1).
:- color(1,2), color(2,2).
:- color(1,3), color(2,3).
```

Again note that `edge(1,2)`, derived via a fact, has been removed from the body.

5.1.3 Problem Solution

Provided that a given graph is colorable with `n` colors, a solution can be read off an answer set of the program consisting of the instance and the encoding. For the graph in Figure 3, the following answer set can be computed:

To find an answer set, invoke:

```
gringo \
examples/color.lp \
examples/graph.lp | \
clasp
```

or alternatively:

```
clingo \
examples/color.lp \
examples/graph.lp
```

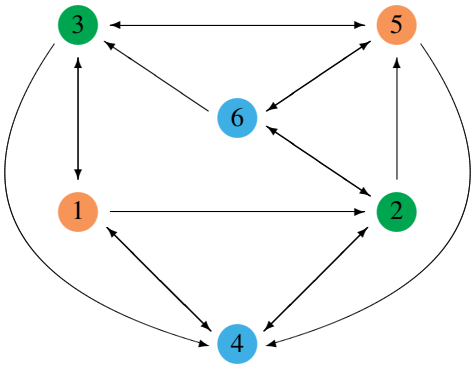

Figure 4: A 3-Coloring for the Graph in Figure 3.

```
Answer: 1
... color(1,2) color(2,1) color(3,1) color(4,3) color(5,2) color(6,3)
```

Note that we have omitted the six instances of `node`/1 and the 17 instances of `edge`/2 in order to emphasize the actual solution, which is depicted in Figure 4. Such output projection can also be specified within a logic program file by using the declaratives `#hide` and `#show`, described in Section 4.1.12.

5.2 Traveling Salesperson

We now consider the well-known Traveling Salesperson Problem (TSP), where the task is to decide whether there is a round trip that visits each node in a graph exactly once (viz., a Hamiltonian cycle) and whose accumulated edge costs must not exceed some budget B. We tackle a slightly more general variant of the problem by not a priori fixing B to any integer. Rather, we want to compute a minimum budget B along with a round trip of cost B. This problem is FP^{NP}-complete (cf. [59]), that is, it can be solved with a polynomial number of queries to an NP-oracle. As with N-Coloring, we provide a uniform problem definition by separating the encoding from instances.

5.2.1 Problem Instance

We reuse graph specifications in terms of predicates `node`/1 and `edge`/2 as in Section 5.1.1. In addition, facts over predicate `cost`/3 are used to define edge costs:

```
1 % Edge Costs
2 cost(1,2,2).  cost(1,3,3).  cost(1,4,1).
3 cost(2,4,2).  cost(2,5,2).  cost(2,6,4).
4 cost(3,1,3).  cost(3,4,2).  cost(3,5,2).
5 cost(4,1,1).  cost(4,2,2).
6 cost(5,3,2).  cost(5,4,2).  cost(5,6,1).
7 cost(6,2,4).  cost(6,3,3).  cost(6,5,1).
```

Figure 5 shows the (directed) graph from Figure 3 along with the associated edge costs. Symmetric edges have the same costs here, but differing costs would also be possible.

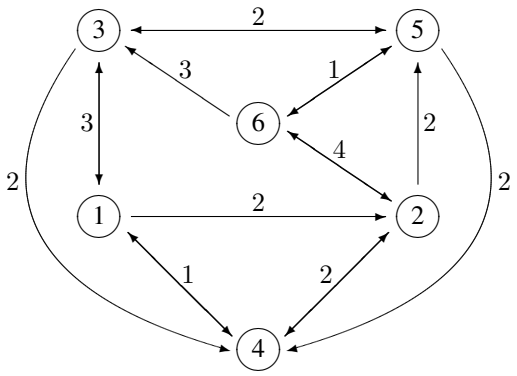

Figure 5: The Graph from Figure 3 along with Edge Costs.

5.2.2 Problem Encoding

The first subproblem consists of describing a Hamiltonian cycle, constituting a candidate for a minimum-cost round trip. Using the Generate-Define-Test pattern [45], we encode this subproblem via the following non-ground rules:

```
1  % Generate
2  1 { cycle(X,Y) : edge(X,Y) } 1 :- node(X).
3  1 { cycle(X,Y) : edge(X,Y) } 1 :- node(Y).
4  % Define
5  reached(Y) :- cycle(1,Y).
6  reached(Y) :- cycle(X,Y), reached(X).
7  % Test
8  :- node(Y), not reached(Y).
9  % Display
10 #hide.
11 #show cycle/2.
```

The Generate rules in Line 2 and 3 assert that every node must have exactly one outgoing and exactly one incoming edge, respectively, belonging to the cycle. By inserting the available edges, for node `1`, Line 2 and 3 are grounded as follows:

```
1 { cycle(1,2), cycle(1,3), cycle(1,4) } 1.
1 { cycle(3,1), cycle(4,1) } 1.
```

The full ground program is obtained by invoking:

```
gringo -t \
examples/ham.lp \
examples/min.lp \
examples/costs.lp \
examples/graph.lp
```

Observe that the first rule groups all outgoing edges of node `1`, while the second one does the same for incoming edges. We proceed by considering the Define rules in Line 5 and 6, which recursively check whether nodes are reached by a cycle candidate produced via the Generate part. Note that the rule in Line 5 builds on the assumption that the cycle "starts" at node `1`, that is, any successor `Y` of `1` is reached by the cycle. The second rule in Line 6 states that, from a reached node `X`, an adjacent node `Y` can be reached via a further edge in the cycle. Note that this definition admits positive recursion among the ground instances of `reached`/1, in which case a ground program is called *non-tight* [22, 23]. The "correct" treatment of (positive) recursion due to the justification requirement w.r.t. the reduct (cf. Section 2.3) is a particular feature of answer set semantics, which is hard to mimic in either Boolean Satisfiability [9] or Constraint Programming [62]. In our present problem, this feature makes sure that all

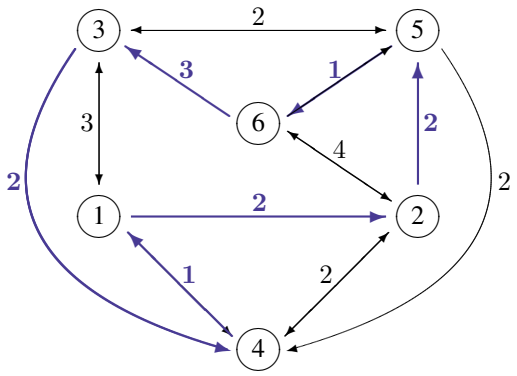

Figure 6: A Minimum-Cost Round Trip.

nodes are reached by a global cycle from node `1`, thus, excluding isolated subcycles. In fact, the Test in Line 8 stipulates that every node in the given graph is reached, that is, the instances of `cycle`/2 in an answer set must be the edges of a Hamiltonian cycle. Finally, the additional Display part in Line 10 and 11 states that answer sets should be projected to instances of `cycle`/2, as only they describe a solution. We have so far not considered edge costs, and answer sets for the above part of the encoding correspond to Hamiltonian cycles, that is, candidates for a minimum-cost round trip.

In order to minimize costs, we add the following optimization statement:

```
13  % Optimize
14  minimize [ cycle(X,Y) : cost(X,Y,C) = C ].
```

Here, edges belonging to the cycle are weighted according to their costs. After grounding, the minimization in Line 14 ranges over 17 instances of `cycle`/2, one for each weighted edge in Figure 5.

To compute the six Hamiltonian cycles for the graph in Figure 3, invoke:

```
gringo \
examples/ham.lp \
examples/graph.lp | \
clasp -n 0
```

or alternatively:

```
clingo -n 0 \
examples/ham.lp \
examples/graph.lp
```

5.2.3 Problem Solution

Finally, we explain how the unique minimum-cost round trip (depicted in Figure 6) can be computed. The catch is that we are now interested in optimal answer sets, rather than in arbitrary ones. In order to determine the optimum, we can start by gradually decreasing the costs associated to answer sets until we cannot find a strictly better one anymore. To this end, it is important to invoke `clasp` (or `clingo`) with option "`-n 0`" (cf. Section 6.4), making it enumerate successively better answer sets w.r.t. the provided optimization statements (cf. Section 4.1.11). Any answer set is printed as soon as it has been computed, and the last one is optimal. If there are multiple optimal answer sets, an arbitrary one among them is computed. For the graph in Figure 5, the optimal answer set (cf. Figure 6) is unique, and its computation can proceed as follows:

```
Answer: 1
cycle(1,3) cycle(2,4) cycle(3,5) \
cycle(4,1) cycle(5,6) cycle(6,2)
Optimization: 13
Answer: 2
cycle(1,2) cycle(2,5) cycle(3,4) \
cycle(4,1) cycle(5,6) cycle(6,3)
```

To compute the minimum-cost round trip for the graph in Figure 5, invoke:

```
gringo \
examples/ham.lp \
examples/min.lp \
examples/costs.lp \
examples/graph.lp | \
clasp -n 0
```

or alternatively:

```
clingo -n 0 \
examples/ham.lp \
examples/min.lp \
examples/costs.lp \
examples/graph.lp
```

```
Optimization: 11
```

Given that no answer is obtained after the second one, we know that `11` is the optimum value, but there might be more optimal answer sets that have not been computed yet. In order to find them too, we can use the following command line options of `clasp` (cf. Section 6.4): "`--opt-value=11`" in order to initialize the optimum and "`--opt-all`" to compute also equitable (rather than only strictly better) answer sets. After obtaining only the second answer given above, we are sure that this is the unique optimal answer set, whose associated edge costs (cf. Figure 6) correspond to the reported optimization value `11`. Note that, with `maximize` statements in the input, this correlation might be less straightforward because they are compiled into `minimize` statements in the process of generating `lparse`'s output format [68]. Furthermore, if there are multiple optimization statements, `clasp` (or `clingo`) will report separate optimization values ordered by significance. Finally, the two-stage invocation scheme exercised above, first determining the optimum and afterwards all (and only) optimal answer sets, is recommended for general use. Otherwise, if using option "`--opt-all`" right away without knowing the optimum, one risks the enumeration of plenty suboptimal answer sets. We also invite everyone to explore command line option "`--opt-restart`" (cf. Section 6.4) in order to see whether it improves search efficiency on optimization problems.

The full invocation is:

```
gringo \
examples/ham.lp \
examples/min.lp \
examples/costs.lp \
examples/graph.lp |\
clasp -n 0 \
--opt-value=11 \
--opt-all
```

or alternatively:

```
clingo -n 0 \
--opt-value=11 \
--opt-all \
examples/ham.lp \
examples/min.lp \
examples/costs.lp \
examples/graph.lp
```

5.3 Blocks-World Planning

The Blocks-World is a well-known planning domain, where finding *shortest* plans has received particular attention [41]. In an eventually propositional formalism like ASP, a bound on the plan length must be fixed before search can proceed. This is usually accomplished by including some constant `t` in an encoding, which is then replaced with the actual bound during grounding. Of course, if the length of a shortest plan is unknown, an ASP system must repeatedly be queried while varying the bound. With a traditional ASP system, processing the same planning problem with a different bound involves grounding and solving from scratch. In order to reduce such redundancies, the incremental ASP system `iclingo` [28] can gradually increase a bound, doing only the necessary additions in each step. Note that planning is a natural application domain for `iclingo`, but other problems including some mutable bound can be addressed too, thus, `iclingo` is not a specialized planning system. However, we use Blocks-World Planning to illustrate the exploitation of `iclingo`'s incremental computation mode.

5.3.1 Problem Instance

As with the other three problems above, an instance is given by a set of facts, here, over predicates `block`/1 (declaring blocks), `init`/1 (defining the initial state), and `goal`/1 (specifying the goal state). A well-known Blocks-World instance is described by:[20]

```
1 % Sussman Anomaly
2 %
3 block(b0).
4 block(b1).
5 block(b2).
6 %
```

[20] Blocks-World instances "`examples/world`i`.lp`" for $i \in \{0,1,2,3,4\}$ are adaptations of the instances provided at [21].

```
 7 % initial state:
 8 %
 9 %  2
10 %  0 1
11 % -------
12 %
13 init(on(b1,table)).
14 init(on(b2,b0)).
15 init(on(b0,table)).
16 %
17 % goal state:
18 %
19 %  2
20 %  1
21 %  0
22 % -------
23 %
24 goal(on(b1,b0)).
25 goal(on(b2,b1)).
26 goal(on(b0,table)).
```

Note that the facts in Line 13–15 and 24–26 specify the initial and the goal state depicted in Line 9-11 and 19–22, respectively. We here use (uninterpreted) function `on`/2 to illustrate another important feature available in `gringo`, `clingo`, and `iclingo`, namely, the possibility of instantiating variables to compound terms.

5.3.2 Problem Encoding

Our Blocks-World Planning encoding for `iclingo` makes use of declaratives `#base`, `#cumulative`, and `#volatile`, separating the encoding into a static, a cumulative, and a volatile (query) part. Each of them can be further refined into Generate, Define, Test, and Display constituents, as indicated in the comments below:

```
 1 #base.
 2 % Define
 3 location(table).
 4 location(X) :- block(X).
 5 holds(F,0) :- init(F).
 6 %
 7 #cumulative t.
 8 % Generate
 9 1 { move(X,Y,t) : block(X) : location(Y) : X != Y } 1.
10 % Test
11 :- move(X,Y,t),
12    1 { holds(on(A,X),t-1) : block(A) : A != X,
13        holds(on(B,Y),t-1) : block(B) : B != X : B != Y : Y != table }.
14 % Define
15 holds(on(X,Y),t) :- move(X,Y,t).
16 holds(on(X,Z),t) :- holds(on(X,Z),t-1), block(X), location(Z), X != Z,
17                      { move(X,Y,t) : location(Y) : Y != X : Y != Z } 0.
18 %
```

```
19 #volatile t.
20 % Test
21 :- goal(F), not holds(F,t).
22 %
23 #base.
24 % Display
25 #hide.
26 #show move/3.
```

In the initial `#base` part (Line 1–5), we define blocks and constant `table` as instances of predicate `location`/1. Moreover, we use instances of `init`/1 to initialize predicate `holds`/2 for step `0`, thus, defining the conditions before the first incremental step. Note that variable `F` is instantiated to compound terms over function `on`/2.

The `#cumulative` statement in Line 7 declares constant `t` as a placeholder for step numbers in the cumulative encoding part below. Here, the Generate rule in Line 9 states that exactly one block `X` must be moved to a location `Y` (different from `X`) at each step `t`. The integrity constraint in Line 11–13 is used to test whether moving block `X` to location `Y` is possible at step `t` by denying `move(X,Y,t)` to hold if there is either some block `A` on `X` or some block `B` distinct from `X` on `Y` (this condition is only checked if `Y` is a block, viz., different from constant `table`) at step `t-1`. Finally, the Define rule in Line 15 propagates a move to the state at step `t`, while the rule in Line 16–17 states that a block `X` stays on a location `Z` if it is not moved to any other location `Y`. Note that we require atoms `block(X)` and `location(Z)` to bind variables `X` and `Z` in the body of the latter rule, as recursive predicate `holds`/2 cannot be used for this purpose (cf. the definition of level-restrictedness in Section 3.1).

The `#volatile` statement in Line 19 declares the next part as a query depending on step number `t`, but not accumulated over successive steps. In fact, the integrity constraint in Line 21 tests whether goal conditions are satisfied at step `t`.

Our incremental encoding concludes with a second `#base` part, as specified in Line 23. Note that, for the meta-statements with Display functionality (Line 25–26), it is actually unimportant whether they belong to a static, cumulative, or volatile program part, as answer sets are projected (to instances of `move`/3) in either case. However, by ending the encoding file with a `#base` statement, we make sure that the contents of a concatenated instance file is included in the static program part. This is also the default of `iclingo` (as well as of `gringo` and `clingo` that can be used for non-incremental computations), but applying this default behavior triggers a warning (cf. Section 7.2).

To observe the ground program dealt with internally `iclingo` at a step n, invoke:

```
gringo -t \
--ifixed=n \
examples/blocks.lp \
examples/worldi.lp
```

where $i \in \{0, 1, 2, 3, 4\}$.

Finally, let us stress important prerequisites for obtaining a well-defined incremental computation result from `iclingo`. First, the ground instances of head atoms of rules in the static, cumulative, and volatile program part must be pairwisely disjoint. Furthermore, ground instances of head atoms in the volatile part must not occur in the static and cumulative parts, and those of the cumulative part must not be used in the static part. Finally, ground instances of head atoms in either the cumulative or the volatile part must be different for each pair of distinct steps. This is the case for our encoding because both atoms over `move`/3 and those over `holds`/2 include `t` as an argument in the heads of the rules in Line 9, 15, and 16–17. As the smallest step number to replace `t` with is `1`, there also is no clash with the ground atoms over `holds`/2 obtained from the head of the static rule in Line 5. Further details on the sketched requirements and their formal background can be found in [28]. Arguably, many problems including some mutable bound can be encoded such that the prerequisites apply. Some attention should of course be spent on putting rules into the right program parts.

5.3.3 Problem Solution

We can now use `iclingo` to *incrementally* compute the shortest sequence of moves that brings us from the initial to the goal state depicted in the instance in Section 5.3.1:

```
Answer: 1
move(b2,table,1) move(b1,b0,2) move(b2,b1,3)
```

To this end, invoke:

```
iclingo -n 0 \
examples/blocks.lp \
examples/world0.lp
```

This unique answer set tells us that the given problem instance can be solved by moving block `b2` to the `table` in order to then put `b1` on top of `b0` and finally `b2` on top of `b1`. This solution is computed by `iclingo` in three grounding and solving steps, where, starting from the `#base` program, constant `t` is successively replaced with step numbers `1`, `2`, and `3` in the `#cumulative` and in the `#volatile` part. While the query postulated in the `#volatile` part cannot be fulfilled in steps `1` and `2`, `iclingo` stops its incremental computation after finding an answer set in step `3`. The scheme of iterating steps until finding at least one answer set is the default behavior of `iclingo`, which can be customized via command line options (cf. Section 6.3).

Finally, let us describe how solutions obtained via an incremental computation can be computed in the standard way, that is, in a single pass. To this end, the step number can be fixed to some n via option "`--ifixed=`n" (cf. Section 6.1), enabling `gringo` or `clingo` to generate the ground program present inside `iclingo` at step n. Note that `#volatile` parts are here only instantiated for the final step n, while `#cumulative` rules are added for all steps $1, \ldots, n$. Option "`--ifixed=`n" can be useful for investigating the contents of a ground program dealt with at step n or for using an external solver (other than `clasp`). In the latter case, repeated invocations with varying n are required if the bound of an optimal solution is a priori unknown.

For non-incremental solving, invoke:

```
gringo --ifixed=n \
examples/blocks.lp \
examples/worldi.lp |\
clasp -n 0
```

or alternatively:

```
clingo -n 0 \
--ifixed=n \
examples/blocks.lp \
examples/worldi.lp
```

where $i \in \{0, 1, 2, 3, 4\}$.

6 Command Line Options

In this section, we briefly describe the meanings of command line options supported by `gringo` (Section 6.1), `clingo` (Section 6.2), `iclingo` (Section 6.3), and `clasp` (Section 6.4). Each of these tools displays its available options when invoked with flag `--help` or `-h`, respectively.[21] The approach of distinguishing long options, starting with "`--`," and short ones of the form "`-`*`l`*," where *`l`* is a letter, follows the GNU Coding Standards [38]. For obvious reasons, short forms are made available only for the most common (long) options. Some options, also called flags, do not take any argument, while others require arguments. An argument *`arg`* is provided to a (long) option *`opt`* by writing "`--`*`opt`*`=`*`arg`*" or "`--`*`opt arg`*," while only "`-`*`l arg`*" is accepted for a short option *`l`*. For each command line option, we below indicate whether it requires an argument, and if so, we also describe its meaning.

6.1 `gringo` Options

An abstract invocation of `gringo` looks as follows:

```
gringo [ options | filenames ]
```

[21] Note that our description of command line options is based on Version 2.0.2 of `gringo`, `clingo`, and `iclingo` as well as Version 1.1.2 of `clasp`. While it is rather unlikely that command line options will disappear in future versions, additional ones might be introduced. We will try to keep this document up-to-date, but checking the help information shipped with a new version is always a good idea.

Note that options and filenames do not need to be passed to `gringo` in any particular order. If neither a filename nor an option that makes `gringo` exit (see below) is provided, `gringo` reads from the standard input. In the following, we list and describe the options accepted by `gringo` along with their particular arguments (if required):

`--help,-h`
Print help information and exit.

`--version,-v`
Print version information and exit.

`--syntax`
Print syntax information (about the input language) and exit.

`--stats`
Print (extended) statistic information before termination.

`--verbose,-V`
Print additional (progress) information during computation.

`--debug`
Print internal representations of rules during grounding. (This may be used to identify either semantic errors in an input program or performance bottlenecks.)

`--const,-c c=t`
Replace occurrences (in the input program) of a constant `c` with a term `t`.

`--text,-t`
Output ground program in (human-readable) text format.

`--lparse,-l`
Output ground program in `lparse`'s numerical format [68].

`--aspils,-a 1|2|3|4|5|6|7`
Output ground program in *ASPils* format [27], where the argument specifies one of the seven normal forms defined in [27].

`--ground,-g`
Enable lightweight mode for processing a ground input program. (This option is recommended to omit unnecessary overhead if the input program is already ground, but it leads to a syntax error (cf. Section 7.1) otherwise.)

`--bindersplit yes|no`
Enable or disable binder-splitting [33] in the instantiation of rules. (This option is included mainly for comparison purposes, and generally, it is recommended to keep the default argument value `yes`.)

`--ifixed n`
Use `n` as fix step number if the input program contains `#cumulative` or `#volatile` statements. (This option permits the handling of programs written for `iclingo` in a traditional single pass computation.)

`--ibase`
Process only the static part (that can be initiated by a `#base` statement) of an input program. (This option may be used to investigate the basic setting of a problem including some mutable bound.)

The default command line when invoking `gringo` is as follows:

```
gringo --lparse --bindersplit=yes
```

That is, `gringo` usually outputs a ground program in `lparse`'s numerical format, dealt with by various solvers (cf. Section 2.1), and it performs binder-splitting in rule instantiation.

6.2 `clingo` Options

ASP system `clingo` combines grounder `gringo` and solver `clasp` via an internal interface. An abstract invocation of `clingo` looks as follows:

```
clingo [ number | options | filenames ]
```

A numerical argument is permitted for backward compatibility to the usage of solver `smodels` [66], where it specifies the maximum number of answer sets to be computed (`0` standing for all answer sets). As with `gringo`, a number, options, and filenames do not need to be passed to `clingo` in any particular order. Given that `clingo` combines `gringo` and `clasp`, it accepts all options described in the previous section and in Section 6.4. In particular, (long) options `--help`, `--version`, and `--syntax` make `clingo` print the desired information and exit, while `--text`, `--lparse`, and `--aspils` instruct `clingo` to output a ground program (rather than solving it) like `gringo`. If neither a filename nor an option that makes `clingo` exit (see Section 6.1) is provided, `clingo` reads from the standard input. Beyond the options described in Section 6.1 and 6.4, `clingo` has a single additional option:

`--clasp`
: Run `clingo` as a plain solver (using embedded `clasp`).

Finally, the default command line when invoking `clingo` consists of all `clasp` defaults (cf. Section 6.4) and option `--bindersplit=yes` of `gringo`.

6.3 `iclingo` Options

Incremental ASP system `iclingo` extends `clingo` by interleaving grounding and solving for problems including a mutable bound, and an abstract invocation of `iclingo` is as with `clingo`:

```
iclingo [ number | options | filenames ]
```

The external behavior of `iclingo` is similar to `clingo`, described in the previous section, except for the fact that option `--ifixed` is ignored by `iclingo` if not run as a grounder (via one of (long) options `--text`, `--lparse`, or `--aspils`). However, option `--clingo` (see below) may be used to let `iclingo` work like `clingo`. The additional options of `iclingo` focus on customizing incremental computations:

`--istats`
: Print statistic information for each incremental solving step.

`--imin` *`n`*
: Perform at least *n* incremental solving steps before termination. (This may be used to force steps regardless of the termination condition set via `--istop`.)

--imax *n*
: Perform at most *n* incremental solving steps before termination. (This may be used to limit steps regardless of the termination condition set via `--istop`.)

--istop SAT|UNSAT
: Terminate after an incremental solving step in which some (`SAT`) or no (`UNSAT`) answer set has been found.

--iquery *n*
: Start with incremental solving at step number *n*. (This may be used to skip some solving steps, still accumulating static and cumulative rules for these steps.)

--ilearnt keep|forget
: Maintain (`keep`) or delete (`forget`) learnt constraints in-between incremental solving steps. (This option configures the behavior of embedded `clasp`.)

--iheuristic keep|forget
: Maintain (`keep`) or delete (`forget`) heuristic information in-between incremental solving steps. (This option configures the behavior of embedded `clasp`.)

--clingo
: Run `iclingo` as a non-incremental ASP system (like `clingo`).

As with `clingo`, the default command line when invoking `iclingo` consists of all `clasp` defaults, explained in the next section, option `--bindersplit=yes` of `gringo` (cf. Section 6.1), along with `--istop=SAT`, `--iquery=1`, `--ilearnt=keep`, and `--iheuristic=forget`. That is, incremental solving starts at step number `1` and stops after a step in which some answer set has been found. In-between incremental solving steps, embedded `clasp` maintains learnt constraints but deletes heuristic information.

6.4 `clasp` Options

Stand-alone `clasp` [29] is a solver for ground programs in `lparse`'s numerical format [68]. An abstract invocation of `clasp` looks as follows:

```
clasp [ number | options ]
```

As with `clingo` and `iclingo`, a numerical argument specifies the maximum number of answer sets to be computed, where `0` stands for all answer sets. (The number of requested answer sets can likewise be set via long option `--number` or its short form `-n`.) If neither a filename (via long option `--file` or its short form `-f`) nor an option that makes `clasp` exit (see below) is provided, `clasp` reads from the standard input. In fact, it is typical to use `clasp` in a pipe with `gringo` in the following way:

```
gringo [ ... ] | clasp [ ... ]
```

In such a pipe, `gringo` instantiates an input program and outputs the ground rules in `lparse`'s numerical format, which is then consumed by `clasp` that computes and outputs answer sets. Note that `clasp` offers plenty of options to configure its behavior. We thus categorize them according to their functionalities in the following description.

6.4.1 General Options

We below group general options of `clasp`, used to configure its global behavior.

`--help,-h`
Print help information and exit.

`--version,-v`
Print version information and exit.

`--stats`
Print (extended) statistic information before termination.

`--file,-f` *`filename`*
Read from file *`filename`* (rather than from the standard input).

`--number,-n` *`n`*
Compute at most *`n`* answer sets, *`n`* = `0` standing for compute all answer sets.

`--quiet,-q`
Do not print computed answer sets. (This is useful for benchmarking.)

`--seed` *`n`*
Use seed *`n`* (rather than `1`) for random number generator.

`--brave`
Compute the brave consequences (union of all answer sets) of a logic program.

`--cautious`
Compute the cautious consequences (intersection of all answer sets) of a logic program.

`--opt-all`
Compute all optimal answer sets (cf. Section 4.1.11). (This is implemented by enumerating [30] answer sets that are not worse than the best one found so far.)

`--opt-restart`
Restart the search from scratch (rather than doing enumeration [30]) after finding a provisionally optimal answer set. (This may speed up finding the optimum.)

`--opt-value` *`n1`*`[,`*`n2`*`,`*`n3`*`...]`
Initialize objective function(s) to minimize with *`n1`*`[,`*`n2`*`,`*`n3`*`...]`. Depending on the reasoning mode, only equitable or strictly better answer sets are computed.

`--supp-models`
Compute supported models [3] (rather than answer sets).

`--dimacs`
Compute models of a propositional formula in DIMACS/CNF format [13].

`--trans-ext all|choice|weight|no`
Compile extended rules [66] into normal rules of form (1). Arguments `choice` and `weight` state that all "choice rules" or all "weight rules," respectively, are to be compiled into normal rules, while `all` means that both and `no` that none of them are subject to compilation.

`--eq` *`n`*
Run equivalence reasoning [32] for n iterations, $n = -1$ and $n = 0$ standing for run to fixpoint or do not run equivalence reasoning, respectively.

`--sat-prepro yes|no|`*`n1`*`[,`*`n2`*`,`*`n3`*`]`
Run *SatElite*-like preprocessing [17] for at most $n1$ iterations ($n1 = -1$ standing for run to fixpoint), using cutoff $n2$ for variable elimination ($n2 = -1$ standing for no cutoff), and for no longer than $n3$ seconds ($n3 = -1$ standing for no time limit). Arguments `yes` and `no` mean $n1 = n2 = n3 = -1$ (that is, run to fixpoint) or that *SatElite*-like preprocessing is not to be run at all, respectively.

`--rand-watches yes|no`
Initially choose watched literals randomly (with argument `yes`) or systematically (with argument `no`).

Having introduced the general options of `clasp`, let us note that the options below `--dimacs` in the above list are quite low-level and more or less an issue of fine-tuning. More important is the fact that virtually all optimization functionalities are only provided by `clasp` if the maximum number of answer sets to be computed is set to `0` (standing for all answer sets), as it is likely to stop search too early otherwise. The same applies to computing either brave or cautious consequences (via one of the flags `--brave` and `--cautious`).

6.4.2 Search Options

The options listed below can be used to configure the main search strategies of `clasp`.

`--lookback yes|no`
Enable or disable lookback techniques (cf. Section 6.4.3). (This option is included mainly for comparison purposes, and generally, it is recommended to keep the default argument value `yes`.)

`--lookahead atom|body|hybrid|auto|no`
Apply failed-literal detection [26] to atoms (with argument `atom`), to rule bodies (with argument `body`), to atoms and rule bodies like in `nomore++` [1] (with argument `hybrid`), or let `clasp` pick one of the previous three possibilities based on program properties (with argument `auto`). Failed-literal detection is switched off via argument `no`.

`--initial-lookahead`
Apply failed-literal detection (to atoms) in preprocessing, but not (necessarily) during search.

`--heuristic Berkmin|Vmtf|Vsids|Unit|None`
Use *BerkMin*-like decision heuristic [40] (with argument `Berkmin`), *Siege*-like decision heuristic [63] (with argument `Vmtf`), *Chaff*-like decision heuristic [55] (with argument `Vsids`), *Smodels*-like decision heuristic [66] (with argument `Unit`), or (arbitrary) static variable ordering (with argument `None`).

`--rand-freq` *`p`*
Perform random (rather than heuristic) decisions with probability $0 \leq p \leq 1$.

--rand-prob yes|no|*n1,n2*
: Run *Satzoo*-like random probing [16], initially performing *n1* passes of up to *n2* conflicts making random decisions. Arguments `yes` and `no` mean $n1 = 50$, $n2 = 20$ or that random probing is not to be run at all, respectively.

Let us note that switching the above options can have dramatic effects (both positively and negatively) on the search performance of `clasp`. Sticking to the default argument values `yes` for `--lookback` and `no` for `--lookahead` is generally recommended. If nonetheless performance bottlenecks are observed, it is worthwhile to give `Vmtf` and `Vsids` for `--heuristic` a try, in particular, when the program under consideration is huge but scarcely yields conflicts during search.

6.4.3 Lookback Options

The following options have an effect only if `--lookback` is turned on, that is, its argument value must be `yes`.

--restarts,-r *n1*[,*n2*,*n3*]|no
: Choose and parameterize a restart policy. If a single argument *n1* is provided, `clasp` restarts search from scratch after a number of conflicts determined by a universal sequence [49], where *n1* constitutes the base unit. If two arguments *n1*,*n2* are specified, `clasp` runs a geometric sequence [18], restarting every $n1 * n2^i$ conflicts, where i is the number of restarts performed so far. Given three arguments *n1*,*n2*,*n3*, `clasp` repeats geometric restart sequence $n1*n2^i$ when it reaches an outer limit $n1 * n3^j$ [8], where j counts how often the outer limit has been hit so far. Finally, restarts are disabled via argument `no`.

--local-restarts
: Count conflicts locally [64] (rather than globally) for deciding when to restart.

--bounded-restarts
: Perform bounded restarts in answer set enumeration [30].

--save-progress
: Use cached (rather than heuristic) decisions [60] if available.

--shuffle,-s *n1*,*n2*
: Shuffle internal data structures after *n1* restarts ($n1 = 0$ standing for no shuffling) and then reshuffle every *n2* restarts ($n2 = 0$ standing for no reshuffling).

--deletion,-d *n1*[,*n2*,*n3*]
: Limit the number of learnt constraints to $\min\{(c/n1) * n2^i, c * n3\}$, where c is the initial number of constraints and i is the number of restarts performed so far.

--reduce-on-restart
: Delete a portion of learnt constraints after every restart.

--strengthen bin|tern|all|no
: Check binary (with argument `bin`), binary and ternary (with argument `tern`), or all (with argument `all`) antecedents for self-subsumption [17] in order to strengthen a constraint to learn. Strengthening is disabled via argument `no`.

--recursive-str
: Recursively apply strengthening, as proposed in [67].

--loops common|distinct|shared|no
: Learn loop nogood [31] per atom in an unfounded set [70] (with argument `common`), shrink unfounded set before learning another loop nogood (with argument `distinct`), learn loop formula [47] for atoms in an unfounded set (with argument `shared`), or do not record unfounded sets at all (with argument `no`).

--contraction *n*
: Temporarily truncate learnt constraints over more than *n* variables [63].

As with search options described in the previous section, switching lookback options may have a significant impact on the performance of `clasp`. In particular, we suggest trying the universal restart sequence [49] with different base units *n1* or even disabling restarts (both via (long) option `--restarts` or its short form `-r`) in case that performance needs to be improved.

Finally, let us consider the default command line of `clasp`:

```
clasp 1 --seed=1 --trans-ext=no --eq=5 --sat-prepro=no
        --rand-watches=yes --lookback=yes --lookahead=no
        --heuristic=Berkmin --rand-freq=0.0 --rand-prob=no
        --restarts=100,1.5 --shuffle=0,0 --deletion=3.0,1.1,3.0
        --strengthen=all --loops=common --contraction=250
```

Considering only the most significant defaults, numeric argument `1` instructs `clasp` to terminate immediately after finding an answer set, and with `--lookback=yes`, `--restarts=100,1.5` lets `clasp` apply a geometric restart policy.

7 Errors and Warnings

This section explains the most frequent errors and warnings related to inappropriate inputs or command line options that, if they occur, lead to messages sent to the standard error stream. The difference between errors and warnings is that the former involve immediate termination, while the latter are hints pointing at possibly corrupt input that can still be processed further. In the below description of errors (Section 7.1) and warnings (Section 7.2), we refrain from attributing them to a particular one among the tools `gringo`, `clasp`, `clingo`, and `iclingo`, in view of the fact that they share a number of functionalities.

7.1 Errors

We start our description with errors that may be encountered during grounding, where the following one indicates a syntax error in the input:

```
syntax error on line L column C unexpected token: T
```

To correct this error, please investigate the line *L* and check whether something looks strange there (like a missing period, an unmatched parenthesis, etc.).

The next error occurs if an input program is not level-restricted (cf. Section 3.1):

```
the following rule cannot be grounded, \
weakly restricted variables: { V, ... }
        rule
```

Along with the error message, the *rule* and the name *V* of at least one variable causing the problem are reported. The first action to take usually consists of checking whether variable *V* is actually in the scope of any atom (in the positive body of *rule*) that can bind it.[22] If *V* is a local variable belonging to an atom A on the left-hand side of a condition (cf. Section 4.1.8) or to an aggregate (cf. Section 4.1.10), an atom over some domain predicate might be included in a condition to bind *V*. In particular, if A itself is over a domain predicate, the problem is often easily fixed by writing "$A : A$." If the problem is not as simple, then the predicates of all atoms that in principle may be used to bind *V* are involved in positive recursion through the predicate of some atom in the head of *rule*. In this case, the positive bodies of some of the rules subject to recursion must be augmented with additional atoms over non-recursive predicates that can bind variables, in order to eventually establish level-restrictedness. As breaking recursion is a rather global matter, the reported *rule* is not necessarily the best place to tackle it.

The following error is related to conditions (cf. Section 4.1.8):

```
the following rule cannot be grounded, \
non domain predicates : { p(...), ... }
        rule
```

The problem is that an atom `p(...)` such that its predicate *p/i* is not a domain predicate (cf. Section 3.2) is used on the right-hand side of a condition within *rule*. The error is corrected by either removing the atom or by replacing it with another atom over a domain predicate.

The next errors may occur within an arithmetic evaluation (cf. Section 4.1.4):

```
trying to convert string to int
trying to convert functionsymbol to int
```

It means that either a (symbolic) constant or a compound term (over an uninterpreted function with non-zero arity) has occurred in the scope of some built-in arithmetic function. Unfortunately, no context information is provided yet, which is left as an issue to future work (cf. Section 8).

The below errors are related to built-in comparison predicates (cf. Section 4.1.5):

```
comparing different types
comparing function symbols
```

The first error indicates that one of the built-in comparison predicates <, <=, >, and >= has been applied to an integer or a (symbolic) constant and a term of a different sort, while the second error means that compound terms have occurred in such a comparison. The mere reason for these errors is that the required functionalities have not been implemented yet, so that the errors are likely to disappear in the future (cf. Section 8).

An error occurs if a constant *c* ought to be replaced with a term *t* containing *c*:

```
cyclic constant definition
```

To resolve such an error, check the occurrences of `#const` in the input as well as `--const` or `-c` options in the command line, and change them in a way such that a constant that is to be replaced does not (transitively) occur in a term to be inserted.

The next error may occur if a statement from the input language of `iclingo` (cf. Section 4.2) is used in the input of `gringo`, `clingo`, or `iclingo` run non-incrementally (via one of options `--text`, `--lparse`, `--aspils`, or `--clingo`):

[22] Recall from Section 4.1.4 and 4.1.5 that a variable in the scope of a built-in arithmetic function may not be bound by a corresponding atom and that built-in comparison predicates do not bind any variable.

```
A fixed number of incremental steps is needed to ground the program.
```

This problem is solved by providing a step number via option `--ifixed`.

The following error may be reported by `iclingo`:

```
The following statement cant be used with the incremental interface:
        rule
```

The *rule* contains a `#maximize`, `#minimize`, or `#compute` statement, which are currently not supported within incremental computations. Their support by `iclingo` is an issue to future work (cf. Section 8).

The next error may occur if *ASPils* [27] output is requested via option `--aspils`:

```
e not allowed in this normal form, \
please choose normal form n.
```

The chosen normal form (in-between `1` and `7`) does not cover the input language expression *e*, and one among the normal forms *n* suggested in the error message ought to be provided as an argument instead.

The following error message is issued by (embedded) `clasp`:

```
Input Error at line L: Symbol table expected!
```

This error means that the input does not comply with `lparse`'s numerical format [68]. It is not unlikely that the input can be processed by `gringo`, `clingo`, or `iclingo`.

The next error indicates that input in `lparse`'s numerical format [68] is corrupt:

```
Input Error at line L: Atom out of bounds
```

There is no way to resolve this problem. If the input has been generated by `gringo`, `clingo`, or `iclingo`, please report the problem to the authors of this guide.

The following error message is issued by (embedded) `clasp`:

```
Input Error at line L: Unsupported rule type!
```

It means that some rule type in `lparse`'s numerical format [68] is not supported. Most likely, the program under consideration contains rules with disjunction in the head.

A similar error may occur with `clingo` or `iclingo`:

```
sorry clasp cannot handle disjunctive rules!
```

The program under consideration contains rules with disjunction in the head, which are currently not supported by `clasp`, but by `claspD` [15]. The integration of `clasp` and `claspD` is a subject to future work (cf. Section 8).

All of the tools `gringo`, `clasp`, `clingo`, and `iclingo` try to expand incomplete (long) options to recognized ones. Parsing command line options may nonetheless fail due to the following three reasons:

```
unknown option: o
ambiguous option: 'o' could be:
  o1
  o2
  ...
'a': invalid value for Option 'o'
```

The first error means that a provided option *o* could not be expanded to one that is recognized, while the second error expresses that the result of expanding *o* is ambiguous. Finally, the third error occurs if a provided argument *a* is invalid for option *o*. In either case, option `--help` can be used to see the recognized options and their arguments.

7.2 Warnings

The following warnings may be raised by `gringo`, `clingo`, or `iclingo`:

```
Warning: p/i is never defined.
Warning: sum with negative bounds in the body of a rule
Warning: multiple definitions of #const c
```

The first one, stating that a predicate `p/i` has occurred in some rule body, but not in the head of any rule, might point at a mistyped predicate. The second warning means that negative weights within a `sum` aggregate (cf. Section 4.1.10) have been compiled away [66], which is semantically delicate [24]. Thus, we strongly recommend to avoid negative weights within a `sum` aggregate in the positive body of a rule. Finally, the third warning recognizes that multiple `#const` statements over the same constant `c` are contained in the input, but only the first of them will be considered.

The next four warnings may be issued by `iclingo`:

```
Warning: There are no #base, #cumulative or #volatile sections.
Warning: Option ifixed will be ignored!
Warning: There are statements not within a #base, \
         #cumulative or #volatile section.
         These Statements are put into the #base section.
Warning: Classical negation is not handled correctly \
         in combination with the incremental output.
         You have to add rules like: :- a, -a. on your own! \
         (at least for now)
```

For omitting the first warning, one may use option `--clingo`, as the input program contains no statements particular to `iclingo` (cf. Section 4.2). Similarly, option `--clingo` may be used to suppress the second warning, stating that a provided option `--ifixed` is ignored in incremental mode (if `iclingo` is not run as a grounder via one of options `--text`, `--lparse`, or `--aspils`). The third warning states that rules have been inserted into the static program part without a preceding `#base` statement. This warning can be avoided by ending an encoding with `#base.` and by supplying an instance file afterwards, as illustrated in Section 5.3. The fourth warning refers to the fact that there currently is no built-in support for classical negation (cf. Section 4.1.2) by `iclingo`. Hence, integrity constraints of the form "`:-` A`, -`A`.`" have to be added explicitly by a user.

8 Future Work

We conclude this guide with a brief outlook on the future development of `gringo`, `clasp`, `clingo`, and `iclingo`. An important goal of future releases will be improving usability by adding functionalities that make some errors and warnings obsolete or, otherwise, by providing helpful context information along with the remaining ones. In particular, we consider adding support for yet missing traditional features in incremental computations of `iclingo` and also the integration of `clasp` and `claspD` [15], which would enable `clasp`, `clingo`, and `iclingo` to deal with disjunctive programs (cf. [19]) or, more generally, logic programs such that standard reasoning tasks are complete for the second level of the polynomial hierarchy (cf. [59]). Moreover, we are investigating less demanding restrictedness notions (cf. Section 3) that can broaden the class of acceptable input programs. For the representation of ground programs,

ASPils format [27] has been suggested to overcome limitations of `lparse`'s output format [68], and we work on *ASPils* support in `clasp`. In the long term, limitations inherent to present ASP systems, such as space explosion sometimes faced when representing multi-valued variables in a propositional formalism, might be extinguished by systems combining ASP with neighboring paradigms like, e.g., Constraint Programming [62]. Prototypical approaches in such directions already exist today [52, 57].

References

[1] C. Anger, M. Gebser, T. Linke, A. Neumann, and T. Schaub. The `nomore++` approach to answer set solving. In G. Sutcliffe and A. Voronkov, editors, *Proceedings of the Twelfth International Conference on Logic for Programming, Artificial Intelligence, and Reasoning (LPAR'05)*, volume 3835 of *Lecture Notes in Artificial Intelligence*, pages 95–109. Springer-Verlag, 2005. 6, 17, 44

[2] C. Anger, K. Konczak, T. Linke, and T. Schaub. A glimpse of answer set programming. *Künstliche Intelligenz*, 19(1):12–17, 2005. 5

[3] K. Apt, H. Blair, and A. Walker. Towards a theory of declarative knowledge. In Minker [53], pages 89–148. 43

[4] Asparagus. Dagstuhl Initiative. `http://asparagus.cs.uni-potsdam.de/`. 56

[5] C. Baral. *Knowledge Representation, Reasoning and Declarative Problem Solving*. Cambridge University Press, 2003. 5

[6] C. Baral, G. Brewka, and J. Schlipf, editors. *Proceedings of the Ninth International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'07)*, volume 4483 of *Lecture Notes in Artificial Intelligence*. Springer-Verlag, 2007. 53

[7] C. Baral, G. Greco, N. Leone, and G. Terracina, editors. *Proceedings of the Eighth International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'05)*, volume 3662 of *Lecture Notes in Artificial Intelligence*. Springer-Verlag, 2005. 52, 54

[8] A. Biere. PicoSAT essentials. *Journal on Satisfiability, Boolean Modeling and Computation*, 4:75–97, 2008. 45

[9] A. Biere, M. Heule, H. van Maaren, and T. Walsh, editors. *Handbook of Satisfiability*. IOS Press. To appear. 31, 34

[10] E. Dantsin, T. Eiter, G. Gottlob, and A. Voronkov. Complexity and expressive power of logic programming. *ACM Computing Surveys*, 33(3):374–425, 2001. 12

[11] M. Davis, G. Logemann, and D. Loveland. A machine program for theorem-proving. *Communications of the ACM*, 5:394–397, 1962. 12

[12] M. Davis and H. Putnam. A computing procedure for quantification theory. *Journal of the ACM*, 7:201–215, 1960. 12

[13] Satisfiability suggested format. DIMACS Center for Discrete Mathematics and Theoretical Computer Science, 1993. `ftp://dimacs.rutgers.edu/pub/challenge/satisfiability/doc/`. 31, 43

[14] W. Dowling and J. Gallier. Linear-time algorithms for testing the satisfiability of propositional Horn formulae. *Journal of Logic Programming*, 1:267–284, 1984. 10

[15] C. Drescher, M. Gebser, T. Grote, B. Kaufmann, A. König, M. Ostrowski, and T. Schaub. Conflict-driven disjunctive answer set solving. In G. Brewka and J. Lang, editors, *Proceedings of the Eleventh International Conference on Principles of Knowledge Representation and Reasoning (KR'08)*, pages 422–432. AAAI Press, 2008. 17, 48, 49

[16] N. Eén. Satzoo. `http://een.se/niklas/Satzoo/`, 2003. 45

[17] N. Eén and A. Biere. Effective preprocessing in SAT through variable and clause elimination. In F. Bacchus and T. Walsh, editors, *Proceedings of the Eigth International Conference on Theory and Applications of Satisfiability Testing (SAT'05)*, volume 3569 of *Lecture Notes in Computer Science*, pages 61–75. Springer-Verlag, 2005. 44, 45

[18] N. Eén and N. Sörensson. An extensible SAT-solver. In E. Giunchiglia and A. Tacchella, editors, *Proceedings of the Sixth International Conference on Theory and Applications of Satisfiability Testing (SAT'03)*, volume 2919 of *Lecture Notes in Computer Science*, pages 502–518. Springer-Verlag, 2004. 45

[19] T. Eiter and G. Gottlob. On the computational cost of disjunctive logic programming: Propositional case. *Annals of Mathematics and Artificial Intelligence*, 15(3-4):289–323, 1995. 17, 49

[20] T. Eiter and A. Polleres. Towards automated integration of guess and check programs in answer set programming: a meta-interpreter and applications. *Theory and Practice of Logic Programming*, 6(1-2):23–60, 2006. 17

[21] E. Erdem. The blocks world. `http://people.sabanciuniv.edu/esraerdem/ASP-benchmarks/bw.html`. 36

[22] E. Erdem and V. Lifschitz. Tight logic programs. *Theory and Practice of Logic Programming*, 3(4-5):499–518, 2003. 34

[23] F. Fages. Consistency of Clark's completion and the existence of stable models. *Journal of Methods of Logic in Computer Science*, 1:51–60, 1994. 34

[24] P. Ferraris. Answer sets for propositional theories. In Baral et al. [7], pages 119–131. 9, 22, 49

[25] P. Ferraris and V. Lifschitz. Weight constraints as nested expressions. *Theory and Practice of Logic Programming*, 5(1-2):45–74, 2005. 10, 22

[26] J. Freeman. *Improvements to propositional satisfiability search algorithms*. PhD thesis, University of Pennsylvania, 1995. 44

[27] M. Gebser, T. Janhunen, M. Ostrowski, T. Schaub, and S. Thiele. A versatile intermediate language for answer set programming: Syntax proposal. Unpublished draft, 2008. `http://www.cs.uni-potsdam.de/wv/pdfformat/gejaosscth08a.pdf`. 40, 48, 50

[28] M. Gebser, R. Kaminski, B. Kaufmann, M. Ostrowski, T. Schaub, and S. Thiele. Engineering an incremental ASP solver. In M. Garcia de la Banda and E. Pontelli, editors, *Proceedings of the Twenty-fourth International Conference on Logic Programming (ICLP'08)*, volume 5366 of *Lecture Notes in Computer Science*. Springer-Verlag, 2008. 7, 30, 36, 38

[29] M. Gebser, B. Kaufmann, A. Neumann, and T. Schaub. clasp: A conflict-driven answer set solver. In Baral et al. [6], pages 260–265. 6, 15, 17, 30, 42

[30] M. Gebser, B. Kaufmann, A. Neumann, and T. Schaub. Conflict-driven answer set enumeration. In Baral et al. [6], pages 136–148. 43, 45

[31] M. Gebser, B. Kaufmann, A. Neumann, and T. Schaub. Conflict-driven answer set solving. In M. Veloso, editor, *Proceedings of the Twentieth International Joint Conference on Artificial Intelligence (IJCAI'07)*, pages 386–392. AAAI Press/MIT Press, 2007. 46

[32] M. Gebser, B. Kaufmann, A. Neumann, and T. Schaub. Advanced preprocessing for answer set solving. In M. Ghallab, C. Spyropoulos, N. Fakotakis, and N. Avouris, editors, *Proceedings of the Eighteenth European Conference on Artificial Intelligence (ECAI'08)*, pages 15–19. IOS Press, 2008. 44

[33] M. Gebser, T. Schaub, and S. Thiele. GrinGo: A new grounder for answer set programming. In Baral et al. [6], pages 266–271. 6, 12, 15, 40, 56

[34] M. Gelfond and N. Leone. Logic programming and knowledge representation — the A-Prolog perspective. *Artificial Intelligence*, 138(1-2):3–38, 2002. 5

[35] M. Gelfond and V. Lifschitz. The stable model semantics for logic programming. In R. Kowalski and K. Bowen, editors, *Proceedings of the Fifth International Conference on Logic Programming (ICLP'88)*, pages 1070–1080. MIT Press, 1988. 9

[36] M. Gelfond and V. Lifschitz. Classical negation in logic programs and disjunctive databases. *New Generation Computing*, 9:365–385, 1991. 9, 16

[37] E. Giunchiglia, Y. Lierler, and M. Maratea. Answer set programming based on propositional satisfiability. *Journal of Automated Reasoning*, 36(4):345–377, 2006. 6, 17

[38] GNU coding standards. Free Software Foundation, Inc. `http://www.gnu.org/prep/standards/standards.html`. 39

[39] GNU general public license. Free Software Foundation, Inc. `http://www.gnu.org/copyleft/gpl.html`. 4

[40] E. Goldberg and Y. Novikov. BerkMin: A fast and robust SAT solver. In *Proceedings of the Fifth Conference on Design, Automation and Test in Europe (DATE'02)*, pages 142–149. IEEE Press, 2002. 44

[41] N. Gupta and D. Nau. On the complexity of blocks-world planning. *Artificial Intelligence*, 56(2-3):223–254, 1992. 36

[42] T. Janhunen, I. Niemelä, D. Seipel, P. Simons, and J. You. Unfolding partiality and disjunctions in stable model semantics. *ACM Transactions on Computational Logic*, 7(1):1–37, 2006. 17

[43] N. Leone, G. Pfeifer, W. Faber, T. Eiter, G. Gottlob, S. Perri, and F. Scarcello. The DLV system for knowledge representation and reasoning. *ACM Transactions on Computational Logic*, 7(3):499–562, 2006. 6, 12, 17

[44] Y. Lierler. cmodels – SAT-based disjunctive answer set solver. In Baral et al. [7], pages 447–451. 17

[45] V. Lifschitz. Answer set programming and plan generation. *Artificial Intelligence*, 138(1-2):39–54, 2002. 5, 32, 34

[46] V. Lifschitz and H. Turner. Splitting a logic program. In P. Van Hentenryck, editor, *Proceedings of the Eleventh International Conference on Logic Programming (ICLP'94)*, pages 23–37. MIT Press, 1994. 15

[47] F. Lin and Y. Zhao. ASSAT: computing answer sets of a logic program by SAT solvers. *Artificial Intelligence*, 157(1-2):115–137, 2004. 6, 17, 46

[48] J. Lloyd. *Foundations of Logic Programming*. Springer-Verlag, 1987. 8

[49] M. Luby, A. Sinclair, and D. Zuckerman. Optimal speedup of Las Vegas algorithms. *Information Processing Letters*, 47(4):173–180, 1993. 45, 46

[50] V. Marek and M. Truszczyński. Stable models and an alternative logic programming paradigm. In K. Apt, W. Marek, M. Truszczyński, and D. Warren, editors, *The Logic Programming Paradigm: a 25-Year Perspective*, pages 375–398. Springer-Verlag, 1999. 5, 8, 31

[51] J. Marques-Silva and K. Sakallah. GRASP: A search algorithm for propositional satisfiability. *IEEE Transactions on Computers*, 48(5):506–521, 1999. 12

[52] V. Mellarkod and M. Gelfond. Integrating answer set reasoning with constraint solving techniques. In J. Garrigue and M. Hermenegildo, editors, *Proceedings of the Ninth International Symposium of Functional and Logic Programming (FLOPS'08)*, volume 4989 of *Lecture Notes in Computer Science*, pages 15–31. Springer-Verlag, 2008. 50

[53] J. Minker, editor. *Foundations of Deductive Databases and Logic Programming*. Morgan Kaufmann Publishers, 1988. 14, 51

[54] D. Mitchell. A SAT solver primer. *Bulletin of the European Association for Theoretical Computer Science*, 85:112–133, 2005. 12

[55] M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an efficient SAT solver. In *Proceedings of the Thirty-eighth Conference on Design Automation (DAC'01)*, pages 530–535. ACM Press, 2001. 12, 44

[56] I. Niemelä. Logic programs with stable model semantics as a constraint programming paradigm. *Annals of Mathematics and Artificial Intelligence*, 25(3-4):241–273, 1999. 5, 8, 31

[57] R. Nieuwenhuis, A. Oliveras, and C. Tinelli. Solving SAT and SAT modulo theories: From an abstract Davis-Putnam-Logemann-Loveland procedure to DPLL(T). *Journal of the ACM*, 53(6):937–977, 2006. 50

[58] U. Nilsson and J. Małuszyński. *Logic, Programming and Prolog*. John Wiley & sons, 1995. 5

[59] C. Papadimitriou. *Computational Complexity*. Addison-Wesley, 1994. 31, 33, 49

[60] K. Pipatsrisawat and A. Darwiche. A lightweight component caching scheme for satisability solvers. In J. Marques-Silva and K. Sakallah, editors, *Proceedings of the Tenth International Conference on Theory and Applications of Satisfiability Testing (SAT'07)*, volume 4501 of *Lecture Notes in Computer Science*, pages 294–299. Springer-Verlag, 2007. 45

[61] Potsdam answer set solving collection. University of Potsdam. `http://potassco.sourceforge.net/`. 1, 4, 5

[62] F. Rossi, P. van Beek, and T. Walsh, editors. *Handbook of Constraint Programming*. Elsevier, 2006. 31, 34, 50

[63] L. Ryan. Efficient algorithms for clause-learning SAT solvers. Master's thesis, Simon Fraser University, 2004. 44, 46

[64] V. Ryvchin and O. Strichman. Local restarts. In H. Kleine Büning and X. Zhao, editors, *Proceedings of the Eleventh International Conference on Theory and Applications of Satisfiability Testing (SAT'08)*, volume 4996 of *Lecture Notes in Computer Science*, pages 271–276. Springer-Verlag, 2008. 45

[65] J. Schlipf. The expressive powers of the logic programming semantics. *Journal of Computer and System Sciences*, 51:64–86, 1995. 8, 31

[66] P. Simons, I. Niemelä, and T. Soininen. Extending and implementing the stable model semantics. *Artificial Intelligence*, 138(1-2):181–234, 2002. 4, 6, 10, 17, 22, 24, 27, 41, 43, 44, 49

[67] N. Sörensson and N. Eén. MiniSat v1.13 – a SAT solver with conflict-clause minimization. `http://minisat.se/downloads/MiniSat_v1.13_short.ps.gz`, 2005. 45

[68] T. Syrjänen. Lparse 1.0 user's manual. `http://www.tcs.hut.fi/Software/smodels/lparse.ps.gz`. 4, 6, 12, 22, 24, 27, 30, 36, 40, 42, 48, 50, 56

[69] T. Syrjänen. Omega-restricted logic programs. In T. Eiter, W. Faber, and M. Truszczyński, editors, *Proceedings of the Sixth International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'01)*, volume 2173 of *Lecture Notes in Computer Science*, pages 267–279. Springer-Verlag, 2001. 12, 56

[70] A. Van Gelder, K. Ross, and J. Schlipf. The well-founded semantics for general logic programs. *Journal of the ACM*, 38(3):620–650, 1991. 46

[71] J. Ward and J. Schlipf. Answer set programming with clause learning. In V. Lifschitz and I. Niemelä, editors, *Proceedings of the Seventh International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'04)*, volume 2923 of *Lecture Notes in Artificial Intelligence*, pages 302–313. Springer-Verlag, 2004. 7, 17

A Differences to the Language of `lparse`

We below provide a (most likely incomplete) list of differences between the input languages of `gringo` [33] and `lparse` [68]. First of all, it is important to note that the language of `gringo` significantly extends the one of `lparse`. For instance, `min` and `max` aggregates (cf. Section 4.1.10) are not supported by `lparse`. Furthermore, `gringo` provides full support for variables within compound terms (over uninterpreted functions with non-zero arity). That is, an atom like `p(f(X))` in the positive body of a rule can potentially be used to bind `X` (cf. Section 3.1), while `lparse` would treat `f(X)` like an arithmetic function (cf. Section 4.1.4) whose variables cannot be bound. Finally, `gringo` deals with level-restricted programs (cf. Section 3.1), while `lparse` requires programs to be ω-restricted [69]. As the latter are more restrictive, programs for `lparse` tend to be more verbose than the ones for `gringo`. Thus, we do not suggest writing programs in the input language of `gringo` for compatibility to `lparse`.

However, a bulk of existing encodings are written for `lparse` (see, e.g., [4]), and `gringo` (likewise, `clingo` and `iclingo`) should actually be able to deal with most of them. If this is not case, one of the following might be the reason:

- The input contains primed atoms like `p'`, which are (currently) not supported by `gringo`.
- Symbolic names are used for built-in constructs, e.g., `plus` or `eq` for built-in arithmetic function + or predicate ==, respectively. With a few exceptions (e.g., `abs` introduced in Section 4.1.4), such names are not associated to built-in constructs by `gringo`, as they may accidentally clash with users' names otherwise.
- The input contains (or is instantiated to) a `count` aggregate (or a cardinality constraints, respectively) containing duplicates of literals. Such duplicates are not removed by `lparse`, e.g., it treats `2{p(c),p(c)}` as a synonym for `2[p(c)=1,p(c)=1]`. In contrast, `gringo` associates curly brackets to a set (and square brackets to a multiset), as described in Section 4.1.10, so that `2{p(c),p(c)}` is the same as `2{p(c)}`, where the latter can clearly not hold.
- Pooling is expanded differently, e.g., `lparse` interprets `p(X,Y;X,Z)` as a shorthand for `p(X,Y),p(X,Z)`, while `gringo` expands it to `p(X,Y,Z), p(X,X,Z)`, as explained in Section 4.1.9.
- The input contains some (rather unusual) meta-statement, e.g., `#external`, not supported by `gringo`.

As indicated above, the provided list is probably incomplete. If you would like some difference(s) to be added, please contact the authors of this guide.