Abstract

This document provides an introduction to the Answer Set Programming (ASP) tools gringo, clasp, and clingo, developed at the University of Potsdam. The basic idea of ASP is to express a problem in the form of a logic program so that its logical models, called answer sets, provide the solutions to the original problem. The first tool, gringo, is a so-called grounder translating user-provided logic programs (with variables) into equivalent propositional logic programs (without variables). The second tool, clasp, is a so-called solver computing the answer sets of the propositional programs issued by gringo. The third tool, clingo, combines the functionalities of gringo and clasp, and additionally integrates the scripting languages lua and python either through libraries or embedded code. This guide, for one, aims at enabling ASP novices to make use of the aforementioned tools. For another, it provides a reference of the tools’ features that ASP adepts might be tempted to exploit.

The guide generally refers to gringo and clingo series 4 as well as clasp series 3; full conformity is obtained with gringo 4.5, clingo 4.5 and clasp 3.1. Please make sure that you have corresponding (or later) versions available.

This document includes many illustrative examples. For convenience, they can be saved to disk by clicking their file names. Depending on the viewer, a right or double-click should initiate saving.
## Contents

1 Introduction 9
  1.1 Download and Installation 9
  1.2 Outline 10

2 Quickstart 12
  2.1 Problem Instance 12
  2.2 Problem Encoding 13
  2.3 Problem Solution 15
  2.4 Summary 16

3 Input Languages 17
  3.1 Input Language of gringo and clingo 17
    3.1.1 Terms 17
    3.1.2 Normal Programs and Integrity Constraints 19
    3.1.3 Classical Negation 22
    3.1.4 Disjunction 22
    3.1.5 Double Negation and Head Literals 23
    3.1.6 Boolean Constants 24
    3.1.7 Built-in Arithmetic Functions 24
    3.1.8 Built-in Comparison Predicates 25
    3.1.9 Intervals 27
    3.1.10 Pooling 28
    3.1.11 Conditions and Conditional Literals 28
    3.1.12 Aggregates 30
    3.1.13 Optimization 37
    3.1.14 External Functions 39
    3.1.15 Meta-Statements 43
  3.2 Input Language of clasp 47

4 Multi-shot Solving 49

5 Examples 50
  5.1 n-Coloring 50
    5.1.1 Problem Instance 50
    5.1.2 Problem Encoding 50
    5.1.3 Problem Solution 51
  5.2 Traveling Salesperson 52
    5.2.1 Problem Instance 52
    5.2.2 Problem Encoding 53
    5.2.3 Problem Solution 54
  5.3 Blocks World Planning 55
    5.3.1 Problem Instance 56
## Contents

5.3.2 Problem Encoding ........................................ 57  
5.3.3 Problem Solution ......................................... 58  

6 Command Line Options ...................................... 60  
6.1 gringo Options ............................................... 60  
6.2 clingo Options ............................................... 61  
6.3 clasp Options ................................................. 62  
6.3.1 General Options .......................................... 62  
6.3.2 Solving Options .......................................... 63  
6.3.3 Fine-Tuning Options ..................................... 65  

7 Errors, Warnings, and Infos ......................... 68  
7.1 Errors ........................................................ 68  
7.1.1 Parsing Command Line Options ...................... 68  
7.1.2 Parsing and Checking Logic Programs ............ 69  
7.1.3 Parsing Logic Programs in smodels Format ...... 70  
7.1.4 Multi-shot Solving ....................................... 71  
7.2 Warnings ..................................................... 71  
7.2.1 File Included Multiple Times ....................... 71  
7.2.2 Unbounded CSP Variables ............................ 72  
7.3 Infos ......................................................... 72  
7.3.1 Undefined Operations ................................. 72  
7.3.2 Undefined Atoms ........................................ 72  

8 Meta-Programming ......................................... 74  

9 Heuristic-driven Solving .................................. 75  
9.1 Heuristic Programming .................................. 75  
9.1.1 Heuristic modifier sign ................................ 75  
9.1.2 Showing heuristic information ..................... 76  
9.1.3 Heuristic modifier level ............................. 76  
9.1.4 Dynamic heuristic modifications .................. 77  
9.1.5 Heuristic modifiers true and false ............... 77  
9.1.6 Priorities among heuristic modifications ....... 78  
9.1.7 Heuristic modifiers init and factor ............... 79  
9.1.8 Monitoring domain choices ......................... 79  
9.1.9 Heuristics for Blocks World Planning ........... 79  
9.2 Command Line Structure-oriented Heuristics ..... 82  
9.3 Computing Subset Minimal Answer Sets with Heuristics .... 82  

10 Optimization and Preference Handling ........ 84  
10.1 Multi-objective Optimization with clasp and clingo ........................................ 84  
10.2 Preference Handling with asprin .................... 84  
10.2.1 Computing optimal answer sets ................. 84
List of Figures

10.2.2 Computing multiple optimal answer sets 86
10.2.3 Input language of asprin 87
10.2.4 Preference relations and preference types 90
10.2.5 asprin library 91
10.2.6 Implementing preference types 95

11 Constraint Programming 101
  11.1 ASP modulo CSP solving with clingcon 101
  11.2 Solving CSPs with aspartame 101
  11.3 Constraint Programming with gringo 101

12 Solver Configuration 103
  12.1 Portfolio-Solving with claspfolio 103
  12.2 Problem-oriented Configuration of clasp with piclasp 104

13 Future Work 107

A Complementary Resources 108

B Differences to the Language of gringo 3 109

C Differences to the Language of iclingo and oclingo 110

References 111

Index 118

List of Figures

1 Towers of Hanoi: Initial and Goal Situation. 12
2 Grammar for Terms. 18
3 A Directed Graph with 6 Nodes and 17 Edges. 51
4 A 3-Coloring for the Graph in Figure 3. 52
5 The Graph from Figure 3 along with Edge Costs. 53
6 A Minimum-cost Round Trip. 54

Listings

examples/toh_ins.lp 12
examples/toh_enc.lp 13
examples/fly.lp 20
examples/bird.lp 21
examples/flycn.lp 22
examples/flynn.lp 23
examples/bool.lp ........................................ 24
examples/arithf.lp .................................... 24
examples/arithc.lp .................................... 25
examples/symbc.lp .................................... 25
examples/define.lp .................................... 26
examples/unify.lp .................................... 26
examples/int.lp ....................................... 27
examples/pool.lp ...................................... 28
examples/cond.lp ...................................... 29
examples/sort.lp ...................................... 29
examples/aggr.lp ...................................... 35
examples/opt.lp ....................................... 38
examples/gcd–lua.lp .................................. 40
examples/gcd–py.lp .................................... 40
examples/gcd.lp ....................................... 40
examples/rng–lua.lp ................................... 40
examples/rng–py.lp .................................... 40
examples/rng.lp ....................................... 40
examples/term–lua.lp .................................. 41
examples/term–py.lp ................................... 41
examples/term.lp ...................................... 42
examples/cmp–lua.lp ................................... 42
examples/cmp–py.lp .................................... 42
examples/cmp.lp ....................................... 42
examples/showa.lp .................................... 44
examples/showt.lp ..................................... 44
examples/const.lp ..................................... 44
examples/ext.lp ....................................... 45
examples/part.lp ...................................... 46
examples/part–lua.lp .................................. 46
examples/part–py.lp ................................... 46
examples/include.lp ................................... 47
examples/graph.lp .................................... 50
examples/color.lp ..................................... 51
examples/costs.lp .................................... 52
examples/ham.lp ...................................... 53
examples/min.lp ...................................... 54
examples/world0.lp ................................... 56
examples/blocks.lp ................................... 57
examples/psign.lp ..................................... 75
examples/nsign.lp ..................................... 76
examples/level.lp ..................................... 76
examples/dynamic.lp .................................. 77
examples/priority.lp ................................... 78
1 Introduction

The “Potsdam Answer Set Solving Collection” (Potassco; [28, 32, 72]) gathers a variety of tools for Answer Set Programming (ASP; [2, 6, 12, 47, 48, 49, 61, 66, 68]), including grounder gringo, solver clasp, and their combination within the integrated ASP system clingo. Their common goal is to enable users to rapidly solve computationally difficult problems in ASP, a declarative programming paradigm based on logic programs and their answer sets.

This guide, for one, aims at enabling ASP novices to make use of the aforementioned tools. For another, it provides a reference of the tools’ features that ASP adepts might be tempted to exploit. A formal introduction to (a large fragment of) the input language of gringo (and clingo) and its precise semantics is given in [26]. The foundations and algorithms underlying the grounding and solving technology used in gringo and clasp is described in detail in [32]. For further aspects of ASP we refer the interested reader to the literature [6, 48].

In fact, we focus in this guide on ASP and thus the computation of answer sets of a logic program [50]. Moreover, clasp can be used as a full-fledged SAT, MaxSAT, or PB solver (see [10]), accepting propositional CNF formulas in (extended) DIMACS format as well as PB formulas in OPB and WBO format.

1.1 Download and Installation

The Potassco tools gringo, clasp, and clingo are written in C++ and published under the GNU General Public License [54]. Source packages as well as precompiled binaries for Linux, MacOS, and Windows are available at [72]. For building the tools from sources, please download the most recent source package, consult the included README file, and make sure that the machine to build on has all required software installed. If you still encounter problems in the building process, please consult the support pages at [72] or use the Potassco mailing list: potassco-users@lists.sourceforge.net.

An alternative way to install the tools is to use a package manager. Currently, packages and ports are available for Debian, Ubuntu, Arch Linux (AUR), and for MacOS X (via Homebrew or MacPorts). Note that packages installed this way are not always up to date; the latest versions are available at our Sourceforge page at [72].

Afterward, 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), you can invoke the following two commands:

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

Note that gringo, clasp, and clingo run on the command line (Linux shell, Windows command prompt, or the like). To invoke them, their binaries can be “in-
stalled” simply by putting them into some directory in the system path. In an invocation, one usually provides the file names of input (text) files as arguments to either gringo or clingo, while the output of gringo is typically piped into clasp. Thus, the standard invocation schemes are as follows:

\[
\text{gringo [ options | files ] | clasp [ options | number ]}
\]

\[
\text{clingo [ options | files | number ]}
\]

A numerical argument provided to either clasp or clingo determines the maximum number of answer sets to be computed, where 0 means “compute all answer sets”. By default, only one answer set is computed (if it exists).

### 1.2 Outline

This guide introduces the fundamentals of using gringo, clasp, and clingo. In particular, it aims at enabling the reader to benefit from them by significantly reducing the “time to solution” on difficult computational problems. To this end, Section 2 provides an introductory example that serves both as a prototype of problem modeling using logic programs and also as an appetizer of the modeling language of gringo. The main part of this document, Section 3, is dedicated to the input languages of our tools, where Section 3.1 details the joint input language of gringo and clingo, while solver formats supported by clasp are not supposed to be written directly by a user and just briefly described in Section 3.2. Then, the control capacities of clingo needed for multi-shot solving are detailed in Section 4. For further illustration, Section 5 describes how three well-known example problems can be solved with our tools. 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. The following sections address adept extensions of the basic modeling language and control capacities. In particular, Section 8 elaborates meta-programming functionalities that allow for reinterpreting logic programs by means of ASP. Techniques for incorporating domain-specific heuristics into the ASP solving process are presented in Section 9. Section 10 is dedicated to advanced methods for preference handling and optimization. Moreover, Section 11 provides concepts developed particularly for dealing with multi-valued variables and quantitative constraints. In order to tune efficiency, Section 12 further introduces principled approaches to solver configuration. Finally, we conclude with a summary in Section 13.

For readers familiar with the gringo 3 series, Appendix B lists the most prominent differences to the current series. Otherwise, gringo and clingo series 4 should accept most inputs recognized by gringo 3 (and the seminal grounder lparse [78]). The input of solver clasp can be generated by all versions of gringo (as well as lparse). Be aware that there are some syntactic and semantic changes between the language of the series 3 and 4, so already existing encodings have to be adapted to be used with series 4. Throughout this document, we provide

\[\text{1A grounder that constitutes the traditional front-end of solver smodels [76]}\]
illustrative examples. Many of them can actually be run. You find instructions on how to accomplish this (or sometimes meta-remarks) in margin boxes, like the one on the right. Occurrences of ‘\’ usually mean that text in a command line, broken for space reasons, is actually continuous.

After all these preliminaries, it is time to start our guided tour through the main Potassco [72] tools. We hope that you will find it enjoyable and helpful!
2 Quickstart

As an introductory example, we consider a simple Towers of Hanoi puzzle, consisting of three pegs and four disks of different size. As shown in Figure 1, the goal is to move all disks from the left peg to the right one, where only the topmost disk of a peg can be moved at a time. Furthermore, a disk cannot be moved to a peg already containing some disk that is smaller. Although there is an efficient algorithm to solve our simple Towers of Hanoi puzzle, we do not exploit it and below merely specify conditions for sequences of moves being solutions.

In ASP, it is custom to provide a uniform problem definition [66, 68, 75]. Following this methodology, we separately specify an instance and an encoding (applying to every instance) of the following problem: given an initial placement of the disks, a goal situation, and a number \( n \), decide whether there is a sequence of \( n \) moves that achieves the goal. We will see that this problem can be elegantly described in ASP and solved by domain-independent tools like gringo and clasp. Such a declarative solution is now exemplified.

2.1 Problem Instance

We describe the pegs and disks of a Towers of Hanoi puzzle via facts over the predicates \( \text{peg}/1 \) and \( \text{disk}/1 \) (the number denotes the arity of the predicate). Disks are numbered by consecutive integers starting at 1, where a disk with a smaller number is considered to be bigger than a disk with a greater number. The names of the pegs can be arbitrary; in our case, we use \( a \), \( b \), and \( c \). Furthermore, the predicates \( \text{init}_\text{on}/2 \) and \( \text{goal}_\text{on}/2 \) describe the initial and the goal situation, respectively. Their arguments, the number of a disk and the name of a peg, determine the location of a disk in the respective situation. Finally, the predicate \( \text{moves}/1 \) specifies the number of moves in which the goal must be achieved. When allowing 15 moves, the Towers of Hanoi puzzle shown in Figure 1 is described by the following facts:

1. \( \text{peg}(a;b;c). \)
2. \( \text{disk}(1..4). \)
3. \( \text{init}_\text{on}(1..4,a). \)
4. \( \text{goal}_\text{on}(1..4,c). \)
5. \( \text{moves}(15). \)
Note that the ‘;’ in the first line is syntactic sugar (detailed in Section 3.1.10) that expands the statement into three facts: \( \text{peg}(a) \), \( \text{peg}(b) \), and \( \text{peg}(c) \). Similarly, ‘1..4’ used in Line 2–4 refers to an interval (detailed in Section 3.1.9). Here, it abbreviates distinct facts over four values: 1, 2, 3, and 4. In summary, the facts in Line 1–5 describe the Towers of Hanoi puzzle in Figure 1 along with the requirement that the goal ought to be achieved within 15 moves.

2.2 Problem Encoding

We now proceed by encoding Towers of Hanoi via schematic rules, i.e., rules containing variables (whose names start with uppercase letters) that are independent of a particular instance. Typically, an encoding can be logically partitioned into a Generate, a Define, and a Test part [61]. An additional Display part allows for restricting the output to a distinguished set of atoms, and thus, for suppressing auxiliary predicates. We follow this methodology and mark the respective parts via comment lines beginning with ‘%’ in the following encoding:

1  % Generate
2  \{ move(D,P,T) : disk(D), peg(P) \} = 1 :- moves(M),
   T = 1..M.
3  % Define
4  move(D,T) :- move(D,_,T).
5  on(D,P,0) :- init_on(D,P).
6  on(D,P,T) :- move(D,P,T).
7  on(D,P,T+1) :- on(D,P,T), not move(D,T+1),
               not moves(T).
8  blocked(D-1,P,T+1) :- on(D,P,T), not moves(T).
9  blocked(D-1,P,T) :- blocked(D,P,T), disk(D).
10 % Test
11 :- move(D,P,T), blocked(D-1,P,T).
12 :- move(D,T), on(D,P,T-1), blocked(D,P,T).
13 :- goal_on(D,P), not on(D,P,M), moves(M).
14 :- \{ on(D,P,T) \} != 1, disk(D), moves(M), T = 1..M.
15 % Display
16 #show move/3.

Note that the variables \( D, P, T \), and \( M \) are used to refer to disks, pegs, the number of a move, and the length of the sequence of moves, respectively.

The Generate part, describing solution candidates, consists of the rule in Line 2. It expresses that, at each point \( T \) in time (other than 0), exactly one move of a disk \( D \) to some peg \( P \) must be executed. The head of the rule (left of ‘:-’) is a so-called cardinality constraint (see Section 3.1.12). It consists of a set of literals, expanded using the conditions behind the colon (detailed in Section 3.1.11), along with the guard ‘\( \neq 1 \)’. The cardinality constraint is satisfied if the number of true elements is equal to one, as specified by the guard. Since the cardinality constraint occurs
2.2 Problem Encoding

As the head of a rule, it allows for deriving ("guessing") atoms over the predicate move/3 to be true. In the body (right of ‘:-’), we define (detailed in Section 3.1.8), T = 1..M, to refer to each time point T from 1 to the maximum time point M. We have thus characterized all sequences of M moves as solution candidates for Towers of Hanoi. Up to now, we have not yet imposed any further conditions, e.g., that a bigger disk must not be moved on top of a smaller one.

The Define part in Line 4-9 contains rules defining auxiliary predicates, i.e., predicates that provide properties of a solution candidate at hand. (Such properties will be investigated in the Test part described below.) The rule in Line 4 simply projects moves to disks and time points. The resulting predicate move/2 can be used whenever the target peg is insignificant, so that one of its atoms actually subsumes three possible cases. Furthermore, the predicate on/3 captures the state of a Towers of Hanoi puzzle at each time point. To this end, the rule in Line 5 identifies the locations of disks at time point 0 with the initial state (given in an instance). State transitions are modeled by the rules in Line 6 and 7. While the former specifies the direct effect of a move at time point T, i.e., the affected disk D is relocated to the target peg P, the latter describes inertia: the location of a disk D carries forward from time point T to T+1 if D is not moved at T+1. Observe the usage of not moves(T) in Line 7, which prevents deriving disk locations beyond the maximum time point. Finally, we define the auxiliary predicate blocked/3 to indicate that a smaller disk, with a number greater than D-1, is located on a peg P. The rule in Line 8 derives this condition for time point T+1 from on(D,P,T), provided that T is not the maximum time point. The rule in Line 9 further propagates the status of being blocked to all bigger disks on the same peg. Note that we also mark D-1 = 0, not referring to any disk, as blocked, which is convenient for eliminating redundant moves in the Test part described next.

The Test part consists of the integrity constraints in Line 11-14, rules that eliminate unintended solution candidates. The first integrity constraint in Line 11 asserts that a disk D must not be moved to a peg P if D-1 is blocked at time point T. This excludes moves putting a bigger disk on top of a smaller one and, in view of the definition of blocked/3, also disallows that a disk is put back to its previous location. Similarly, the integrity constraint in Line 12 expresses that a disk D cannot be moved at time point T if it is blocked by some smaller disk on the same peg P. Note that we use move(D,T) here because the target of an illegal move does not matter in this context. The fact that the goal situation, given in an instance, must be achieved at maximum time point M is represented by the integrity constraint in Line 13. The final integrity constraint in Line 14 asserts that, for every disk D and time point T, there is exactly one peg P such that on(D,P,T) holds. Although this condition is implied by the definition of on/3 in Line 6 and 7 with respect to the moves in a solution, making such knowledge explicit via an integrity constraint turns out to improve the solving efficiency.

Finally, the meta-statement (detailed in Section 3.1.15) of the Display part in Line 16 indicates that only atoms over the predicate move/3 ought to be printed, thus suppressing the predicates used to describe an instance as well as the auxiliary
predicates move/2, on/3, and blocked/3. This is for more convenient reading of a solution, given that it is fully determined by atoms over move/3.

2.3 Problem Solution

We are now ready to solve our Towers of Hanoi puzzle. To compute an answer set representing a solution, invoke one of the following commands:

```
clingo toh_ins.lp toh_enc.lp
grringo toh_ins.lp toh_enc.lp | clasp
```

The output of the solver, clingo in this case, should look somehow like this:

```
clingo version 4.4.0
Reading from toh_ins.lp ...
Solving...
Answer: 1
move(4,b,1) move(3,c,2) move(4,c,3) move(2,b,4) \move(4,a,5) move(3,b,6) move(4,b,7) move(1,c,8) \move(4,c,9) move(3,a,10) move(4,a,11) move(2,c,12) \move(4,b,13) move(3,c,14) move(4,c,15)
SATISFIABLE
Models : 1+
Calls : 1
Time : 0.017s (Solving: 0.01s 1st Model: 0.01s \Unsat: 0.00s)
CPU Time : 0.010s
```

The first line shows the clingo version. The following two lines indicate clingo’s state. clingo should print immediately that it is reading. Once this is done, it prints Solving... to the command line. The Towers of Hanoi instance above is so easy to solve that you will not recognize the delay, but for larger problems it can be noticeable. The line starting with Answer: indicates that the (output) atoms of an answer set follow in the next line. In this example, it contains the true instances of move/3 in the order of time points, so that we can easily read off the following solution from them: first move disk 4 to peg b, second move disk 3 to peg c, third move disk 4 to peg c, and so on. We use ‘\’ to indicate that all atoms over move/3 actually belong to a single line. Note that the order in which atoms are printed does not bear any meaning (and the same applies to the order in which answer sets are found). Below this solution, we find the satisfiability status of the problem, which is reported as SATISFIABLE by the solver. The ‘1+’ in the line starting with Models tells us that one answer set has been found. Other possibilities include UNSATISFIABLE and UNKNOWN, the latter in case of an abort.

Calls to the solver are kept to a minimum. In this example, 1 answer set has been found. The remaining time is used to find the next model. The solver’s output is terminated as soon as an answer set has been found. The output shows that there is a single answer set. The time spent to find the answer set is 0.017 seconds. The CPU time is 0.010 seconds.

The given instance has just one solution. In fact, the ‘+’ from ‘1+’ disappears if you compute all solutions by invoking:

```
clingo toh_ins.lp \toh_enc.lp 0
or alternatively:

grringo toh_ins.lp \toh_enc.lp | clasp 0
```
solver are of interest in multi-shot solving (see Section 4). The final lines report statistics including total run-time (wall-clock Time as well as CPU Time) and the amount of time spent on search (Solving), along with the fractions taken to find the first solution (1st Model) and to prove unsatisfiability (Unsat). More information about available options, e.g., to obtain extended statistics output, can be found in Section 6.

2.4 Summary

To conclude our quickstart, let us summarize some “take-home messages”. For solving our Towers of Hanoi puzzle, we first provided facts representing an instance. Although we did not discuss the choice of predicates, an appropriate instance representation is already part of the modeling in ASP and not always as straightforward as here. Second, we provided an encoding of the problem applying to any instance. The encoding consisted of parts generating solution candidates, deriving their essential properties, testing that no solution condition is violated, and finally projecting the output to characteristic atoms. With the encoding at hand, we could use off-the-shelf ASP tools to solve our instance, and the encoding can be reused for any further instance that may arise in the future.

---

4 No unsatisfiability proof is done here, hence, this time is zero. But for example, when enumerating all models, this is the time spent between finding the last model and termination.
3 Input Languages

This section provides an overview of the input languages of grounder gringo, combined grounder and solver clingo, and solver clasp. The joint input language of gringo and clingo is detailed in Section 3.1. Finally, Section 3.2 is dedicated to the inputs handled by clasp.

3.1 Input Language of gringo and clingo

The tool gringo [46] is a grounder capable of transforming user-defined logic programs (usually containing variables) into equivalent ground (that is, variable-free) programs. The output of gringo can be piped into solver clasp [38, 43], 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.

Usually, logic programs are specified in one or more (text) files whose names are provided as arguments in an invocation of either gringo or clingo. In what follows, we describe the constructs belonging to the input language of gringo and clingo.

3.1.1 Terms

Every (non-propositional) logic program includes terms, mainly to specify the arguments of atoms (see below). The grammar for gringo’s (and clingo’s) terms is shown in Figure 2.

The basic building blocks are simple terms: integers, constants, strings, and variables as well as the tokens ‘_’, #sup, and #inf. An integer is represented by means of an arithmetic expression, further explained in Section 3.1.7. Constants and variables are distinguished by their first letters, which are lowercase and uppercase, respectively, where leading occurrences of ‘_’ are allowed (may be useful to circumvent name clashes). Furthermore, a string is an arbitrary sequence of characters enclosed in double quotes (“¨”), where any occurrences of ‘\’, newline, and double quote must be escaped via ‘\\’, ‘\n’, or ‘\”’, respectively.

While a constant or string represents itself, a variable is a placeholder for all variable-free terms in the language of a logic program. Unlike a variable name whose recurrences within a rule refer to the same variable, the token ‘_’ (not followed by any letter) stands for an anonymous variable that does not recur anywhere. (One can view this as if a new variable name is invented on each occurrence of ‘_’.) In addition, there are the special constants #sup and #inf representing the greatest and smallest element among all variable-free terms, respectively; we illustrate their use in Section 3.1.12.

5The set of all terms constructible from the available constants and function symbols is called Herbrand universe.

6Their is a total order defined on variable-free terms; for details see Section 3.1.8.
3.1 Input Language of gringo and clingo

Figure 2: Grammar for Terms.
Next, (uninterpreted) functions are complex terms composed of a name (like a constant) and one or more terms as arguments. For instance, \( \text{at}(\text{peter}, \text{time}(12), X) \) is a function with three arguments: constant \( \text{peter} \), another function \( \text{time}(12) \) with an integer argument, and variable \( X \). Finally, there are tuples, which are similar to functions but without a name. Examples for tuples are: the empty tuple () and the tuple \( (\text{at}, \text{peter}, \text{time}(12), X) \) with four elements. Tuples may optionally end in a comma ',' for distinguishing one-elementary tuples. That is, \( (t, ) \) is a one-elementary tuple, while a term of form \( (t) \) is equivalent to \( t \). For instance, \( (42, ) \) is a one-elementary tuple, whereas \( (42) \) is not, and the above quadruple is equivalent to \( (\text{at}, \text{peter}, \text{time}(12), X, ) \).

### 3.1.2 Normal Programs and Integrity Constraints

Rules of the following forms are admitted in a normal logic program (with integrity constraints):

- **Fact:** \( A_0 \).
- **Rule:** \( A_0 : - L_1, \ldots, L_n \).
- **Integrity Constraint:** \( : - L_1, \ldots, L_n \).

The head \( A_0 \) of a rule or a fact is an atom of the same syntactic form as a constant or function. In the body of a rule or an integrity constraint, every \( L_j \) for \( 1 \leq j \leq n \) is a literal of the form \( A \) or \( \text{not } A \), where \( A \) is an atom and the connective \( \text{not} \) denotes default negation. We say that a literal \( L \) is positive if it is an atom, and negative otherwise. While the head atom \( A_0 \) of a fact must unconditionally be true, the intuitive reading of a rule corresponds to an implication: if all positive literals in the rule’s body are true and all negative literals are satisfied, then \( A_0 \) must be true. On the other hand, an integrity constraint is a rule that filters solution candidates, meaning that the literals in its body must not jointly be satisfied.

A set of (propositional) atoms is called a model of a logic program if it satisfies all rules, facts, and integrity constraints. Atoms are considered true if and only if they are in the model. In ASP, a model is called an answer set if every atom in the model has an (acyclic) derivation from the program. See [50, 47, 62] for formal definitions of answer sets of logic programs.

To get the idea, let us consider some small examples.

**Example 3.1.** Consider the following logic program:

\[
\begin{align*}
a & : - b. \\
b & : - a.
\end{align*}
\]

When \( a \) and \( b \) are false, the bodies of both rules are false as well, so that the rules are satisfied. Furthermore, there is no (true) atom to be derived, which shows that the empty set is an answer set. On the other hand, if \( a \) is true but \( b \) is not, then the first rule is unsatisfied because the body holds but the head does not. Similarly, the second rule is unsatisfied if \( b \) is true and \( a \) is not. Hence, an answer set cannot
contain only one of the atoms \(a\) and \(b\). It remains to investigate the set including both \(a\) and \(b\). Although both rules are satisfied, \(a\) and \(b\) cannot be derived acyclically: \(a\) relies on \(b\), and vice versa. That is, the set including both \(a\) and \(b\) is not an answer set. Hence, the empty set is the only answer set of the logic program. We say that there is a positive cycle through \(a\) and \(b\) subject to minimization.

Consider the following logic program:

\[
\begin{align*}
a & :- \neg b. \\
b & :- \neg a.
\end{align*}
\]

Here, the empty set is not a model because both rules are unsatisfied. However, the sets containing only \(a\) or only \(b\) are models. To see that each of them is an answer set, note that \(a\) is derived by the rule \(a :- \neg b\). if \(b\) is false; similarly, \(b\) is derived by \(b :- \neg a\). if \(a\) is false. Note that the set including both \(a\) and \(b\) is not an answer set because neither atom can be derived if both are assumed to be true: the bodies of the rules \(a :- \neg b\). and \(b :- \neg a\). are false. Hence, we have that either \(a\) or \(b\) belongs to an answer set of the logic program.

To illustrate the use of facts and integrity constraints, let us augment the previous logic program:

\[
\begin{align*}
a & :- \neg b. \\
b & :- \neg a. \\
c & .
\end{align*}
\]

\[
:- c, \neg b.
\]

Since \(c\) is a fact, atom \(c\) must unconditionally be true, i.e., it belongs to every model. In view of this, the integrity constraint \(:- c, \neg b\). tells us that \(b\) must be true as well in order to prevent its body from being satisfied. However, this kind of reasoning does not provide us with a derivation of \(b\). Rather, we still need to make sure that the body of the rule \(b :- \neg a\). is satisfied, so that atom \(a\) must be false. Hence, the set containing \(b\) and \(c\) is the only answer set of our logic program.

In the above examples, we used propositional logic programs to exemplify the idea of an answer set: a model of a logic program such that all its true atoms are (acyclically) derivable. In practice, logic programs are typically non-propositional, i.e., they include schematic rules with variables. The next example illustrates this.

Example 3.2. Consider a child from the south pole watching cartoons, where it sees a yellow bird that is not a penguin. The child knows that penguins can definitely not fly (due to small wingspread), but it is unsure about whether the yellow bird flies. This knowledge is generalized by the following schematic rules:

\[
\begin{align*}
1 \quad fly(X) & :- bird(X), \neg neg\_fly(X). \\
2 \quad neg\_fly(X) & :- bird(X), \neg fly(X). \\
3 \quad neg\_fly(X) & :- penguin(X).
\end{align*}
\]

The first rule expresses that it is generally possible that a bird flies, unless the contrary, subject to the second rule, is the case. The definite knowledge that penguins cannot fly is specified by the third rule.
Later on, the child learns that the yellow bird is a chicken called “tweety”, while its favorite penguin is called “tux”. The knowledge about these two individuals is represented by the following facts:

4. bird(tweety).  chicken(tweety).
5. bird(tux).  penguin(tux).

When we instantiate the variable \(X\) in the three schematic rules with \(tweety\) and \(tux\), we obtain the following ground rules:

\[
\begin{align*}
\text{fly}(tweety) & :- \text{bird}(tweety), \neg \text{neg\_fly}(tweety). \\
\text{fly}(tux) & :- \text{bird}(tux), \neg \text{neg\_fly}(tux). \\
\text{neg\_fly}(tweety) & :- \text{bird}(tweety), \neg \text{fly}(tweety). \\
\text{neg\_fly}(tux) & :- \text{bird}(tux), \neg \text{fly}(tux). \\
\text{neg\_fly}(tweety) & :- \text{penguin}(tweety). \\
\text{neg\_fly}(tux) & :- \text{penguin}(tux). \\
\end{align*}
\]

Further taking into account that \(tweety\) and \(tux\) are known to be birds, that \(tux\) is a penguin, while \(tweety\) is not, and that penguins can definitely not fly, we can simplify the previous ground rules to obtain the following ones:

\[
\begin{align*}
\text{fly}(tweety) & :- \neg \text{neg\_fly}(tweety). \\
\text{neg\_fly}(tweety) & :- \neg \text{fly}(tweety). \\
\text{neg\_fly}(tux) & :- \text{penguin}(tux). \\
\end{align*}
\]

Now it becomes apparent that \(tweety\) may fly or not, while \(tux\) surely does not fly. Thus, there are two answer sets for the three schematic rules above, instantiated with \(tweety\) and \(tux\).

The above example illustrated how variables are used to represent all instances of rules with respect to the language of a logic program. In fact, grounder gringo (or the grounding component of clingod) takes care of instantiating variables such that an equivalent propositional logic program is obtained. To this end, rules are required to be \textit{safe}, i.e., all variables in a rule must occur in some positive literal (a literal not preceded by \(\neg\)) in the body of the rule. For instance, the first two schematic rules in Example 3.2 are safe because they include \textit{bird}(X) in their positive bodies. This tells gringo (or clingod) that the values to be substituted for \(X\) are limited to birds.

Up to now, we have introduced terms, facts, (normal) rules, and integrity constraints. Before we proceed to describe handy extensions to this simple core language, keep in mind that the role of a rule (or fact) is that an atom in the head can be derived to be true if the body is satisfied. Unlike this, an integrity constraint implements a test, but it cannot be used to derive any atom. This universal meaning still applies when more sophisticated language constructs, as described in the following, are used.
3.1.3 Classical Negation

The connective \texttt{not} expresses default negation, i.e., a literal \texttt{not A} is assumed to hold unless atom \texttt{A} is derived to be true. In contrast, the classical (or strong) negation of an atom \cite{51} holds only if it can be derived. Classical negation, indicated by symbol \texttt{\textasciitilde}, is permitted in front of atoms. That is, if \texttt{A} is an atom, then \texttt{\textasciitilde A} is an atom representing the complement of \texttt{A}. The semantic relationship between \texttt{A} and \texttt{\textasciitilde A} is simply that they must not jointly hold. Hence, classical negation can be understood as a syntactic feature allowing us to impose an integrity constraint \texttt{:- A, \textasciitilde A} without explicitly writing it in a logic program. Depending on the logic program at hand, it may be possible that neither \texttt{A} nor \texttt{\textasciitilde A} is contained in an answer set, thus representing a state where the truth and the falsity of \texttt{A} are both unknown.

\textbf{Example 3.3.} Using classical negation, we can rewrite the schematic rules in Example 3.2 in the following way:

1. \texttt{fly(X) :- bird(X), not \textasciitilde fly(X).}
2. \texttt{\textasciitilde fly(X) :- bird(X), not fly(X).}
3. \texttt{\textasciitilde fly(X) :- penguin(X).}

Given the individuals \texttt{tweety} and \texttt{tux}, classical negation is reflected by the following (implicit) integrity constraints:

4. \texttt{:- fly(tweety), \textasciitilde fly(tweety).}
5. \texttt{:- fly(tux), \textasciitilde fly(tux).}

There are still two answer sets, containing \texttt{\textasciitilde fly(tux)} and either \texttt{fly(tweety)} or \texttt{\textasciitilde fly(tweety)}.

Now assume that we add the following fact to the program:

\texttt{fly(tux).}

Then, \texttt{fly(tux)} must unconditionally be true, and \texttt{\textasciitilde fly(tux)} is still derived by an instance of the third schematic rule. Since every answer set candidate containing both \texttt{fly(tux)} and \texttt{\textasciitilde fly(tux)} triggers the (implicit) integrity constraint in Line 5, there is no longer any answer set. 

3.1.4 Disjunction

Disjunctive logic programs permit connective \texttt{\textasciitilde;\textasciitilde} between atoms in rule heads.\footnote{Note that disjunction in rule heads was not supported by \texttt{clasp} and \texttt{clingo} versions before series 3 and 4, respectively.}

\textbf{Fact:} \texttt{A_0; \ldots; A_m.}

\textbf{Rule:} \texttt{A_0; \ldots; A_m :- L_1, \ldots, L_n.}

A disjunctive head holds if at least one of its atoms is true. Answer sets of a disjunctive logic program satisfy a minimality criterion that we do not detail here.
3.1 Input Language of gringo and clingo

(see [21, 35] for an implementation methodology in disjunctive ASP). We only mention that the simple disjunctive program a; b. has two answer sets, one containing a and another one containing b, while both atoms do not jointly belong to an answer set. After adding the rules of Example 3.1, a single answer set containing both a and b is obtained. This illustrates that disjunction in ASP is neither strictly exclusive or inclusive but subject to minimization.

In general, the use of disjunction may increase computational complexity [20]. We thus suggest to use “choice constructs” (detailed in Section 3.1.12) instead of disjunction, unless the latter is required for complexity reasons.

3.1.5 Double Negation and Head Literals

The input language of gringo also supports double default negated literals, written not not A. They are satisfied whenever their positive counterparts are. But like negative literals of form not A, double negated ones are also preceded by not and do thus not require an (acyclic) derivation from the program; it is sufficient that they are true in the model at hand.

Consider the logic program:

a :- not not b.
b :- not not a.

This program has an empty answer set, like the program in Example 3.1, as well as the additional answer set containing both a and b. This is because neither ‘not not a’ nor ‘not not b’ requires an acyclic derivation from the program. Note that, in contrast to Example 3.1, the above program does not induce mutual positive dependencies between a and b. Given this, a and b can thus be both true or false, just like in classical logic.

Also, negative literals are admitted in the head of rules. When disregarding disjunction, this offers just another way to write integrity constraints, putting the emphasis on the head literal. In fact, the rule not A0 ::- L1, ..., Ln.. is equivalent to :- L1, ..., Ln, not not A0.., and with double negation in the head, rule not not A0 ::- L1, ..., Ln.. is equivalent to :- L1, ..., Ln, not A0.

Example 3.4. Consider the logic program:

1 fly(X) :- bird(X), not not fly(X).
2 not fly(X) :- penguin(X).

The possibility that a bird flies is expressed with a double negation in the first line. Solutions with flying penguins are filtered out in the second line. Like in Example 3.2 there are two answer sets, but without an explicit atom to indicate that a bird does not fly. Hence, the answer set where tweety does not fly contains no atoms over predicate fly/1.

Remark 3.1. Note that negative head literals are also supported in disjunctions. For more information see [63].
3.1 Input Language of gringo and clingo

3.1.6 Boolean Constants

Sometimes it is useful to have literals possessing a constant truth value. Literals over the two Boolean constants #true and #false, which are always true or false, respectively, have a constant truth value.

Example 3.5. Consider the following program:

The unique answer set of the program, can be inspected by invoking:

```
clingo bool.lp 0
```
or alternatively:

```
gringo bool.lp | clasp 0
```
Note that this program simply produces an empty grounding:

```
clingo --text \\
bool.lp
```
or alternatively:

```
gringo --text \\
bool.lp
```

The first rule uses #true in the head. Because this rule is a fact, it is trivially satisfied. Similarly, the rules in Line 2 and 3 have satisfied heads. The bodies of the last three integrity constraints are false. Hence, the constraints do not cause a conflict. Note that neither of the rules above derives any atom. Thus, we obtain the empty answer set for the program.

See Example 3.14 below for an application of interest.

3.1.7 Built-in Arithmetic Functions

Besides integers (constant arithmetic functions), written as sequences of the digits 0...9 possibly preceded by ‘-’, gringo and clingo support a variety of arithmetic functions that are evaluated during grounding. The following symbols are used for these functions: + (addition), – (subtraction, unary minus), * (multiplication), / (integer division), \ (modulo), ** (exponentiation), |\| (absolute value), & (bitwise AND), ? (bitwise OR), ^ (bitwise exclusive OR), and ~ (bitwise complement).

Example 3.6. The usage of arithmetic functions is illustrated by the program:

The unique answer set of the program, obtained after evaluating all arithmetic functions, can be inspected by invoking:

```
clingo --text \\
arithf.lp
```
or alternatively:

```
gringo --text \\
arithf.lp
```

```bash
1 left (7).
2 right (2).
3 plus ( L + R ) :- left(L), right(R).
4 minus ( L - R ) :- left(L), right(R).
5 uminus ( - R ) :- right(R).
6 times ( L * R ) :- left(L), right(R).
7 divide ( L / R ) :- left(L), right(R).
8 modulo ( L \ R ) :- left(L), right(R).
9 absolute(| - R|) :- right(R).
10 power ( L ** R ) :- left(L), right(R).
11 bitand ( L & R ) :- left(L), right(R).
12 bitor ( L ? R ) :- left(L), right(R).
13 bitxor ( L ^ R ) :- left(L), right(R).
14 bitneg ( ~ R ) :- right(R).
```
Note that the variables \( L \) and \( R \) are instantiated to 7 and 2, respectively, before arithmetic evaluations. Consecutive and non-separative (e.g., before \( '(' \) ) spaces can optionally be dropped. The four bitwise functions apply to signed integers, using two’s complement arithmetic.

**Remark 3.2.** An occurrence of a variable in the scope of an arithmetic function only counts as positive in the sense of safety (cf. Page 21) for simple arithmetic terms. Such simple arithmetic terms are terms with exactly one variable occurrence composed of the arithmetic functions \( '+' \), \( '-' \), \( '*' \), and integers. Moreover, if multiplication is used, then the constant part must not evaluate to 0 for the variable occurrence to be considered positive. E.g., the rule \( q(X) :- p(2*(X+1)). \) is considered safe, but the rule \( q(X) :- p(X+X). \) is not.

### 3.1.8 Built-in Comparison Predicates

Grounder gringo (and clingo) feature a total order among variable-free terms (without arithmetic functions). The built-in predicates to compare terms are \( = \) (equal), \( 
eq \) (not equal), \( < \) (less than), \( \leq \) (less than or equal), \( > \) (greater than), and \( \geq \) (greater than or equal). Comparison literals over the above comparison predicates are used like other literals (cf. Section 3.1.2) but are evaluated during grounding.

**Example 3.7.** The application of comparison literals to integers is illustrated by the following program:

```
1 num(1). num(2).
2 eq (X,Y) :- X = Y, num(X), num(Y).
3 neq(X,Y) :- X \neq Y, num(X), num(Y).
4 lt (X,Y) :- X < Y, num(X), num(Y).
5 leq(X,Y) :- X \leq Y, num(X), num(Y).
6 gt (X,Y) :- X > Y, num(X), num(Y).
7 geq(X,Y) :- X \geq 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 literals, so that the latter actually compare the results of arithmetic evaluations.

**Example 3.8.** Comparison literals can also be applied to constants and functions, as illustrated by the following program:

```
1 sym(1). sym(a). sym(f(a)).
2 eq (X,Y) :- X = Y, sym(X), sym(Y).
3 neq(X,Y) :- X \neq Y, sym(X), sym(Y).
4 lt (X,Y) :- X < Y, sym(X), sym(Y).
5 leq(X,Y) :- X \leq Y, sym(X), sym(Y).
6 gt (X,Y) :- X > Y, sym(X), sym(Y).
7 geq(X,Y) :- X \geq Y, sym(X), sym(Y).
```

As above, by invoking:

```
clingo --text \ symbc.lp
```

or alternatively:

```
gringo --text \ symbc.lp
```

one can inspect the simplified ground program obtained by evaluating built-ins.
Integers are compared in the usual way, constants are ordered lexicographically, and functions both structurally and lexicographically. Furthermore, all integers are smaller than constants, which in turn are smaller than functions.

The built-in comparison predicate ‘=’ has another interesting use case. Apart from just testing whether a relation between two terms holds, it can be used to define shorthands (via unification) for terms.

**Example 3.9.** This usage is illustrated by the following program:

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

The body of the rule in Line 2 defines four comparison predicates over ‘=’, which directly or indirectly depend on X and Y. The values of X and Y are obtained via instances of the predicate `num/1`. The first comparison predicate depends on X to provide shortcut XX. Similarly, the second comparison predicate depends on Y to provide shortcut YY. The third comparison predicate provides variable Y’ because it occurs in a simple arithmetic term, which is solved during unification. The last comparison predicate provides no variables and, hence, is just a test, checking whether its left-hand and right-hand sides are equal.

**Example 3.10.** This example illustrates how to unify with function terms and tuples:

```
1  sym(f(a,1,2)).  sym(f(a,2,4)).  sym(f(a,b)).
2  sym( (a,1,2)).  sym( (a,2,4)).  sym( (a,b)).
3  unify1(X) :- f(a,X,X+1) = F,  sym(F).
4  unify2(X) :- (a,X,X+1) = T,  sym(T).
```

Here, `f(a,X,X+1)` or `(a,X,X+1)`, respectively, is unified with instances of the predicate `sym/1`. To this end, arguments of `sym/1` with matching arity are used to instantiate the variable X occurring as the second argument in terms on the left-hand sides of ‘=’. With a value for X at hand, we can further check whether the arithmetic evaluation of X+1, occurring as the third argument, coincides with the corresponding value given on the right-hand side of ‘=’.

**Remark 3.3.** Note that comparison literals can be preceded by `not` or `not not`. In the first case, this is equivalent to using the complementary comparison literal (e.g., ‘<’ and ‘>=’ complement each other). In the second case, the prefix has no effect on the meaning of the literal.

An occurrence of a variable in the scope of a built-in comparison literal over ‘!=’, ‘<’, ‘<=’, ‘>’, or ‘>=’ does not count as a positive occurrence in the sense of safety (cf. Page 21), i.e., such comparison literals are not considered to be positive.
Unlike with the built-in comparison literals above, comparisons predicates over ‘=’ are considered as positive (body) literals in the sense of safety (cf. Page 21), so that variables occurring on one side can be instantiated. However, this only works when unification can be made directionally, i.e., it must be possible to instantiate one side without knowing the values of variables on the other side. For example, the rule \( p(X) :- X = Y, Y = X. \) is not accepted by gringo (or clingo) because values for \( X \) rely on values for \( Y \), and vice versa. Only simple arithmetic terms can be unified with (cf. Remark 3.2). Hence, variable \( X \) in literal \( X \times X = 8 \) must be bound by some other positive literal.

### 3.1.9 Intervals

Line 1 of Example 3.9 contains five facts of the form \( \text{num}(k) \). over consecutive integers \( k \). For a more compact representation, gringo and clingo support integer intervals of the form \( i..j \). Such an interval, representing each integer \( k \) such that \( i \leq k \leq j \), is expanded during grounding. An interval is expanded differently depending on where it occurs. In the head of a rule, an interval is expanded conjunctively, while in the body of a rule, it is expanded disjunctively. So we could have simply written \( \text{num}(1..5) \). to represent the five facts.

**Example 3.11.** Consider the following program:

1. \( \text{size}(3) \).
2. \( \text{grid}(1..S,1..S) :- \text{size}(S) \).

Because all intervals in the second rule occur in the rule head, they expand conjunctively. Furthermore, the two intervals expand into the cross product \((1..3) \times (1..3)\), resulting in the following set of facts:

2. \( \text{grid}(1,1). \text{grid}(1,2). \text{grid}(1,3). \text{grid}(2,1). \text{grid}(2,2). \text{grid}(2,3). \text{grid}(3,1). \text{grid}(3,2). \text{grid}(3,3). \)

Similarly, intervals can be used in a rule body. Typically, this is done using comparison literals over ‘=‘, which expand disjunctively:

2. \( \text{grid}(X,Y) :- X = 1..S, Y = 1..S, \text{size}(S) \).

This rule expands into the same set of facts as before. But intervals in comparison literals have the advantage that additional constraints can be added. For example, one could add the comparison literals \( X-Y \neq 0 \) and \( X+Y-1 \neq S \) to the rule body to exclude the diagonals of the grid.

**Remark 3.4.** An occurrence of a variable in the specification of the bounds of an integer interval, like \( S \) in Line 2 of Example 3.11, does not count as a positive occurrence in the sense of safety (cf. Page 21). Hence, such a variable must also have another positive occurrence elsewhere; here in \( \text{size}(S) \).
3.1 Input Language of gringo and clingo

3.1.10 Pooling

The token ‘;’ admits pooling alternative terms to be used as arguments of an atom, function, or tuple. Argument lists written in the form \((\ldots, X; Y, \ldots)\) abbreviate multiple options: \((\ldots, X), (Y, \ldots)\). Pools are expanded just like intervals, i.e., conjunctively in the head and disjunctively in the body of a rule. In fact, the interval \(1..3\) is equivalent to the pool \((1;2;3)\).

Example 3.12. The following program makes use of pooling. It is similar to Example 3.11 but with the difference that, unlike intervals, pools have a fixed size:

\[
1 \text{grid}((1;2;3),(1;2;3)) .
\]

Because all pools in this rule occur in the head, they are expanded conjunctively. Furthermore, the two pools expand into the cross product \((1..3) \times (1..3)\), resulting again in the following set of facts:

\[
\begin{align*}
\text{grid}(1,1). & \quad \text{grid}(1,2). \quad \text{grid}(1,3). \\
\text{grid}(2,1). & \quad \text{grid}(2,2). \quad \text{grid}(2,3). \\
\text{grid}(3,1). & \quad \text{grid}(3,2). \quad \text{grid}(3,3). 
\end{align*}
\]

Like intervals, pools can also be used in the body of a rule, where they are expanded disjunctively:

\[
1 \text{grid}(X,Y) :- X = (1;2;3), \ Y = (1;2;3) .
\]

This rule expands into the same set of facts as before. As in Example 3.11, additional constraints involving \(X\) and \(Y\) can be added.

For another example on pooling, featuring non-consecutive elements, see Section 5.1.1.

3.1.11 Conditions and Conditional Literals

A conditional literal is of the form

\[ L_0 : L_1, \ldots, L_n \]

where every \(L_j\) for \(0 \leq j \leq n\) is a literal, \(L_1, \ldots, L_n\) is called condition, and ‘:’ resembles mathematical set notation. Whenever \(n = 0\), we get a regular literal and denote it as usual by \(L_0\).

For example, the rule

\[
a :- b : c.
\]

\[8\]We make use of the fact that one-elementary tuples must be made explicit by a trailing ‘,’ (cf. Section 3.1.1). E.g., \((1;1,)\) expands into \((1)\) and \((1,)\), where \((1)\) is equal to the integer 1. On the other hand, note that the rule \(p(X) :- X = (1,2;3,4)\) is expanded into \(p((1,2))\) and \(p((3,4))\), given that \((1,2)\) and \((3,4)\) are proper tuples, and the same facts are also obtained from \(p((1,2;3,4))\). Unlike that, \(p(1,2;3,4)\) yields \(p(1,2)\) and \(p(3,4)\) because ‘;’ here splits an argument list, rather than a tuple.
yields a whenever either c is false (and thus no matter whether b holds or not) or both b and c are true.

**Remark 3.5.** Logically, \( L_0 \) and \( L_1, \ldots, L_n \) act as head and body, respectively, which gives \( L_0 : L_1, \ldots, L_n \) the flavor of a nested implication (see [57] for details).

Together with variables, conditions allow for specifying collections of expressions within a single rule or aggregate. This is particularly useful for encoding conjunctions (or disjunctions) over arbitrarily many ground atoms as well as for the compact representation of aggregates (detailed in Section 3.1.12).

**Example 3.13.** The following program uses, in Line 5 and 6, conditions in a rule body and in a rule head, respectively:

```plaintext
1 person(jane). person(john).
2 day(mon). day(tue). day(wed). day(thu).
   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.
```

The rules in Line 5 and 6 are instantiated as follows:

```
meet :- available(jane), available(john).
   on(mon); on(tue); on(wed); on(thu); on(fri) :- meet.
```

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

Further following set notation, a condition can be composed by separating literals with a comma, viz. `,'. Note that commas are used to separate both literals in rule bodies as well as conditions. To resolve this ambiguity, a condition is terminated with a semicolon `;' (rather than `,') when further body literals follow.

**Example 3.14.** The following program uses a literal with a composite condition in the middle of the rule body. Note the semicolon `;' after the condition:

```plaintext
1 set(1..4).
2 next(X,Z) :- set(X), #false : X < Y, set(Y), Y < Z; set(Z), X < Z.
```

---

9Recall our suggestion from Section 3.1.4 to use “choice constructs” (detailed in Section 3.1.12) instead of disjunction, unless the latter is required for complexity reasons. This also means that conditions must not be used outside of aggregates in rule heads if disjunction is unintended.
3.1 Input Language of gringo and clingo

The conditional literal in the second rule evaluates to false whenever there is an element $Y$ between $X$ and $Z$. Hence, all rule instantiations where $X$ and $Z$ are not direct successors are discarded because they have a false body. On the other hand, whenever $X$ and $Z$ succeed each other, the condition is false for all elements $Y$. This means that the literal with condition stands for an empty conjunction, which is true:

```
next(1,2). next(2,3). next(3,4).
```

We obtain an answer set where the elements of `set/1` are ordered via `next/2`.

**Remark 3.6.** There are three important issues about the usage of conditions:

1. Any variable occurring within a condition does not count as a positive occurrence outside the condition in the sense of safety (cf. Page 21). Variables occurring in atoms not subject to any condition are *global*. Each variable within an atom in front of a condition must be global or have a positive occurrence on the right-hand side of the condition.

2. During grounding, the instantiation of global variables takes precedence over non-global ones, that is, the former are instantiated before the latter. As a consequence, variables that occur globally are substituted by terms before a condition is further evaluated. Hence, the names of variables in conditions must be chosen with care, making sure that they do not *accidentally* match the names of global variables.

3. We suggest using *domain predicates* [78] or built-ins (both used in Line 3 of Example 3.14) in conditions. Literals over such predicates are completely evaluated during grounding. In a logic program, domain predicates can be recognized by observing that they are neither subject to negative recursion (through `not`) nor to disjunction or “choice constructs” (detailed in Section 3.1.12) in the head of any rule. The domain predicates defined in Example 3.14 are `set/1` and `next/1`. Literals with such conditions expand to arbitrary length disjunctions or conjunctions in the head or body of a rule, respectively. Otherwise, conditions give rise to nested implications. For further details see [57].

---

3.1.12 Aggregates

Aggregates are expressive modeling constructs that allow for forming values from groups of selected items. Together with comparisons they allow for expressing conditions over these terms. For instance, we may state that the sum of a semester’s course credits must be at least 20, or that the sum of prizes of shopping items should not exceed 30 Euros.
More formally, an aggregate is a function on a set of tuples that are normally subject to conditions. By comparing an aggregated value with given values, we can extract a truth value from an aggregate’s evaluation, thus obtaining an aggregate atom. Aggregate atoms come in two variants depending on whether they occur in a rule head or body.

**Body Aggregates** The form of an aggregate atom occurring in a rule body is as follows:

\[ s_1 <_1 \alpha \{ \begin{array}{c} t_1 : L_1; \ldots ; t_n : L_n \end{array} \} <_2 s_2 \]

Here, all \( t_i \) and \( L_i \), forming aggregate elements, are non-empty tuples of terms and literals (as introduced in Section 3.1.1), respectively. \( \alpha \) is the name of some function that is to be applied to the term tuples \( t_i \) that remain after evaluating the conditions expressed by \( L_i \). Finally, the result of applying \( \alpha \) is compared by means of the comparison predicates \( <_1 \) and \( <_2 \) to the terms \( s_1 \) and \( s_2 \), respectively. Note that one of the guards \( <_1 \) or \( <_2 \) (or even both) can be omitted; left out comparison predicates \( <_1 \) or \( <_2 \) default to \( \leq \), thus interpreting \( s_1 \) and \( s_2 \) as lower or upper bound, respectively.

Currently, **gringo (and clingo)** support the aggregates \#count (the number of elements; used for expressing cardinality constraints), \#sum (the sum of weights; used for expressing weight constraints), \#sum+ (the sum of positive weights), \#min (the minimum weight), and \#max (the maximum weight). The weight refers to the first element of a term tuple. Aggregate atoms, as described above, are obtained by writing either \#count, \#sum, \#sum+, \#min, or \#max for \( \alpha \). Note that, unlike the other aggregates, the \#count aggregate does not require weights.

For example, instances of the natural language examples for aggregates given at the beginning of this section can be expressed as follows.

\[ 20 \leq \#sum \{ \begin{array}{c} 4 : \text{course(db)};  \ldots ; 6 : \text{course(ai)};  \\ 8 : \text{course(project)}; 3 : \text{course(xml)} \end{array} \} \leq 30 \]

Both aggregate atoms can be used in the body of a rule like any other atom, possibly preceded by negation. Within both aggregate atoms, atoms like \text{course(ai)} or \text{broom} are associated with weights. Assuming that \text{course(db)}, \text{course(ai)} as well as \text{bananas} and \text{broom} are true, the aggregates inner sets evaluate to \{4; 6\} and \{3; 10\}, respectively. After applying the \#sum aggregate function to both sets, we get \( 20 \leq 10 \) and \( 13 \leq 30 \); hence, in this case, the second aggregate atom holds while the first one does not.

As indicated by the curly braces, the elements within aggregates are treated as members of a set. Hence, duplicates are not accounted for twice. For instance, the following aggregate atoms express the same:

\[ \#count \{ 42 : a; \ t : \text{not b} \} = 2 \]

\[ \#count \{ 42 : a; 42 : a; \ t : \text{not b}; \ t : \text{not b} \} = 2 \]
That is, if $a$ holds but not $b$, both inner sets reduce to \{42; t\}; and so both aggregate atoms evaluate to true. However, both are different from the aggregate

\[
\#\text{count} \{ 42 : a; t : \text{not } b; s : \text{not } b \} = 2
\]

that holds if both $a$ and $b$ are false, yielding \#count\{t; s\} = 2.

Likewise, the elements of other aggregates are understood as sets. Consider the next two summation aggregates:

\[
\#\text{sum} \{ 3 : \text{cost}(1,2,3); 3 : \text{cost}(2,3,3) \} = 3
\]

\[
\#\text{sum} \{ 3,1,2 : \text{cost}(1,2,3); 3,2,3 : \text{cost}(2,3,3) \} = 6
\]

As done in Section 5.2.1, an atom like \text{cost}(1,2,3) can be used to represent an arc from node 1 to 2 with cost 3. If both \text{cost}(1,2,3) and \text{cost}(2,3,3) hold, the first sum evaluates to 3, while the second yields 6. Note that all term tuples, the singular tuple 3 as well as the ternary tuples 3, 1, 2 and 3, 2, 3 share the same weight, viz. 3. However, the set property makes the first aggregate count edges with the same cost only once, while the second one accounts for each edge no matter whether they have the same cost or not. To see this, observe that after evaluating the conditions in each aggregate, the first one reduces to \#\text{sum}\{3\}, while the second results in \#\text{sum}\{3,1,2;3,2,3\}. In other words, associating each cost with its respective arc enforces a multi-set property; in this way, the same cost can be accounted for several times.

**Head Aggregates** Whenever a rule head is a (single) aggregate atom, the derivable head literals must be distinguished. This is done by appending such atoms (or in general literals) separated by an additional ‘:’ to the tuples of the aggregate elements:

\[
s_1 <_1 \alpha \{ t_1 : L_1; \ldots; t_n : L_n \} \alpha <_2 s_2
\]

Here, all $L_i$ are literals as introduced in Section 3.1.2, while all other entities are as described above. The second colon in $t_i : L_i$ is dropped whenever $L_i$ is empty, yielding $t_i : L_i$.

**Remark 3.7.** Aggregate atoms in the head can be understood as a combination of unrestricted choices with body aggregates enforcing the constraint expressed by the original head aggregate. In fact, when producing smodels format, all aggregate atoms occurring in rule heads are transformed away. For details consult [76, 32].

**Shortcuts** There are some shorthands that can be used in the syntactic representation of aggregates. The expression

\[
s_1 <_1 \alpha \{ L_1 : L_1; \ldots; L_n : L_n \} \alpha <_2 s_2
\]

where all entities are defined as above is a shortcut for

\[
s_1 <_1 \#\text{count} \{ t_1 : L_1; \ldots; t_n : L_n \} <_2 s_2
\]
if it appears in the head of a rule, and it is a shortcut for

\[ s_1 <_1 \#\text{count} \{ t_1 : L_1, L_1; \ldots; t_n : L_n, L_n \} <_2 s_2 \]

if it appears in the body of a rule. In both cases, all \( t_i \) are pairwise distinct term tuples generated by gringo whenever the distinguished (head) literals \( L_i \) are different. Just like with aggregates, the guards \( 's_1 <_1' \) and \( '<_2 s_2' \) are optional, and the symbols \( '<_1' \) and \( '<_2' \) default to \( '<=' \) if omitted.

For example, the rule

\[ \{ a; b \} \]

is expanded to

\[ \#\text{count} \{ 0,a : a; 0,b : b \} \]

Here, gringo generates two distinct term tuples \( 0,a \) and \( 0,b \). With clingo, we obtain four answer sets representing all sets over \( a \) and \( b \).

Recurrences of literals yield identical terms, as we see next. The rule

\[ \{ a; a \} \]

is expanded to

\[ \#\text{count} \{ 0,a : a; 0,a : a \} \]

In fact, within the term tuple produced by gringo, the first term indicates the number of preceding default negations, and the second reproduces the atom as a term in order to make the whole term tuple unique. To see this, observe that the integrity constraint

\[ :- \{ a; not b; not not c \} > 0. \]

is expanded to

\[ :- \#\text{count} \{ 0,a : a; 1,b : not b; 2,c : not not c \} > 0. \]

**Remark 3.8.** By allowing the omission of \#count, so-called “cardinality constraints” [76] can almost be written in their traditional notation (without keyword, yet different separators), as put forward in the lparse grounder [78].

Having discussed head and body aggregate atoms, let us note that there is a second way to use body aggregates: they act like positive literals when used together with comparison predicate \( '=' \). For instance, variable \( X \) is safe in the following rules:

\[
\begin{align*}
\text{cnt}(X) & :- X = \#\text{count} \{ 2 : a; 3 : a \}. \\
\text{sum}(X) & :- X = \#\text{sum} \{ 2 : a; 3 : a \}. \\
\text{pos}(X) & :- X = \#\text{sum}+ \{ 2 : a; 3 : a \}. \\
\text{min}(X) & :- X = \#\text{min} \{ 2 : a; 3 : a \}. \\
\text{max}(X) & :- X = \#\text{max} \{ 2 : a; 3 : a \}.
\end{align*}
\]
3.1 Input Language of gringo and clingo

Under the assumption that atom \( a \) holds, the atoms \( \text{cnt}(2), \text{sum}(5), \text{pos}(5), \text{min}(2), \) and \( \text{max}(3) \) are derived by the above rules. If \( a \) does not hold, we derive \( \text{cnt}(0), \text{sum}(0), \text{pos}(0), \text{min}(\#\text{sup}), \) and \( \text{max}(\#\text{inf}) \). Here, the special constants \( \#\text{sup} \) and \( \#\text{inf} \) (introduced in Section 3.1.1), obtained by applying \( \#\text{min} \) and \( \#\text{max} \) to the empty set of weights, indicate the neutral elements of the aggregates. These constants can also be used as weights, subject to \( \#\text{min} \) and \( \#\text{max} \) (in order to exceed any other ground term):

\[
\text{bot} \leftarrow \#\text{min} \{ \#\text{inf} : a \} -1000.
\]
\[
\text{top} \leftarrow 1000 \#\text{max} \{ \#\text{sup} : a \}.
\]

Assuming that atom \( a \) holds, the atoms \( \text{bot} \) and \( \text{top} \) are derived by the above rules because both \( \#\text{inf} \leq -1000 \) and \( 1000 \leq \#\text{sup} \) hold (cf. Section 3.1.8 for details how terms are ordered).

**Remark 3.9.** Although it seems convenient to use aggregates together with the ‘\( '=' \)’ predicate, this feature should be used with care. If the literals of an aggregate belong to domain predicates (see Remark 3.6) or built-ins, the aggregate is evaluated during grounding to exactly one value. Otherwise, if the literals do not belong to domain predicates, the value of an aggregate is not known during grounding, in which case gringo or clingo unwraps all possible outcomes of the aggregate’s evaluation. The latter can lead to a space blow-up, which should be avoided whenever possible. For instance, unwrapping the aggregate in

\[
\{ a; b; c \}.
\]
\[
\leftarrow \#\text{sum} \{ 1 : a; 2 : b; 3 : c \} = N, \; N > 3.
\]

yields three integrity constraints:

\[
\leftarrow \#\text{sum} \{ 1 : a; 2 : b; 3 : c \} = 4.
\]
\[
\leftarrow \#\text{sum} \{ 1 : a; 2 : b; 3 : c \} = 5.
\]
\[
\leftarrow \#\text{sum} \{ 1 : a; 2 : b; 3 : c \} = 6.
\]

Such duplication does not happen when we use a comparison predicate instead:

\[
\leftarrow \#\text{sum} \{ 1 : a; 2 : b; 3 : c \} > 3.
\]

Hence, it is advisable to rather apply comparison predicate ‘\( '>' \)’ directly. In general, aggregates should only be used to bind variables if they refer solely to domain predicates and built-ins.

**Non-ground Aggregates** After considering the syntax and semantics of ground aggregate atoms, we now turn our attention to non-ground aggregates. Regarding contained variables, only variable occurrences in the guards give rise to global variables. Hence, any variable in an aggregate element must be bound by either a positive global occurrence or a variable that occurs positively in its condition \( L_i \). Variable names in aggregate elements have to be chosen carefully to avoid clashes with global variables. Furthermore, pools and intervals in aggregate elements give rise to multiple aggregate elements; very similar to the disjunctive unpacking of pools.
3.1 Input Language of gringo and clingo

and intervals in rules. The following example, making exhaustive use of aggregates, demonstrates a variety of features (note that it ignores Remark 3.9).

**Example 3.15.** 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; 1,2,5).
2. course(2,1,4; 2,2,4).
3. course(3,1,6; 3,3,6).
4. course(4,1,3; 4,3,3; 4,4,3).
5. course(5,1,4; 5,4,4).
6. course(6,2,2; 6,3,2).
7. course(7,2,4; 7,3,4; 7,4,4).
8. course(8,3,5; 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 student’s first condition is to enroll in 3 to 6 courses:

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

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

\[
3 \leq \#\text{count} \{ 0, \text{enroll}(1) : \text{enroll}(1); \\
0, \text{enroll}(2) : \text{enroll}(2); \\
0, \text{enroll}(3) : \text{enroll}(3); \\
0, \text{enroll}(4) : \text{enroll}(4); \\
0, \text{enroll}(5) : \text{enroll}(5); \\
0, \text{enroll}(6) : \text{enroll}(6); \\
0, \text{enroll}(7) : \text{enroll}(7); \\
0, \text{enroll}(8) : \text{enroll}(8) \} \leq 6.
\]

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. :- #count { C,S : enroll(C), course(C,S,H) } 10.
11. :- 2 #count { C,2 : not enroll(C), course(C,2,H) }.
12. :- 6 #count { C,3 : enroll(C), course(C,3,H); \\
C,4 : enroll(C), course(C,4,H) }.
Each of the three integrity constraints above contains a `#count` aggregate, in which `,’ is used to construct composite conditions (introduced in Section 3.1.11). Recall that aggregates operate on sets and thus duplicates are removed; hence, we use term tuples to take into account courses together with their subject areas. Thus, the integrity constraint in Line 10 is instantiated as follows:  

\begin{verbatim}
10 : - 10 >= #count { 1,1 : enroll(1); 1,2 : enroll(1);
2,1 : enroll(2); 2,2 : enroll(2);
3,1 : enroll(3); 3,3 : enroll(3);
4,1 : enroll(4); 4,3 : enroll(4); 4,4 : enroll(4);
5,1 : enroll(5); 5,4 : enroll(5);
6,2 : enroll(6); 6,3 : enroll(6);
7,2 : enroll(7); 7,3 : enroll(7); 7,4 : enroll(7);
8,3 : enroll(8); 8,4 : enroll(8) }.
\end{verbatim}

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”).

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

\begin{verbatim}
14 hours(C,H) : - course(C,S,H).
15 max_hours(20).
\end{verbatim}

The rule in Line 14 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:  

\begin{verbatim}
16 : - not M-2 #sum { H,C : enroll(C), hours(C,H) } M,
    max_hours(M).
17 : - #min { H,C : enroll(C), hours(C,H) } 2.
18 : - 6 #max { H,C : enroll(C), hours(C,H) }.
\end{verbatim}

As illustrated in Line 16, we may use default negation via ‘not’ in front of aggregate atoms, and bounds may be specified by terms with variables. In fact, by instantiating M to 20, we obtain the following ground instance of the integrity constraint in Line 16:

\footnote{Because contact hours are uniquely associated to a course, gringo’s shortcut expansion of `:- { course(C,S,H) : enroll(C) } 10.` is equivalent to the rule in Line 10 here. Similar equivalences hold for the other `#count` aggregates.}
The above integrity constraint states that the $\text{#sum}$ of contact hours per week must lie in-between 18 and 20. Note that the $\text{#min}$ and $\text{#max}$ aggregates in Line 17 and 18, respectively, work on the same set of aggregate elements as in Line 16. While the integrity constraint in Line 17 stipulates that any course to enroll must include more than 2 contact hours, the one in Line 18 prohibits enrolling for courses of 6 or more contact hours. Of course, the last two requirements could also be formulated as follows:

$$
17 \quad \neg \text{enroll}(C), \text{hours}(C,H), H \leq 2.
$$

$$
18 \quad \neg \text{enroll}(C), \text{hours}(C,H), H \geq 6.
$$

Finally, the following rules illustrate the use of aggregates together with comparison predicate '='.

$$
19 \quad \text{courses}(N) \ :- \ N = \text{#count} \{ C : \text{enroll}(C) \}.
$$

$$
20 \quad \text{hours}(N) \ :- \ N = \text{#sum} \{ H,C : \text{enroll}(C), \text{hours}(C,H) \}.
$$

The role of aggregates here is different from before, as they now serve to bind an integer to variable $N$. The effect of Line 19 and 20, which do not follow the recommendation in Remark 3.9, 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 clingo or clasp shows 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.

Remark 3.10. Users familiar with gringo 3 may remember that conditions in aggregates had to be either literals over domain predicates or built-ins. This restriction does not exist anymore in gringo and clingo 4.

### 3.1.13 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 dlv’s weak constraints [14]. The form of weak constraints is similar to integrity constraints (cf. Section 3.1.2) being associated with a term tuple:

$$
\neg \quad L_1, \ldots, L_n. \quad [w@p, t_1, \ldots, t_n]
$$

The priority ‘@p’ is optional. For simplicity, we first consider the non-prioritized case omitting ‘@p’. Whenever the body of a weak constraint is satisfied, it contributes its term tuple (as with aggregates, each tuple is included at most once) to
3.1 Input Language of gringo and clingo

A cost function. This cost function accumulates the integer weights \( w \) of all contributed tuples just like a \#sum aggregate does. The semantics of a program with weak constraints is intuitive: an answer set is optimal if the obtained cost is minimal among all answer sets of the given program. Whenever there are different priorities attached to tuples, we obtain a (possibly zero) cost for each priority. To determine whether an answer set is optimal, we do not just compare two single costs but lexicographically compare cost tuples whose elements are ordered by priority (greater is more important). Note that a tuple is always associated with a priority; if it is omitted, then the priority defaults to zero. A weak constraint is safe if the variables in its term tuples are bound by the atoms in the body and the safety requirements for the body itself are the same as for integrity constraints.

As an alternative way to express an optimization problem, there are optimization statements. A minimize statement of the form

\[
\text{#minimize}\ \{\ \ w_1@p_1, t_1 : L_1, \ldots, w_n@p_n, t_n : L_n \ \}\ .
\]

represents the following \( n \) weak constraints:

\[
\sim L_1. [w_1@p_1, t_1] \ldots \sim L_n. [w_n@p_n, t_n]
\]

Moreover, maximize statements can be viewed as minimize statements with inverse weights. Hence, a maximize statement of the form

\[
\text{#maximize}\ \{\ \ w_1@p_1, t_1 : L_1, \ldots, w_n@p_n, t_n : L_n \ \}\ .
\]

represents the following \( n \) weak constraints:

\[
\sim L_1. [-w_1@p_1, t_1] \ldots \sim L_n. [-w_n@p_n, t_n]
\]

As with weak constraints, the priorities ‘@\( p_i \)’ are optional and default to zero.

**Example 3.16.** 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 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 1–7 of the following program:

```
1 { hotel(1..5) } = 1.
2 star(1,5). cost(1,170).
3 star(2,4). cost(2,140).
4 star(3,3). cost(3,90).
5 star(4,3). cost(4,75). main_street(4).
6 star(5,2). cost(5,60).
7 noisy :- hotel(X), main_street(X).
8 #maximize { Y@1,X : hotel(X), star(X,Y) }.
9 #minimize { Y/Z@2,X : hotel(X), cost(X,Y), star(X,Z) }.
10 :- noisy. [ 1@3 ]
```
3.1 Input Language of gringo and clingo

Line 8–9 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 10 states that avoiding noise is our main priority. The secondary optimization criterion in Line 9 consists of minimizing the cost per star. Finally, the third optimization statement in Line 8 specifies that we want to maximize the number of stars among hotels that are otherwise indistinguishable. The optimization statements in Line 8–10 are instantiated as follows:

8  \[ \text{hotel}(1). \ [ -501, 1] \]
8  \[ \text{hotel}(2). \ [ -401, 2] \]
8  \[ \text{hotel}(3). \ [ -301, 3] \]
8  \[ \text{hotel}(4). \ [ -301, 4] \]
8  \[ \text{hotel}(5). \ [ -201, 5] \]
9  \[ \text{hotel}(1). \ [ 3402, 1] \]
9  \[ \text{hotel}(2). \ [ 3502, 2] \]
9  \[ \text{hotel}(3). \ [ 3002, 3] \]
9  \[ \text{hotel}(4). \ [ 2502, 4] \]
9  \[ \text{hotel}(5). \ [ 3002, 5] \]
10 \[ \text{noisy}. \ [ 103 ] \]

If we now use clasp or clingo 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 with respect to the second most significant optimization statement in Line 9. This tie is broken via the least significant optimization statement in Line 8 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.

3.1.14 External Functions

Utilizing the scripting languages Lua or Python\(^\text{12}\), gringo’s input language can be enriched by arbitrary functions. We focus on functions that are evaluated during grounding here. In Section 4, we explain how to take complete control of the grounding and solving process using the scripting API. We do not give an introduction to Lua or Python here (there are numerous tutorials on the web), but give some examples showing the capabilities of this integration. In the following, we show code snippets for both scripting languages. Note that our precompiled binaries ship with Lua support and can be used to run the Lua examples. To enable Python support, gringo and clingo have to be compiled from source (cf. Section 1.1). A complete reference for the Python scripting API is available at: http://potassco.sourceforge.net/gringo.html

Example 3.17. The first example shows how to add a simple arithmetic function:

\(^{12}\text{http://lua.org and http://python.org}\)
In Line 3, we add a function that calculates the greatest common divisor of two numbers. Integers from a logic program are directly mapped to their Lua and Python equivalents and can be returned from the function. The \texttt{gcd} function can then be used in a logic program:

To inspect the unique answer set of the program, invoke:
\texttt{gringo --text \\ gcd.lp gcd-lua.lp}
or:
\texttt{gringo --text \\ gcd.lp gcd-py.lp}
Calls to \texttt{clingo} are similar.

\begin{verbatim}
#script (lua)

function gcd(a, b)
    if a == 0 then
        return b
    else
        return gcd(b % a, a)
    end
end

#end.

#script (python)

def gcd(a, b):
    if a == 0:
        return b
    else:
        return gcd(b % a, a)

#end.
\end{verbatim}

In Line 3, we add a function that calculates the greatest common divisor of two numbers. Integers from a logic program are directly mapped to their Lua and Python equivalents and can be returned from the function. The \texttt{gcd} function can then be used in a logic program:

To inspect the unique answer set of the program, invoke:
\texttt{gringo --text \\ gcd.lp gcd-lua.lp}
or:
\texttt{gringo --text \\ gcd.lp gcd-py.lp}
Calls to \texttt{clingo} are similar.

\begin{verbatim}
#script (lua)

function gcd(a, b)
    return b
end

#end.

#script (python)

def gcd(a, b):
    return range(a, b+1)
end

#end.
\end{verbatim}

Example 3.18. This example shows how to return multiple values from a function:

To inspect the unique answer set of the program, invoke:
\texttt{gringo --text \\ rng.lp rng-lua.lp}
or:
\texttt{gringo --text \\ rng.lp rng-py.lp}
Calls to \texttt{clingo} are similar.

\begin{verbatim}
#script (lua)

function rng(a, b)
    ret = {}
    for i = a,b do
        table.insert(ret, i)
    end
    return ret
end

#end.

#script (python)

def rng(a, b):
    return range(a, b+1)
end

#end.
\end{verbatim}

In Line 3, we add a function that emulates an interval. Instead of just returning one number, this function returns a table of numbers in Lua and a list of numbers in Python, respectively. The \texttt{rng} function can then be used in a logic program:

To inspect the unique answer set of the program, invoke:
\texttt{gringo --text \\ rng.lp rng-lua.lp}
or:
\texttt{gringo --text \\ rng.lp rng-py.lp}
Calls to \texttt{clingo} are similar.

\begin{verbatim}
#script (lua)

function rng(a, b)
    return b
end

#end.

#script (python)

def rng(a, b):
    return range(a, b+1)
end

#end.
\end{verbatim}

In Line 3, we add a function that calculates the greatest common divisor of two numbers. Integers from a logic program are directly mapped to their Lua and Python equivalents and can be returned from the function. The \texttt{gcd} function can then be used in a logic program:

To inspect the unique answer set of the program, invoke:
\texttt{gringo --text \\ gcd.lp gcd-lua.lp}
or:
\texttt{gringo --text \\ gcd.lp gcd-py.lp}
Calls to \texttt{clingo} are similar.

\begin{verbatim}
#script (lua)

function gcd(a, b)
    if a == 0 then
        return b
    else
        return gcd(b % a, a)
    end
end

#end.

#script (python)

def gcd(a, b):
    if a == 0:
        return b
    else:
        return gcd(b % a, a)

#end.
\end{verbatim}

In Line 3, we add a function that calculates the greatest common divisor of two numbers. Integers from a logic program are directly mapped to their Lua and Python equivalents and can be returned from the function. The \texttt{gcd} function can then be used in a logic program:

To inspect the unique answer set of the program, invoke:
\texttt{gringo --text \\ gcd.lp gcd-lua.lp}
or:
\texttt{gringo --text \\ gcd.lp gcd-py.lp}
Calls to \texttt{clingo} are similar.

\begin{verbatim}
#script (lua)

function gcd(a, b)
    if a == 0 then
        return b
    else
        return gcd(b % a, a)
    end
end

#end.

#script (python)

def gcd(a, b):
    if a == 0:
        return b
    else:
        return gcd(b % a, a)

#end.
\end{verbatim}

In Line 3, we add a function that calculates the greatest common divisor of two numbers. Integers from a logic program are directly mapped to their Lua and Python equivalents and can be returned from the function. The \texttt{gcd} function can then be used in a logic program:

To inspect the unique answer set of the program, invoke:
\texttt{gringo --text \\ gcd.lp gcd-lua.lp}
or:
\texttt{gringo --text \\ gcd.lp gcd-py.lp}
Calls to \texttt{clingo} are similar.
2  p(5,10).
3  rng(X,Y,@rng(X,Y)) :- p(X,Y).

The function is called in Line 3 and the result stored in predicate rng/3. The values in the table or list returned from a call to rng(X,Y) are then successively inserted.

In fact, this function behaves exactly like the interval X..Y.

An interesting use case for returning multiple values is to pull whole instances from external sources, like for example a database or some text file not already in fact format.

As we have seen in the previous example, numbers are mapped to their Lua and Python equivalents. The same holds for quoted strings. To capture constants and functions, there are the Id and Fun classes, respectively, in both Python and Lua. Both objects have a function name() to access the string representation of the constant or the name of the function term, respectively. A Fun object has an additional function args() returning the table or list of arguments, respectively, in Lua and Python. Moreover, tuples are mapped to Python’s tuple data structure and to Fun objects that have an empty name in Lua. Finally, the terms #sup and #inf are mapped to the constants Sup and Inf.

To construct terms from within the scripting language, the Id and Fun classes have a constructor to create objects (in Lua it is a global function). The Id(name) constructor simply takes the string representation of a constant as argument, and the Fun(name,args) constructor additionally takes a sequence or table of terms representing the arguments of a function. Furthermore, the global function Tuple(args) is a shortcut for Fun("",args) to create an object representing a tuple in Lua.

Example 3.19. This example shows how to inspect and create terms:

1  #script (lua)
3  Fun = gringo.Fun
5  function g(c, f)
6    n = c:name()
7    r = Fun(n, f:args())
8    return r
9  end
11  #end.

1  #script (python)
3  from gringo import Fun
5  def g(c, f):
6    n = c.name()
7    r = Fun(n, f)
8    return r
11  #end.

In Line 5, we add a function g that takes a constant and a tuple as arguments and returns a function term with the name of the constant and the tuple as arguments. Note the different handling of tuples in Lua and Python. Whereas in Lua tuples are

13 Strictly speaking there are no classes in Lua, the userdata type together with a metatable is used to emulate classes.
represented using Fun objects, in Python the tuple can directly be passed to the Fun constructor. The $g$ function can then be used in a logic program:

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

```
gringo --text \n term.lp term-lua.lp
or:
gringo --text \n term.lp term-py.lp
```

Calls to clingo are similar.

```
1 p(f, (1,2)).
2 p(g, (a,b)).
3 g(X,Y, @g(X,Y)) :- p(X,Y).
```

The function is called in Line 3 and the result stored in predicate $g/3$. Using this scheme, new terms that cannot be constructed by means of plain ASP can be created during grounding. Another interesting application might be string concatenation.

The last function of interest here is the $cmp(a,b)$ function, which compares two terms $a$ and $b$ like gringo’s built-in comparison predicates (cf. Section 3.1.8) would do. It returns a negative integer if $a < b$, zero if $a = b$, and a positive integer if $a > b$.

Example 3.20. This example shows how to implement the $max$ function:

```
#script (lua)
3 cmp = gringo.cmp
5 function max(a, b)
6     if cmp(a, b) > 0
7         return a
8     else
9         return b
10  end
11  end
#end.
```

```
#script (python)
3 from gringo import cmp
5 def max(a, b):
6     if cmp(a, b) > 0:
7         return a
8     else:
9         return b
13 #end.
```

In Line 5, we add a function $max$ that takes two terms as arguments and returns the maximum of both. We are using the $cmp$ function to compare terms here. Note that the $>$ relation cannot be applied directly because objects can be of different types, e.g., integers and strings. The $max$ function can then be used in a logic program:

```
1 p(1,2).
2 p(a,3).
3 max(X,Y, @max(X,Y)) :- p(X,Y).
```

The function is called in Line 3 and the result stored in predicate $max/3$. A respective ground fact exhibits that the constant $a$ is greater than the integer $3$.

Remark 3.11.

1. The grounder assumes that all external functions are deterministic. That is, when a function is called multiple times with the same arguments during grounding, then it should return the same values. Adding functions that do not comply with this assumption can lead to undesired results.
2. If an error occurs during the evaluation of an external function, a warning is printed and the current instance of a rule or condition is dropped. For example, this happens when the \texttt{gcd} function from Example 3.17 is applied to non-integer arguments.

3.1.15 Meta-Statements

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

Comments To annotate the source code 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 in ‘\%\*’ and ‘\*\%’. As an abstract example, consider:

\begin{verbatim}
tos(jim).  %* enclosed comment *% tos(spock).
tos(bones). % comment till end of line
tos(scotty). tos(chekov).  
\%*  
    comment over multiple lines
\*%
tos(uhura). tos(sulu).
\end{verbatim}

Show Statements Usually, only a subset of the atoms belonging to an answer set characterizes a solution. In order to suppress the atoms of “irrelevant” predicates from the output or even to show arbitrary terms, the \texttt{#show} directive can be used. There are three different kinds of such statements:

- **Show atoms**: \texttt{#show p/n}.
- **Show terms**: \texttt{#show t:L_1,\ldots,L_n}.
- **Show nothing**: \texttt{#show}.

The first \texttt{#show} statement is the most commonly used form. Whenever there is at least one statement of this form, all atoms are hidden, except for those over predicates \(p/n\) given by the respective \texttt{#show} statements. The second form can be used to show arbitrary terms. The term \(t\) is part of the output if the literals in the condition after the ‘:’ hold. Unlike the previous form, this statement does not automatically hide all atoms. To hide all atoms in this case and only show selected terms, the last statement (mnemonic: show nothing) can be added to suppress all atoms in the output.

Example 3.21. This example illustrates the common use case to selectively show atoms:

To inspect the output, invoke:
\begin{verbatim}
clingo showa.lp 0
or alternatively:
gringo showa.lp | clasp 0
\end{verbatim}
3.1 Input Language of gringo and clingo

Example 3.22. This example illustrates how to show terms:

```
1  p(1). p(2). p(3).
2  { q(X) : p(X) }.
3  a :- q(1).
4  #show a/0.
5  #show q/1.
```

Only atoms over q/1 and a appear in the output here.

Example 3.22. This example illustrates how to show terms:

```
1  p(1). p(2). p(3).
2  { holds(q(X)) : p(X) }.
3  holds(a) :- holds(q(1)).
4  #show.
5  #show X : holds(X).
```

When running this example, the same output as in the previous example is produced. This feature is especially handy when applying meta-programming techniques (cf. Section 8) where the signatures of the reified atoms are not fixed and holds(·) atoms would just clutter the output.

Remark 3.12. The second form of #show statements to show terms may contain variables. Regarding safety (cf. Page 21), it behaves similar to a rule, where the term \( t \) takes the role of the head and the condition after the colon the role of the body.

Const Statements Constants appearing in a logic program may actually be placeholders for concrete values provided by a user. An example of this is given in Section 5.1. Via the #const directive, 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 constant on the left-hand side and a term without variables, pools, and intervals on the right-hand side.

Example 3.23. This example is about using the grounder as a simple calculator:

```
1  #const x = 42.
2  #const y = f(x,z).
3  p(x,y).
```

Try running this example using the following calls:

```
gringo --text const.lp
gringo --text const.lp -c x=6 -c z=6
gringo --text const.lp -c x=6+6 -c y=6
gringo --text const.lp -c x="6+6*6"
```

Note that quotes have to be added to prevent the shell from expanding the ‘*’ in the last call or from interpreting parentheses in functions.
3.1 Input Language of gringo and clingo

External Statements  External statements are used to declare atoms that should not be subject to certain simplifications. Namely, atoms marked external are not removed from the bodies of rules, conditions, etc., even if they do not appear in the head of any rule. The main use case is to implement extensions to plain ASP solving, like multi-shot solving detailed in Section 4. An #external statement has the following form:

\[
\text{#external } A:L_1,\ldots,L_n.
\]

Here, \( A \) is an atom over some predicate and the part following the ‘:’ is a condition. The condition is instantiated to obtain a set of external atoms. Note that the condition is discarded after grounding, hence, it is a good idea to use only domain predicates or built-ins after the colon.\(^{14}\)

Example 3.24. Consider the following example:

1. \( p(1) \).
2. \( p(2) \).
3. \( p(3) \).
4. \( \text{#external } q(X) : p(X). \)
5. \( q(1) \).
6. \( r(X) :- q(X). \)

The #external statement in Line 2 gives rise to three external atoms, which appear accordingly in the text output. With these three atoms, the rule in Line 4 yields three ground instantiations, where atoms \( q(2) \) and \( q(3) \) appear in the body. Because we have the fact \( q(1) \) in Line 3, the atom \( q(1) \) is still subject to simplification and removed from the body of the respective instantiation of the rule in Line 4. The idea here is that no matter how \( q(1) \) is supplied externally, there can never be an answer set that does not contain \( q(1) \).

Remark 3.13. External statements that contain variables have very similar requirements regarding safety as rules (cf. Page 21). The atom \( A \) takes the role of the head and the condition after the colon the role of the body.

Program Parts  A logic program can be organized in multiple program parts. To begin a new program part, we write a statement

\[
\text{#program } p(s_1,\ldots,s_n).
\]

where \( p \) is the program part name and the parameters \( s_i \) are constants. If \( n \) is zero, then the parentheses can be omitted. All rules, external statements, and show statements for terms up to the next #program statement or the end of the file belong to the program part \( p/n \). Rules that are not subject to any such directive are included in the base/0 part.

The default behavior of gringo is to ground (and solve in the case of clingo) the base/0 part. Using the scripting API (cf. Section 4), we can ground other parts.

\(^{14}\)Non-domain predicates are supported too because in some situations it might be inconvenient to specify domain predicates.
than base/0 too. Occurrences of constants that are parameters of a part are replaced with ground terms when instantiating the program part.

**Example 3.25.** The following example shows how to instantiate program parts:

```plaintext
1 a.
2 #program a(s,t).
3 b(s,t).
4 #program base.
5 c.
```

The above program is organized in two parts, base/0 and a/2. Note that the fact in the first line is implicitly in the base/0 part. Solving the program as is results in answer set \{a,c\}, because the base/0 part is instantiated by default. Scripts to instantiate the a/2 part as well are as follows:

```plaintext
#script (lua)

function main(prg)
  p = {}
  add(p, \{"base",\{\}})
  add(p, \{"a",\{1,3\}\})
  prg:ground(p)
  prg:solve()

#end.
```

In Line 9, the script grounds the base/0 part (Line 7) as well as the a/2 part with parameters 1 and 3 (Line 8). The call in Line 10 is essential to solve the program with clingo, and even in gringo some post-processing happens, e.g., printing the symbol table of the smodels format [78].

**Remark 3.14.** Program parts are mainly interesting for incremental grounding and solving of logic programs detailed in Section 4. For single-shot solving, program parts are not needed. The feature is merely listed for completeness here.

**Include Statements** Include statements allow for including files from within another file. They have the form

```plaintext
#include "file".
```

where file is a path to another encoding file. When including a file it is first looked up relative to the current working directory. If it is not found there, then it is looked up relative to the file it was included from. Note that program part declarations do
not affect the inclusion of files, that is, including a file is equivalent to passing it on
the command line.

Example 3.26. Suppose that we have a file include.lp with the following statement:

```
1  #include "bird.lp".
```

We can simply pass the file on the command line to include file bird.lp from Ex-
ample 3.2. Since files are included from the current working directory as well
as relative to the file with the include statement, an invocation like \texttt{clingo
examples/include.lp}\ works with either of the following directory layouts:

```
|-- bird.lp
```

```
|-- examples
  |-- include.lp
```

```
|-- examples
  |-- bird.lp
  `-- include.lp
```

3.2 Input Language of clasp

Solver \texttt{clasp} [36] (or \texttt{clingo --mode=clasp}) accepts logic programs
in \texttt{smodels} format [78], SAT and MaxSAT instances in \texttt{DIMACS-cnf}15 and
\texttt{DIMACS-wcnf}16 format, and PB problems in OPB/WBO17 format.

For ASP solving, \texttt{clasp} is typically invoked in a pipe reading a logic program
output by \texttt{gringo} (or \texttt{clingo}):

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

```
clingo [ options | files | number ]
```

Note that \texttt{number} may be provided to specify a maximum number of answer sets
to be computed, where 0 makes \texttt{clasp} compute all answer sets. This maximum
number can also be set via option --models or its abbreviation \texttt{-n} (cf. Section 6.3).
By default, \texttt{clasp} computes one (optimal) answer set (if it exists).

To solve a problem in one of the supported formats stored in a file, an invoca-
tion of \texttt{clasp} looks as follows:

```
classp [ options | number ] file
```

```
clingo --mode=clasp [ options | number ] file
```

---

16http://www.maxsatsat.udl.cat/12/requirements/index.html
17http://www.cril.univ-artois.fr/PB12/format.pdf
In general, \texttt{clasp} autodetects the input format. However, option \texttt{--opt-sat} is necessary to distinguish a MaxSAT instance in DIMACS-wcnf format from a plain SAT instance in DIMACS-cnf format.
4 Multi-shot Solving

This section is not yet ready for publishing and will be included in one of the forthcoming editions of this guide.

Information on multi-shot solving with clingo can be obtained at the following references.

- [33, 34]
- /examples/scripting/ in gringo/clingo distribution
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).\(^{18}\) While the first problem could likewise be solved within neighboring paradigms, the second one requires checking reachability, something that is quite cumbersome to encode in either Boolean Satisfiability [10] or Constraint Programming [73]. The third problem coming from the area of planning illustrates incremental solving with clingo.

5.1 \( n \)-Coloring

As already mentioned in Section 2, it is custom in ASP to provide a uniform problem definition [66, 68, 75]. 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., [70]), and thus it seems unlikely that a worst-case polynomial time algorithm can be found. In view of this, it is convenient to encode the particular problem in a declarative problem solving paradigm like ASP, where efficient off-the-shelf tools like gringo and clasp are available.

5.1.1 Problem Instance

We consider directed graphs specified via facts over predicates node/1 and edge/2.\(^{19}\) The graph in Figure 3 is represented by the following set of facts:

\[
\begin{align*}
1 & \text{ % Nodes} \\
2 & \text{ node}(1..6). \\
3 & \text{ % (Directed) Edges} \\
4 & \text{ edge}(1,(2;3;4)). \text{ edge}(2,(4;5;6)). \text{ edge}(3,(1;4;5)). \\
5 & \text{ edge}(4,(1;2)). \text{ edge}(5,(3;4;6)). \text{ edge}(6,(2;3;5)).
\end{align*}
\]

Recall from Section 3.1 that ‘..’ and ‘;’ in the head expand to multiple rules, which are facts here. Thus, the instance contains 6 nodes and 17 directed 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 [61]. As \( n \)-coloring has a rather simple pattern, the following encoding does not contain any define part:

\(^{18}\)The above examples are also discussed in [32]; you may also like the videos at [72].

\(^{19}\)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 6 Nodes and 17 Edges.

1 % Default
2 #const n = 3.
3 % Generate
4 { 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 directive, described in Section 3.1.15, 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 shortcut for count aggregates (cf. Section 3.1.12). For our example graph and 1 substituted for X, we obtain the following ground rule:

    #count { 0,color(1,1) : color(1,1);
             0,color(1,2) : color(1,2);
             0,color(1,3) : color(1,3) } = 1.

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 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...
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 $\text{FP}^{\text{NP}}$-complete (cf. [70]), 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 `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).
Figure 5: The Graph from Figure 3 along with Edge Costs.

5.2 Traveling Salesperson

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

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 [61], we encode this subproblem via the following non-ground rules:

1 % Generate
2 { cycle(X,Y) : edge(X,Y) } = 1 :- node(X).
3 { 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 #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:

#count { 0,cycle(1,2) : cycle(1,2),
0,cycle(1,3) : cycle(1,3),
0,cycle(1,4) : cycle(1,4) } = 1.
#count { 0,cycle(3,1) : cycle(3,1),
0,cycle(4,1) : cycle(4,1) } = 1.

The full ground program is obtained by invoking:
clingo --text \ ham.lp min.lp \ costs.lp graph.lp
or alternatively:
gringo --text \ ham.lp min.lp \ costs.lp graph.lp
5.2 Traveling Salesperson

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. This definition leads to positive recursion among the ground instances of reached/1, in which case a ground program is called non-tight [23, 24]. The fact that the atoms of an answer set must be derivable is here exploited to make sure that all 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 states that answer sets should be projected to instances of cycle/2, as only they are characteristic for a solution. So far we have not considered edge costs. 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:

```
% Optimize
#minimize { C,X,Y : cycle(X,Y), cost(X,Y,C) }.
```

Here, edges belonging to the cycle are weighted according to their costs. After grounding, the #minimize statement in Line 12 ranges over the 17 instances of cycle/2, one for each weighted edge in Figure 5.

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 with answer sets until we cannot find a strictly better one anymore. By default, clasp (or clingo) successively enumerates better answer sets with respect to the provided optimization statements (cf. Section 3.1.13). 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)
Optimization: 11
```

Given that no answer is obtained after the second one, we know that 11 is the optimum value, but there might be further optimal answer sets that have not been computed yet. To compute all optimal answer sets, we can change clasp’s optimization mode using option ‘--opt-mode=optN’. In this mode, clasp first prints the tentative answer sets where optimality is not yet proven and afterwards prints the optimal answer sets. Note that the first optimal answer set is printed twice in this mode. To omit tentative answer sets in the output and only print optimal answer sets, we can add option ‘--quiet=1’.

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 smodels format [78]. Furthermore, if there are multiple optimization statements or priorities, respectively, clasp (or clingo) will report separate optimization values ordered by priority.

5.3 Blocks World Planning

The blocks world is a well-known planning domain where finding shortest plans has received particular attention [56]. With the single-shot grounding and solving approach we have used in the previous examples, 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, clingo’s scripting API (cf. Section 4)
can be used to solve problems in an incremental fashion. Because planning problems where the search horizon is gradually increased are quite common, clingo provides an easy to use built-in solving and grounding mode for such problems. We use blocks world planning to illustrate the exploitation of clingo’s incremental computation mode.

5.3.1 Problem Instance

As with the other two problems above, an instance is given by a set of facts, here over 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:

```
1 % Sussman Anomaly
2 %
3 block(b0).
4 block(b1).
5 block(b2).
6 %
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. Here we use (uninterpreted) function on/2 to illustrate another important feature available in gringo and clingo, namely, the possibility of instantiating variables to compound terms.

---

Blocks world instances worldi.lp for i ∈ {0, 1, 2, 3, 4} are adaptations of the instances provided at [22].
5.3.2 Problem Encoding

Our blocks world planning encoding for clingo makes use of \#program directives defining subprograms base, step(t), and check(t), separating the encoding into a static part, a specification of state transitions, and a part for checking the goal situation and state constraints, respectively. The base part is instantiated at step zero, the step(t) part is instantiated for steps \( t > 0 \), and the check(t) part for steps \( t \geq 0 \).

Each of them can be further refined into generate, define, test, and display constituents, as indicated in the comments below:

1 #include <incmode>.

3 #program base.
4 % Define
5 location(table).
6 location(X) :- block(X).
7 holds(F,0) :- init(F).

9 #program step(t).
10 % Generate
11 { move(X,Y,t) : block(X), location(Y), X != Y } = 1.
12 % Test
13 :- move(X,Y,t), holds(on(A,X),t-1).
14 :- move(X,Y,t), holds(on(B,Y),t-1), B != X, Y != table.
15 % Define
16 moved(X,t) :- move(X,Y,t).
17 holds(on(X,Y),t) :- move(X,Y,t).
18 holds(on(X,Z),t) :- holds(on(X,Z),t-1), not moved(X,t).

20 #program check(t).
21 % Test
22 :- query(t), goal(F), not holds(F,t).

24 % Display
25 #show move/3.

The first line enables clingo's incremental computation mode. Next, the base part in Line 3–7 defines blocks and the constant table as instances of predicate location/1. Moreover, we use instances of init/1 to initialize holds/2 for the initial state at step 0, thus specifying the setup before the first state transition. Note that variable F is instantiated to compound terms over function on/2.

The step subprogram in Line 9–18 declares constant t as a placeholder for step numbers in the program part below. Remember that the step(t) part is not instantiated for \( t = 0 \). Hence, it can always refer to two successive time steps at \( t \).
5.3 Blocks World Planning

and \(t-1\). The generate rule in Line 11 states that exactly one block \(X\) must be moved to a location \(Y\) (different from \(X\)) for each state transition at step \(t\). The integrity constraints in Line 13 and 14 are used to test whether moving block \(X\) to location \(Y\) is possible at step \(t\). The first integrity constraint ensures that block \(X\) cannot be moved if there is another block \(A\) on top of it. Furthermore, the second integrity constraint excludes all moves where the target block \(Y\) is occupied by some other block \(B\). Because the number of blocks of that can be put on the table is not limited, the condition is only checked if \(Y\) is a block, viz., \(\neq\) \(table\). Also, this constraint allows for void moves, that is, it only eliminates solutions where the block \(X\) being moved is different from \(B\), viz., \(\neq\). The rule in Line 16 marks the block that is moved via predicate \(moved/1\). Finally, the rule in Line 17 propagates a move to the state at step \(t\), while the rule in Line 18 states that a block \(X\) stays at a location \(Z\) if it is not moved.

The subprogram \texttt{check} in Line 20–22 specifies goal conditions to be checked for each state. Note the use of atom \(query(t)\). This atom, provided in incremental mode, allows for posting queries; for each incremental step, there is only one atom over \(query/1\) that is true, namely, \(query(t)\) for the current step \(t\).

Note that the \#\texttt{show} meta-statement in Line 25 does not belong to any program part but affects the visibility of atoms in all program parts. Furthermore, rules not subject to a \#\texttt{program} directive are associated with the base program by default. Hence, we do not have to use such directives in instance files because the base part is exactly where we want the facts from an instance to be included.

Finally, let us stress important prerequisites for obtaining a well-defined incremental computation result from \texttt{clingo}. First, the ground instances of head atoms of rules in each step must be pairwise disjoint. This is the case for our encoding because atoms over \(move/3\), \(moved/2\), and those over \(holds/2\) include \(t\) as an argument in the heads of rules in Line 11–18. As the smallest step number to replace \(t\) with is 1, there is also no clash with the ground atoms over \(holds/2\) obtained from the head of the static rule in Line 7. Further details on the sketched requirements and their formal background can be found in [29]. Arguably, many problems including a mutable bound can be encoded such that this prerequisite applies. Some attention should of course be spent on putting rules into the right program parts.

5.3.3 Problem Solution

We can now use \texttt{clingo} to \textit{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:

\begin{verbatim}
Answer: 1
move(b2,table,1) move(b1,b0,2) move(b2,b1,3)
\end{verbatim}

This unique answer set tells us that the given problem instance can be solved by moving block \(b2\) to the \texttt{table} in order to then put \(b1\) on top of \(b0\) and finally \(b2\)

\footnote{Not so for \#\texttt{show} statements to show terms. These are tied to the program parts they occur in.}
This solution is computed by clingo in four grounding and solving steps, where, starting from the base and the check part in which \( t \) is replaced with 0, the constant \( t \) is successively replaced with step numbers 1, 2, and 3 in the step and check parts. While the goal conditions in the check part cannot be fulfilled in steps 0, 1, and 2, clingo stops its incremental computation after finding an answer set in step 3. The scheme of iterating steps until finding some answer set is the default behavior of the incremental mode.

Sometimes it might be interesting to inspect the grounding of an incremental program. This can be achieved using option \(--lparse-debug=plain\). Adding this option, clingo solves as usual but additionally prints the grounded rules to the standard error stream. The rules are printed in the same format as the text output but preceded with ‘%’. 
6 Command Line Options

In this section, we briefly describe the meaning of some selected command line options supported by gringo (Section 6.1), clingo (Section 6.2), and clasp (Section 6.3). Each of these tools display their available options when invoked with flag --help or -h. 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 [53]. 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 | files ]
```

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.

--verbose [=n], -V

Print additional (progress) information during computation. Verbosity level one and two are currently not used by gringo. The flag implies level three. Level three prints internal representations of the logic program.

--const, -c c=t

Replace occurrences (in the input program) of constant c with term t. This overrides constant definitions in a source file without a warning.

--text, -t

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

Note that our description of command line options is based on gringo and clingo series 4 as well as clasp series 3. 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.
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 [ options | files | number ]

The optional numerical argument allows for specifying the maximum number of answer sets to be computed (0 standing for all answer sets). As with gringo, the 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 6.3. In particular, (long) options --help and --version make clingo print the desired information and exit, while --text instructs 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 6.3, clingo has a single additional option:

--lparse-rewrite
Can be used in conjunction with the --text option to print a program in a (human-readable) similar to the smodels format, which is otherwise passed to the solver.

--lparse-debug={none, plain, lparse, all}
This option enables additional debugging output to the standard error stream. The available arguments are:

    none   No additional output is printed.
    plain  Prints rules as with --text but prefixed with %.
    lparse Prints rules as with --text and --lparse-rewrite but prefixed with %%
    all    Combines argument plain and lparse.

--warn=[no-]w This option can be used to enable and disable warnings. To disable a warning, the argument has to prefixed with no-. To enable or disable multiple warnings, this option can be passed multiple times with different arguments. By default all warnings are enabled. The available values for argument w are:

    file-included  See Section 7.2.1.
    variable-unbounded  See Section 7.2.2.
    operation-undefined  See Section 7.3.1.
    atom-undefined  See Section 7.3.2.

When calling gringo without options, it outputs a ground program in smodels format [78], which is a common input language for propositional ASP solvers.
6.3 clasp Options

--mode=m
Choose the mode in which clingo should run. Available values for m are:

clingo  Explicitly select clingo mode (the default).
gringo  In this mode clingo behaves like gringo.
lparse  In this mode clingo behaves like clasp.

Finally, the default command line when invoking clingo consists of all clasp defaults (cf. Section 6.3).

6.3 clasp Options

Stand-alone clasp [36] is an ASP solver for ground logic programs that can also be used as a SAT, MaxSAT, or PB solver (cf. Section 3.2). An abstract invocation of clasp looks as follows:

```plaintext
clash [ options | files | number ]
```

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 --models or its short form -n.) If neither a filename nor an option that makes clasp exit (see below) is provided, clasp reads from the standard input.\(^{23}\) In fact, it is typical to use clasp in a pipe with gringo in the following way:

```plaintext
gringo [ options | files ] | clasp [ options | number ]
```

In such a pipe, gringo instantiates an input program and outputs the ground rules in smodels format, which is then fed to clasp that computes and outputs answer sets. Note that clasp offers plenty of options to configure its behavior. In the following, we present only some important options and categorize them according to their functionalities.

6.3.1 General Options

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

--help[=n], -h
Print help information and exit.
Argument n determines the level of detail that is shown. If n is not given or is equal to 1, only major options are shown. Level n=2 also prints advanced search options. Finally, n=3 prints the full help information.

\(^{23}\)In earlier versions of clasp, filenames had to be given via option --file or its short form -f.
6.3 \textit{clasp} Options

\texttt{--version, -v}
Print version information and exit. The version information also includes whether or not \textit{clasp} was built with support for parallel solving via multithreading.

\texttt{--verbose=[n], -V}
Configure printing of (progress) information during computation. Argument \(n=0\) disables progress information, while \(n=1\) and \(n=2\) print basic information. Extended information is printed for \(n>2\), where levels 4 and 5 are only relevant when solving disjunctive logic programs. Finally, the flag \(-V\) implies the largest available verbosity level.

\texttt{--outf=n}
Configure output format. Available values for \(n\) include 0 for \textit{clasp}'s default output format, 1 for solver competition (ASP, SAT, PB) output, and 2 for output in JSON\textsuperscript{24} format.

\texttt{--quiet=[models[, costs][, calls]], -q}
Configure printing of computed models, associated costs (in case of optimization), and individual call statistics (for multi-shot solving). Arguments are integers in the range 0..2, where 0 means print all, 1 means print last, and 2 means do not print any models, costs, or individual call statistics. If \(--quiet\) or \(-q\) is given as a flag, all arguments are implicitly set to 2.

\texttt{--stats=[1,2]], -s}
Maintain and print basic (1) or extended (2) statistic information.

\texttt{--time-limit=t}
Force termination after \(t\) seconds.

\texttt{--solve-limit=n[, m]}
Force termination after either \(n\) conflicts or \(m\) restarts.

\texttt{--pre}
Run ASP preprocessor then print preprocessed input program and exit.

\texttt{--print-portfolio}
Print default portfolio and exit (cf. \texttt{--parallel-mode}).

6.3.2 Solving Options
The options listed below can be used to configure the main solving and reasoning strategies of \textit{clasp}.

\texttt{--models,-n n}
Compute at most \(n\) models, \(n=0\) standing for compute all models.

\textsuperscript{24}http://json.org/
6.3 *clasp* Options

**--project**

Project answer sets to named atoms and only enumerate unique projected solutions [42].

**--enum-mode, -e mode**

Configure enumeration algorithm applied during solving. Available values for mode are:

- **bt** Enable backtrack-based enumeration [37].
- **record** Enable enumeration based on solution recording. Note that this mode is prone to blow up in space in view of an exponential number of solutions in the worst case.
- **domRec** Enable subset enumeration via domain-based recording (cf. Section 9).
- **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.
- **auto** Use _bt_ for enumeration and _record_ for optimization.

Note: The option is only meaningful if **--models** is not equal to 1. Furthermore, modes **brave** and **cautious** require **--models=0**, which is also the default in that case.

**--opt-mode=mode**

Configure handling of optimization statements. Available values for mode are:

- **opt** Compute an optimal model (requires **--models=0**).
- **enum** Enumerate models with costs less than or equal to some fixed bound (cf. **--opt-bound**).
- **optN** Compute optimum, then enumerate optimal models.
- **ignore** Ignore any optimization statements during computation.

**--opt-bound=n1[,n2,n3...]**

Initialize objective function(s) to minimize with _n1 [,n2,n3...]_.

**--opt-sat**

Treat input in DIMACS-(w)cnf format as MaxSAT optimization problem.

**--parallel-mode, -t n[,mode]**

Enable parallel solving with _n_ threads [44], where _mode_ can be either **compete** for competition-based (portfolio) search or **split** for splitting-based search via distribution of guiding paths.
6.3 clasp Options

6.3.3 Fine-Tuning Options

The following incomplete list of options can be used to fine-tune certain aspects of clasp. For a complete list of options, call clasp with option \--help=3.

\--configuration=c

Use c as default configuration, where c can be:

- **frumpy**: Use conservative defaults similar to those used in earlier clasp versions.
- **jumpy**: Use more aggressive defaults.
- **tweety**: Use defaults geared towards typical ASP problems.
- **trendy**: Use defaults geared towards industrial problems.
- **crafty**: Use defaults geared towards crafted problems.
- **handy**: Use defaults geared towards large problems.
- \textless\texttt{file}\textgreater: Use configuration file to configure solver(s).

Note that using a configuration file enables freely configurable solver portfolios in parallel solving. For an example of such a portfolio, call clasp with option \--print-portfolio.

\--opt-strategy=\{bb,usc\}[,[n]]

Configure optimization strategy. Use either branch-and-bound-based optimization [30] (bb) or unsatisfiable-core-based optimization [1] (usc). The optional argument n can be used to fine-tune the selected strategy. For example, bb,1 enables hierarchical (multi-criteria) optimization [30], while usc,1 enables some form of preprocessing during unsatisfiable-core-based optimization. For further details, call clasp with option \--help=2. Finally, note that the optimization strategy can be set on a per-solver basis in the context of parallel solving, thus allowing for optimization portfolios.

\--restart-on-model

Restart the search after finding a model. This is mainly useful during optimization because it often ameliorates the convergence to an optimum.

\--heuristic=\{Berkmin,Vmtf,Vsids,Unit,None,Domain\}

Use BerkMin-like decision heuristic [55] (with argument Berkmin), Siege-like decision heuristic [74] (with argument Vmtf), Chaff-like decision heuristic [67] (with argument Vsids), Smodels-like decision heuristic [76] (with argument Unit), or (arbitrary) static variable ordering (with argument None). Finally, argument Domain enables a domain-specific decision heuristic as described in Section 9.

\--save-progress[^n]

Enable alternative sign heuristic based on cached truth values [71] if available. Cache truth values on backjumps > n.
6.3 clasp Options

|--restarts, -r sched
Choose and parameterize a restart policy, where sched can be:

  no  Disable restarts.
  F, n Run fixed sequence, restarting every n conflicts.
  *, n, f Run a geometric sequence [19], restarting every n * f^i conflicts,
           where i is the number of restarts performed so far.
  +, n, m Run an arithmetic sequence, restarting every n + m * i conflicts,
           where i is the number of restarts performed so far.
  L, n Restart search after a number of conflicts determined by a universal
       sequence [65], where n constitutes the base unit.
  D, n, f Use a dynamic policy similar to the one of glucose [4]. Given the
           n most recently learned conflict clauses and their average quality
           Q_n, a restart is triggered if Q_n * f > Q, where Q is the global
           average quality.

The geometric and arithmetic sequences take an optional limit lim > 0 to
enable a nested policy [9]. If given, the sequence is repeated after lim + j
restarts, where j counts how often the sequence has been repeated so far.

|--eq=n
Run equivalence reasoning [39] for n iterations, n = -1 and n = 0 standing
for run to fixpoint or do not run equivalence reasoning, respectively.

|--trans-ext= {choice, card, weight, scc, integ, dynamic, all, no}
Compile extended rules [76] into normal rules (cf. Section 3.1.2). Arguments
choice, card, and weight state that all “choice rules”, “cardinality rules”
or “weight rules”, respectively, are to be compiled into normal rules, while
all means that all extended rules and no that none of them are subject to
compilation. If argument dynamic is given, clasp heuristically decides
whether or not to compile individual “cardinality” and “weight rules”. Finally,
scc limits compilation to recursive “cardinality” and “weight rules”, while
integ only compiles those “cardinality rules” that are integrity constraints.

|--sat-prepro=[0, 1, 2, 3] [,.x1]...[,.x5]
Configure SatElite-like preprocessing [18]. Argument 0 (or no) means that
SatElite-like preprocessing is not to be run at all, while 1 enables basic pre-
processing and 2 and 3 successively enable more advanced preprocessing in-
cluding, for example, blocked clause elimination [60]. If --sat-prepro is
given as a flag, 2 is assumed. The optional arguments x1,...,x5 can be used
to set fine grained limits, for example, regarding iterations and runtime.
For further details, call clasp with option --help=2.

Let us note that switching the above options can have dramatic effects (both
positively and negatively) on the search performance of clasp. If performance
bottlenecks are observed, it is worthwhile to first give the different prefabricated default configurations a try (cf. --configuration). Furthermore, we suggest trying different heuristics and restart sequences. For a brief overview on manual fine-tuning, see [41]. Automatic configuration methods are described in Section 12.
7 Errors, Warnings, and Infos

This section explains the most frequent errors, warnings, and info messages related to inappropriate inputs or command line options. All messages are printed to the standard error stream. Errors lead to premature termination, while warnings and info messages provide hints at possibly corrupt input that can still be processed further.

7.1 Errors

Most of the errors in the following start with the prefix:

```plaintext
*** ERROR: (System)
```

where `System` is either `gringo`, `clingo`, or `clasp` depending on the system. In the following sections, we use `Error` to denote this prefix. All of the errors with this prefix are fatal and lead to immediate termination.

7.1.1 Parsing Command Line Options

We start with errors emitted during command line parsing, which are handled equally by `gringo`, `clasp`, and `clingo`. All our tools try to expand incomplete (long) options to recognized ones. Parsing command line options may nonetheless fail due to the following three reasons:

```plaintext
Error: In context 'Context': unknown option: 'Option'
Error: In context 'Context': \
ambiguous option: 'Option' could be:
   Option1
   Option2
   ...
Error: In context '<Context>': \
'Arg': invalid value for: 'Option'
*** Info : (System): Try '--help' for usage information
```

The first error means that the option `Option` could not be expanded to one that is recognized. While the second error expresses that the result of expanding `Option` is ambiguous. It is followed by a list of option candidates `Option1, Option2, ..`, all sharing the same prefix. Finally, the third error occurs if the argument `Arg` is invalid for option `Option`. All three error messages include a context `Context` in which the option is parsed. Often, this is simply the system name but can also be the name of a configuration in a portfolio file or the string `tester` for errors in options regarding the disjunctive tester. The last line is printed in all three cases. It indicates that option `--help` can be used to display the available options and their arguments.
7.1.2 Parsing and Checking Logic Programs

Next, we consider errors emitted during the parsing and checking of logic programs. Unlike the error messages in the previous section, such errors include location information to ease finding and fixing the problem. Each of the error messages below begins with a location followed by the string `error` and a short description of the error in `Message`:

```
File:Line:Column-Column: error: Message
Information
File:Line:Column-Column: note: Message
Information
...
```

The location refers to a string in a source file, specified by file name `File`, line number `Line`, and beginning and ending column number `Column` (column `n` refers to the `n`-th symbol in a line). Error messages are sometimes followed by further descriptions in the string `Information` indented by two spaces. An optional list of similarly structured notes, discernible via the string `note`, can follow this part. Such notes typically refer to locations that are in conflict with the object referred to in the location of the error message. Multiple error messages of this kind might be reported; each error message is terminated with two newlines after the notes.

**Logic Program Parsing** We start our description with errors that may be encountered during parsing, where the following one indicates a syntax error in the input:

```
Location: error: syntax error, unexpected Token
```

To correct this error, please investigate the indicated location and check whether something looks strange there (like a missing period, an unmatched parenthesis, etc.). Note that the parser tries to recover from a syntax error. This typically means that everything up to the next period is ignored.

**Safety Checking** The next error occurs if an input program is not safe:

```
Location: error: unsafe variables in Rule
Location: note: 'Var' is unsafe
...
```

Along with the error message, the affected rule `Rule` and a list of all unsafe variable occurrences `Var` are reported. The first action to take usually consists of checking whether variable `Var` is actually in the scope of any atom (in the positive body of `Rule`) that can bind it.\(^{25}\) Also check for variables that occur in aggregate elements

---

\(^{25}\)Recall from Section 3.1.7 and 3.1.8 that variables in the scope of built-in arithmetic functions are only bound by their corresponding atoms in some special cases and that built-in comparison predicates do not bind variables.
7.1 Errors

(cf. Section 3.1.12) or conditional literals (cf. Section 3.1.11); you might have to
bind them with additional positive atoms in the conditions.

**Script Execution and Parsing** If an error in an embedded script occurs (cf. Section 3.1.14), the following error message is printed:

```
Location: error: failed to execute script:
Information
...
```

The information printed depends on the error that occurred when executing the embedded script. This can for example be parse errors or errors that occurred when executing the script. Typically, the information contains a trace where the error occurred.

**Defining Constants** There are three errors associated to `#const` statements (cf. Section 3.1.15).

```
Location: error: cyclic constant definition:
  Constant
Location: note: cycle involves definition:
  Constant
...

Location: error: redefinition of constant:
  Constant
Location: note: constant also defined here:
  Constant
...
```

The strings `Constant` provide the affected `#const` statements. The first error is printed if the statements rely on each other cyclically. Each statement involved in the cycle is printed in the corresponding notes. The second error message is printed if a constant is defined more than once. The location of the conflicting definition is printed in the note.

If at least one of the errors above is reported, then `gringo` or `clingo` terminates after parsing and checking with the error message:

```
Error: grounding stopped because of errors
```

**Remark 7.1.** No more than 20 errors are printed. If this limit is exceeded, the application stops parsing or safety checking and terminates.

### 7.1.3 Parsing Logic Programs in `smodels` Format

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

Error: parse error in line Line: Message

This error means that the input does not comply with smodels' numerical format [78]. If you are using gringo to ground logic programs, this error should never occur.

7.1.4 Multi-shot Solving

The following error is issued by (embedded) clasp if an atom is defined (it appears in the head of a rule) in two different grounding steps:

Error: redefinition of atom <Atom, Id>

Information

where Atom is the string representation of the atom that is redefined and Id is the unique identifier of the atom introduced when translating the logic program into smodels format. If the scripting API (cf. Section 4) is used for grounding, then the error message is followed by a trace, indicating the source code location where the program has been grounded.

Remark 7.2. Only the case that an atom is redefined is checked by clasp. The case when there is a positive cycle over two or more incremental steps is not detected, which possibly leads to unwanted answer sets.

7.2 Warnings

This section describes warnings that may be reported by gringo or clingo. Unlike errors, warnings do not terminate the application but rather hint at problems, which should be investigated. A program with warnings might lead to unexpected results; there are no guarantees regarding the semantics of such programs. Most warnings have a similar format as the errors described in Sections 7.1.2; the only difference is that the location is followed by the string warning.

Remark 7.3. No more than 20 warnings are printed. If this limit is exceeded, you should definitely fix some warnings.

7.2.1 File Included Multiple Times

If a file is included multiple times, either on the command line or with an include directive, then the following warning is emitted:

Location: warning: already included file: Filename

Remark 7.4. Only the first include of a file is considered. All additional includes are ignored.
7.2.2 Unbounded CSP Variables

In the current implementation, a bound has to be supplied for each CSP variable (cf. Section 11.1). For variables with bounds, the following warning is issued:

```
warning: unbounded constraint variable:
  domain of Variable is set to Domain
```

where Variable is the variable lacking a domain specification and Domain is an (arbitrary) domain chosen for the variable.

7.3 Infos

This section describes information messages that may be reported by gringo or clingo. Info messages indicate issues in the input that have a well defined semantics but are possibly unintended by the user.

An information message is preceded with the string info.

Remark 7.5. Up to 20 info messages are printed. There might me further issues but these will be silently ignored.

7.3.1 Undefined Operations

These may occur within an arithmetic evaluation (cf. Section 3.1.7) or if an error occurs while evaluating an external function (cf. Section 3.1.14):

```
Location: info: term undefined:
  Term
  ...
```

It typically means that either a (symbolic) constant or a compound term (over an uninterpreted function) has occurred in the scope of some built-in arithmetic function. The string Term provides the term that failed to evaluate. The message might be followed by further notes. For example, if the evaluation of an external function failed, by a trace indicating the location of the error within the source code of the external function. Typically, it is simple to fix occurrences of this message - for example, if the term ‘X/Y’ causes a message, it can be silenced by adding the comparison literal ‘Y!=0’ to the body of a rule (or condition). We suggest to silence all of these message in this manner and not simply to disable the message.

Remark 7.6. Instantiations of rules, #show statements, #external statements, weak constraints, aggregate elements, and conditional literals that contain undefined terms are discarded.

7.3.2 Undefined Atoms

This message is emitted if an atom appears in the body of a rule or condition that is never defined in the head of a rule or external statement:
where $\text{Atom}$ is the atom occurrence without a definition. Often, this error indicates that a predicate has been misspelled or that an argument has accidentally been omitted.
8 Meta-Programming

This section is not yet ready for publishing and will be included in one of the forthcoming editions of this guide.

Information on meta-programming can be obtained at the following references.

- gringo 4 ships with the small tool reify to reify logic programs in lpars format and some examples
  - https://sourceforge.net/p/potassco/code/HEAD/tree/trunk/gringo/examples/reify/

- see below for information on meta-programming with gringo 3
  - [35]
  - http://www.cs.uni-potsdam.de/wv/metasp
9 Heuristic-driven Solving

clasp and clingo provide means for incorporating domain-specific heuristics into ASP solving. This allows for modifying the heuristic of the solver from within a logic program or from the command line. A formal description can be found in [40].

The framework is implemented as a new heuristic, named Domain, that extends the Vsids heuristics of clasp and can be activated using option --heuristic=Domain (cf. Section 6.3.3). In what follows, we first describe how to modify the solver's heuristic from within a logic program, and then we explain how to apply modifications from the command line.

9.1 Heuristic Programming

Heuristic information is represented within a logic program via the dedicated predicate _heuristic. Different types of heuristic information can be controlled with the modifiers sign, level, true, false, init and factor. We introduce them below step by step.

9.1.1 Heuristic modifier sign

The modifier sign allows for controlling the truth value assigned to variables subject to a choice within the solver.

The Domain heuristic associates with each atom an integer sign value, which by default is 0. When deciding which truth value to assign to an atom during a choice, the atom is assigned to true, if its sign value is greater than 0. If the sign value is less than 0, it is assigned to false. And if it is 0, the sign is determined by the default sign heuristic.

In order to associate a positive sign with atom a, we can use the heuristic atom _heuristic(a,sign,1). This associates a positive sign with a and tells the solver that upon deciding the atom a, it should be set to true.

Example 9.1. Consider the following program:

```prolog
_heuristic(a,sign,1).
{a}.
```

At the start of the search, the solver propagates the heuristic atom and updates its heuristic knowledge about atom a. Then, it has to decide on a, making it either true or false. Following the current heuristic knowledge, the solver makes a true and returns the answer set {_heuristic(a,sign,1),a}. ■

Remark 9.1. The result would be the same if in the heuristic fact we used instead of 1 any positive integer. ■

Example 9.2. In the next program, the _heuristic fact gives a a negative sign and thus asserts that when deciding upon a it should be set to false:

To inspect the output, invoke:

```
clingo psign.lp \--heuristic=Domain
```
or alternatively:

```
gringo psign.lp \clasp \--heuristic=Domain
```
9.1 Heuristic Programming

To inspect the output, invoke:
```
clingo nsign.lp \n--heuristic=Domain
```
or alternatively:
```
gringo nsign.lp \n| clasp \n--heuristic=Domain
```

As above, the solver starts with propagating the heuristic fact, then updates its heuristic knowledge, decides on atom \( a \) making it false, and finally returns the answer set \{ \_heuristic\(a\),sign,\(-1\)\}.

These two examples illustrate how the heuristic atoms allow for modifying the decisions of the solver, leading to either finding first the answer set with \( a \) or the one without it. However, as long as heuristic atoms appear only in the head of rules, the program’s overall answer sets remain the same (modulo heuristic atoms). For example, if we ask for all answer sets in Example 9.1, we obtain one without \( a \) and one with \( a \), and the same happens with Example 9.2, although in this case the answer sets are computed in opposite order.

9.1.2 Showing heuristic information

For the heuristic atoms to take effect, they as well as the atoms to which they refer to must be shown in the logic program.\(^{26}\) For example, if we add to Example 9.2 the line:
```
#show a/0.
```
then the heuristic atom is not shown, and the solver operates as if it would normally do with the \texttt{Vsids} heuristic. The same would happen if instead we only added:
```
#show \_heuristic/3.
```
because then the atom \( a \), appearing in the heuristic atom, would not be shown.

\textbf{Remark 9.2.} Printing heuristic atoms in the output of \texttt{clingo} or \texttt{clasp} may often be a burden. To overcome this issue, the command line option \texttt{--out-hide-aux} allows us to suppress printing atoms starting with underscore ‘\_’ (without altering their effect).

9.1.3 Heuristic modifier \texttt{level}

The \texttt{Domain} heuristic assigns to each atom a \texttt{level}, and it decides first upon atoms of the highest level. The default value for each atom is 0, and both positive and negative integers are valid.

\textbf{Example 9.3.} In this example, level \texttt{10} is assigned to atom \( a \):
```
\texttt{\_heuristic(\(a\),sign,\texttt{1})}.
\texttt{\_heuristic(b,sign,\texttt{1})}.
\texttt{\_heuristic(\(a\),level,\texttt{10})}.
\{a,b\}.
\texttt{\:- a, b}.
```
The first obtained answer set contains \(a\) along with all three heuristic atoms. The solver propagates the heuristic facts, and given that the level of \(a\) is greater than that of \(b\), it decides first on \(a\) (with positive sign) and then \(b\) is propagated to false. If we added the fact \_heuristic(b,level,20), we would first obtain the answer set containing \(b\) instead of \(a\). This would also be the case if we used \_heuristic(a,level,-10) instead of \_heuristic(a,level,10).

Remark 9.3. The Domain heuristic is an extension of the Vsids heuristic, so when there are many unassigned atoms with the highest level, the heuristic decides, among them, on the one with the highest Vsids score.

9.1.4 Dynamic heuristic modifications

Heuristic atoms can be used as any other atom within logic programs, and they only affect the heuristic of the solver when they are true.

Example 9.4. In the next program, the heuristic atoms for \(c\) depend on \(b\):

\[
\begin{align*}
&_\text{heuristic}(a,\text{sign},1). \\
&_\text{heuristic}(b,\text{sign},1). \\
&_\text{heuristic}(a,\text{level},10). \\
\{a,b\}. \\
: & - a, b. \\
\{c\}. \\
&_\text{heuristic}(c,\text{sign},1) : - b. \\
&_\text{heuristic}(c,\text{sign},-1) : - \text{not}\ b.
\end{align*}
\]

The first obtained answer set contains \(a\) and \_heuristic(c,sign,-1) along with the three heuristic atoms given as facts. At first, the solver proceeds as in Example 9.3. Then, after propagating \(b\) to false, the heuristic fact \_heuristic(c,sign,-1) is propagated. So, when deciding upon \(c\), it gets assigned to false. If we added the fact \_heuristic(b,level,20), the obtained first answer set would contain \(b\) and \(c\) instead of \(a\).

9.1.5 Heuristic modifiers true and false

The modifiers true and false allow us to refer at the same time to the level and the sign of an atom. Internally, for the true and false heuristic atoms, the solver defines new level and sign heuristic atoms following these rules:

\[
\begin{align*}
&_\text{heuristic}(X,\text{level},Y) : - &_\text{heuristic}(X,\text{true},Y). \\
&_\text{heuristic}(X,\text{sign},I) : - &_\text{heuristic}(X,\text{true},Y). \\
&_\text{heuristic}(X,\text{level},Y) : - &_\text{heuristic}(X,\text{false},Y). \\
&_\text{heuristic}(X,\text{sign},-I) : - &_\text{heuristic}(X,\text{false},Y).
\end{align*}
\]

Note that in gringo all atoms are shown, unless #show statements appear in the program.
For instance, the program of Example 9.4 can be rewritten as:

```prolog
_heuristic(b,sign,1).
_heuristic(a,true,10).
(a,b).
:- a, b.
{c}.
_heuristic(c,sign,1) :- b.
_heuristic(c,sign,-1) :- not b.
```

In this case, the fact `_heuristic(a,true,10)` stands for the previous facts `_heuristic(a,level,10)` and `_heuristic(a,sign,1)`.

### 9.1.6 Priorities among heuristic modifications

The Domain heuristic allows for representing priorities between different heuristic atoms that refer to the same atom. The priority is optionally represented by a positive integer as a fourth argument. The higher the integer, the higher the priority of the heuristic atom. For example, `_heuristic(c,sign,1,10)` and `_heuristic(c,sign,-1,20)` are valid heuristic atoms. If both are true, then the sign assigned to c is -1 (because priority 20 overrules 10).

**Example 9.5.** Consider the following program:

```prolog
_heuristic(b,sign,1).
_heuristic(a,true,10).
(a,b).
:- a, b.
{c}.
_heuristic(c,sign,1,10).
_heuristic(c,sign,-1,20) :- not b.
```

The first obtained answer set contains a and `_heuristic(c,sign,-1,20)` along with the three heuristic atoms given as facts. First the solver proceeds as in Example 9.3. Then, after setting b to false by propagation, the heuristic atom `_heuristic(c,sign,-1,20)` is propagated. Given that priority 20 is greater than 10, the sign value of atom c is -1. So, when deciding upon c, it is assigned to false. If we added the fact `_heuristic(c,sign,1,30)`, the first answer set would also contain atom c.

**Remark 9.4.** Whenever we only use ternary `_heuristic` atoms, the assigned priority is the absolute value of the modifiers’ values. For example, if both `_heuristic(c,level,-10)` and `_heuristic(c,level,5)` are true, the level of c is -10 because |−10| > |5|.
9.1.7 Heuristic modifiers init and factor

The modifiers init and factor allow for modifying the scores assigned to atoms by the underlying Vsids heuristic. Unlike the level modifier, init and factor allow us to bias the search without establishing a strict ranking among the atoms.

With init, we can add a value to the initial heuristic score of an atom. For example, if _heuristic(a,init,2) is true, then a value of 2 is added to the initial score that the heuristic assigns to atom a. Note that as the search proceeds, the initial score of an atom decays, so init only affects the beginning of the search.

To bias the whole search, we can use the factor modifier that multiplies the heuristic score of an atom by a given value. For example, if _heuristic(a,factor,2) is true, then the heuristic score for atom a is multiplied by 2.

9.1.8 Monitoring domain choices

The Domain heuristic extends clasp and clingo's search statistics produced with command line option --stats. After 'Domain:', it prints how many decisions were made on atoms appearing inside _heuristic atoms. For instance, the statistics obtained in Example 9.4 read as follows.

```
... Models : 1+
          Time : 0.000s (Solving: 0.00s ...)
          CPU Time : 0.000s
          Choices : 2 (Domain: 2)
          Conflicts : 0
          Restarts : 0
... The line about Choices tells us that two decisions were made and that both where made on atoms contained in _heuristic atoms.
```

9.1.9 Heuristics for Blocks World Planning

We now apply the Domain heuristic to Blocks World Planning. For simplicity, we adapt the encoding of Section 5.3.2 for one-shot solving:

```
time(1..lasttime).
location(table).
location(X) :- block(X).
holds(F,0) :- init(F).

% Generate
```

Heuristics may also be applied in the incremental setting of Section 5.3.2, but we introduce them this way for clarity.
9.1 Heuristic Programming

\{ \text{move}(X,Y,T) : \text{block}(X), \text{location}(Y), X \neq Y \} = 1
\text{:- time}(T).

\% Test
\text{:- move}(X,Y,T), \text{holds(on(A,X),T-1)}.
\text{:- move}(X,Y,T), \text{holds(on(B,Y),T-1)}, B \neq X, Y \neq \text{table}.

\% Define
\text{moved}(X,T) :- \text{move}(X,Y,T).
\text{holds(on}(X,Y),T) :- \text{move}(X,Y,T).
\text{holds(on}(X,Z),T) :- \text{holds(on}(X,Z),T-1), \text{not moved}(X,T).

\% Test
\text{:- goal}(F), \text{not holds}(F,\text{lasttime}).

\% Display
\text{#show move/3}.

Constant lasttime bounds the plan length, and we assume it is provided by command line (for example, with option -c lasttime=3). In this encoding, once all the values for predicate move/3 are given, the values of moved/2 and holds/2 are determined and may be propagated by the solver. This suggests that deciding only on move/3 may be a good strategy. We can do that with the Domain heuristic adding the following heuristic rule:

\_\text{heuristic}(\text{move}(B,L,T),\text{level},1) :- \text{block}(B),\text{location}(L),\text{time}(T).

Given that the level of move/3 is higher, the solver decides first on atoms of that predicate, and the values of the other predicates are propagated.

We may prefer to soften the heuristic modification to simply bias the search towards the move/3 predicate, without establishing a strict preference towards it. For that, we can use, for example, the rule

\_\text{heuristic}(\text{move}(B,L,T),\text{init}, 2) :- \text{block}(B),\text{location}(L),\text{time}(T).

or

\_\text{heuristic}(\text{move}(B,L,T),\text{factor},2) :- \text{block}(B),\text{location}(L),\text{time}(T).

or the combination of both. The first rule adds 2 to the initial score of move/3 atoms, while the second multiplies the heuristic score of move/3 by 2.

Whenever we decide on making a move/3 atom true, the other move/3 atoms for the same time/1 are determined to be false, and can be propagated by the solver. So, deciding on true move/3 atoms may be a good idea. For that, we can either use the true modifier to express a strict preference

\_\text{heuristic}(\text{move}(B,L,T),\text{true},1) :- \text{block}(B),\text{location}(L),\text{time}(T).

or just bias the search with init and sign

\_\text{heuristic}(\text{move}(B,L,T),\text{init},2) :- \text{block}(B),\text{location}(L),\text{time}(T).

\_\text{heuristic}(\text{move}(B,L,T),\text{sign},1) :- \text{block}(B),\text{location}(L),\text{time}(T).
or with factor and sign

_heuristic(move(B,L,T),factor,2) :- block(B), location(L), time(T).
_heuristic(move(B,L,T),sign, 1) :- block(B), location(L), time(T).

So far, we have given the same heuristic values to all move/3 atoms, but other options may be interesting. For example, we may prefer to decide first on earlier move/3 atoms, so that the solver performs a forward search. This can be represented with the following rule:

_heuristic(move(B,L,T),true,lasttime-T+1) :-
  block(B), location(L), time(T).

For lasttime=3, the rule ranks move/3 atoms at time 1 at level 3, those at time 2 at level 2, and those at time 3 at level 1, while always assigning a positive sign. In this manner, the solver decides first on setting a move/3 atom at time 1 to true, then one at time 2, and so on.

Another strategy is to perform a backwards search on move/3 from the last to the first time instant, directed by the goals. For this purpose, we can use the following dynamic heuristic rule:

_heuristic(move(B,L,T),true,T) :- holds(on(B,L),T).

As before, the rule can be softened using sign with init or factor. At the start of the search, the goal’s holds/2 atoms are true at the last time step. With this rule, the solver decides on a move/3 atom that makes one of them true. Then, some holds/2 atoms are propagated to the previous time step, and the process is repeated until reaching the first time instant.

We can also choose to promote atoms of the holds/2 predicate. For example, this can be achieved with any of the following rules.

_heuristic(holds(on(B,L),T),level,1) :-
  block(B), location(L), time(T).
_heuristic(holds(on(B,L),T),init,2) :-
  block(B), location(L), time(T).
_heuristic(holds(on(B,L),T),factor,3) :-
  block(B), location(L), time(T).

Another interesting alternative is the following heuristic rule, that proved to be very useful in practice (see [40]):

_heuristic(holds(on(B,L),T-1),true,lasttime-T+1) :-
  holds(on(B,L),T).

The idea is to make the goal’s holds/2 atoms persist backwards, one by one, from the last time step to the first one. Note that a higher level is given to atoms at earlier time instants. First, the solver decides on one of the goal’s holds/2 atoms at the last but one time step, then it decides to make it persist to the previous situation, and so on. Later, it makes persist backwards another holds/2 atom from the goal. With this heuristic, the idea is not to decide on atoms that lead to much propagation (as with move/3 atoms) but rather to make correct decisions, given that usually the values of holds/2 atoms persist by inertia.
9.2 Command Line Structure-oriented Heuristics

The Domain heuristic also allows us to modify the heuristic of the solver from the command line. For this, it is also activated with option --heuristic=Domain, but now the heuristic modifications are specified by option:

```
--dom-mod=<mod>[,<pick>]
```

where <mod> ranges from 0 to 5 and specifies the modifier:

<table>
<thead>
<tr>
<th>&lt;mod&gt;</th>
<th>Modifier</th>
<th>&lt;mod&gt;</th>
<th>Modifier</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>None</td>
<td>1</td>
<td>level</td>
</tr>
<tr>
<td>2</td>
<td>sign (positive)</td>
<td>3</td>
<td>true</td>
</tr>
<tr>
<td>4</td>
<td>sign (negative)</td>
<td>5</td>
<td>false</td>
</tr>
</tbody>
</table>

<pick> specifies bit-wisely the atoms to which the modification is applied:

- 0: Atoms only
- 1: Atoms that belong to strongly connected components
- 2: Atoms that belong to head cycle components
- 4: Atoms that appear in disjunctions
- 8: Atoms that appear in optimization statements
- 16: Atoms that are shown

Whenever <mod> equals 1, 3 or 5, the level of the selected atoms depends on <pick>. For example, with option --dom-mod=2,8, we apply a positive sign to atoms appearing in optimization statements, and with option --dom-mod=1,20, we apply modifier level to both atoms appearing in disjunctions as well as shown atoms. In this case, atoms satisfying both conditions are assigned a higher level than those that are only shown, and these get a higher level than those only appearing in optimization statements.

Compared to programmed heuristics, the command line heuristics do not allow for applying modifiers init or factor and cannot represent dynamic heuristics. On the other hand, they allow us to directly refer to structural components of the program and do not require any additional grounding. When both methods are combined, the atoms modified by the _heuristic predicate are not affected by the command line heuristics.

9.3 Computing Subset Minimal Answer Sets with Heuristics

Apart from boosting solver performance, domain specific heuristics can also be used for computing subset minimal answer sets (cf. [16, 17]). This can be achieved by assigning false with value 1 to the atoms to minimize.

**Example 9.6.** Consider the following program:

```
1 {a(1..3)}. a(2) :- a(3). a(3) :- a(2). {b(1)}. #show a/1.
```
Both the command line option `--dom-mod=5,16` as well as the addition of the heuristic fact `_heuristic(a(1..3),false,1)` guarantee that the first answer set produced is subset minimal with respect to the atoms of predicate `a/1`. Moreover, both allow for enumerating all subset minimal solutions in conjunction with option `--enum-mod=domRec`. In our example, we obtain the answer sets \{a(1)\} and \{a(2), a(3)\}. Note that in this case solutions are projected on shown atoms.

It is worth mentioning that the enumeration mode `domRec` relies on solution recording and is thus prone to an exponential blow-up in space. In practice, however, this often turns out to be superior to enumerating subset minimal model via disjunctive logic programs, which is guaranteed to run in polynomial space.
10 Optimization and Preference Handling

This section shows how quantitative and qualitative preferences can be used for computing optimal answer sets. While Section 10.1 summarizes the standard optimization capacities of clasp, gringo, and clingo dealing with lexicographic optimization of linear objective functions, Section 10.2 provides a tutorial introduction to asprin’s general preference handling framework.

10.1 Multi-objective Optimization with clasp and clingo

This subsection is not yet ready for publishing and will be included in one of the forthcoming editions of this guide.

Some information on multi-objective optimization can be obtained at the following references.

- Optimization [31, 30, 35, 1]
- Video series on clasp’s optimization capacities http://potassco.sourceforge.net/videos.html
- Consult Section 3.1.13 for language constructs expressing multi-criteria optimization.
- Consult Section 6.3.2 and 6.3.3 for relevant clasp options configurating the optimization process.

10.2 Preference Handling with asprin

asprin provides a general framework for optimizing qualitative and quantitative preferences in ASP. It allows for computing optimal answer sets of logic programs with preferences. While asprin comes with a library of predefined preference types (subset, pareto, etc.), it is readily extensible by new customized preference types. For a formal description of asprin, please consult [11].

The following description conforms with asprin 1.1.

10.2.1 Computing optimal answer sets

Similar to common optimization in ASP, where objective functions are added to logic programs via minimize statements or weak constraints, a preference specification is added to a logic program to single out the optimal answer sets with respect to the given preferences. However, as with minimize statements, such a specification is not part of the program but rather a meta statement referring to its answer sets. Hence, preference specifications are directives and thus preceded by #. For clarity, we also refer to the underlying program as the base program (also in view of distinguishing it from the preference program, implementing the preference specification; see below).
To begin with, let us consider a simple example providing a holistic view on preference handling with \textit{asprin}.

\textbf{Example 10.1.} Consider the following base program.

\begin{verbatim}
dom(1..3).
{ m(1..3) } = 1.
a(1) :- m(1). a(1..2) :- m(2). a(3) :- m(3).
b(1..3) :- m(1). b(1) :- m(2). b(2..3) :- m(3).
#show m/1. #show a/1. #show b/1.
\end{verbatim}

We obtain three answer sets, one with \texttt{m(1)}, \texttt{m(2)}, and \texttt{m(3)}, respectively, and refer to them as \texttt{X1}, \texttt{X2}, and \texttt{X3}.

\textbf{Remark 10.1.} Base programs are \texttt{gringo} and \texttt{clingo} programs as specified in Section 3.1, except that atoms names may not start with the underscore symbol ‘\_’ and weak constraint and minimize and maximize statements are not allowed. 

For a first example, we can use \texttt{asprin} to compute the answer sets of the base program that are subset minimal with respect to atoms of predicate \texttt{a/1}. This can be done with the following preference specification (available in \texttt{preference1.lp}):

\begin{verbatim}
1  #preference(p1,subset){ a(X) : dom(X) }.
2  #optimize(p1).
\end{verbatim}

Line 1 contains a preference statement of name \texttt{p1} and type \texttt{subset} that contains a single (non-ground) preference element. Intuitively, the preference statement \texttt{p1} defines a preference of type \texttt{subset} over atoms of predicate \texttt{a/1}. Line 2 contains an optimization directive that instructs \texttt{asprin} to compute answer sets that are optimal with respect to \texttt{p1}.

\textbf{Remark 10.2.} Unlike \texttt{gringo}'s native optimization statements and weak constraints (cf. Section 3.1.13), \texttt{asprin} separates the declaration of preferences from the actual optimization directive.

To compute an answer set of the base program that is optimal with respect to \texttt{p1}, an implementation of the preference type \texttt{subset} must be provided. This is comprised in \texttt{asprin}'s preference library, contained in file \texttt{asprin.lib}. With it, the computation can be performed by the following command:

\texttt{asprin base.lp preference1.lp asprin.lib}

This command produces the following output:

\begin{verbatim}
asprin version 1.0
Answer: 1
m(2) a(2) a(1) b(1)
\end{verbatim}
At first, asprin finds the answer set $X_2$ of the base program. Then, it looks for an answer set that is preferred to $X_2$ and it finds $X_1$. In the last step, asprin looks for an answer set that is preferred to $X_1$, and given that none is found the optimality of $X_1$ is established. In total, two answer sets were enumerated in the computation of an optimal solution.

Alternatively, we can minimize the extension of predicates $a/1$ and $b/1$ with the following preference specification.

```
#preference(p2,subset){ a(X) : dom(X); b(X) : dom(X) }.
#optimize(p2).
```

Now, we obtain that $X_2$ is already an optimal answer set:

```
Answer: 1
m(2) a(2) a(1) b(1)
OPTIMUM FOUND
```

### 10.2.2 Computing multiple optimal answer sets

In analogy to clasp and cling0, asprin allows for computing $n$ optimal answer sets by adding the number $n$ to the command line; as well, 0 is used to compute all optimal answer sets.

**Example 10.2.** For instance, the command

```
asprin base.lp preference2.lp asprin.lib 0
```

results in the output:

```
Answer: 1
m(2) a(2) a(1) b(1)
Answer: 2
m(1) a(1) b(3) b(2) b(1)
OPTIMUM FOUND
Answer: 3
m(3) a(3) b(3) b(2)
OPTIMUM FOUND
```

The computation of the first optimal answer set, $X_1$, is the same as above. Then, asprin searches for an answer set of the base program that is not worse than $X_1$,
finds $X_3$, and proves that it is optimal. In the last step, \texttt{asprin} looks for some answer set that is not worse than $X_1$ and $X_3$, and given that there is none, it terminates.

Adding the following choice rule (via file \texttt{cl.lp}) to the above optimization process

\begin{verbatim}
{c(1)}.
#show c/1.
\end{verbatim}

yields two additional optimal answer sets, both containing \texttt{c(1)}:

\begin{verbatim}
Answer: 1
m(2) a(2) a(1) b(1)
Answer: 2
m(1) a(1) b(3) b(2) b(1)
OPTIMUM FOUND
Answer: 3
m(1) a(1) b(3) b(2) b(1) c(1)
OPTIMUM FOUND *
Answer: 4
m(3) a(3) b(3) b(2)
OPTIMUM FOUND
Answer: 5
m(3) a(3) b(3) b(2) c(1)
OPTIMUM FOUND *
\end{verbatim}

When \texttt{asprin} looks for an answer set that is not worse than $X_1$, it first looks for answer sets that interpret atoms appearing in the preference statements in the same way as $X_1$. In this way, it finds the second optimal model, that contains \texttt{c(1)}, and prints it followed by the line ‘OPTIMUM FOUND *’. Then, it continues searching, finds $X_3$ and the process continues.

Finally, we can project optimal answer sets on the atoms in preference statements by \texttt{asprin}'s option \texttt{--project}. This yields only the three optimal answer sets not containing \texttt{c(1)}.

\section{Input language of \texttt{asprin}}

\texttt{asprin}'s input language extends the one described in Section 3 by constructs for expressing qualitative and quantitative preferences.

A \textit{weighted formula} is of the form

\begin{verbatim}
t::F
\end{verbatim}

where \texttt{t} is a term tuple\textsuperscript{28} and \texttt{F} is a either a Boolean formula or a naming atom. We may drop :: and simply write \texttt{F} whenever \texttt{t} is empty. Boolean formulas are formed

\textsuperscript{28}Term tuples in the current \texttt{asprin} implementation are defined following Figure 2, except that \textit{terms} cannot contain \textit{functions}, and \textit{simple terms} cannot contain \textit{constants}. Moreover, neither intervals (Section 3.1.9) nor pools (Section 3.1.10) are allowed in weighted formulas. These limitations will be
from atoms, possibly preceded by classical negation (‘¬’), using the connectives not (default negation), \& (conjunction) and | (disjunction). Parentheses can be written as usual, and when omitted, negation has precedence over conjunction, and conjunction over disjunction. Naming atoms of form name(s) refer to the preference associated with preference statement s (see below). Examples of weighted formulas are ‘42::a’, ‘a(X)’, ‘C::edge(X,Y,C)’, ‘W, (X,Y) :: not a(\bar{w},X) \& b(Y)’, and ‘X::name(p(X))’.

If F₁, . . . , Fₙ are weighted formulas, then

\{F₁; . . . ; Fₘ\}

is a set of weighted formulas. We may drop the curly braces if m = 1.

A preference element is of the form

\[ F₁ > . . . > Fₘ \mid\mid F : B \]

where each \( Fᵣ \) is a set of weighted formulas, F is a non-weighted Boolean formula and B is a rule body where all literals belong to domain predicates (see Page 30) or built-ins. We may drop ‘>’ if m = 1, and ‘|| F’ and ‘: B’ whenever F and/or B are empty, respectively. Intuitively, r gives the rank of the respective set of weighted formulas. This can be made subject to condition F by using the conditional ‘||’ . Preference elements provide a (possible) structure to a set of weighted formulas by giving a means of conditionalization and a symbolic way of defining pre-orders (in addition to using weights). Examples of preference elements are ‘42::a’, ‘a(X)’, ‘1::name(p);2::name(q)’, ‘{a(X) ; b(X)} > \{c(X) ; d(X)\}’, and ‘a(X) > b(X) \mid\mid c(X) : \text{dom}(X)’.

Remark 10.3. Here and below, the rule body B is intended exclusively to provide instantiations for the variables appearing in the expressions to its left. Accordingly, the literals in B must be built-ins or belong to domain predicates of the accompanying logic program. This ensures that B can be fully evaluated during grounding.

Remark 10.4. Preference elements are required to be safe, i.e., all variables in a preference element must occur in some positive literal in its body or in the body of the preference statement containing it (see below).

A preference statement is of the form

\[#\text{preference}(s, t)\{e₁; . . . ; eₙ\} : B\]

where s is a term giving the preference name, t is a ground term providing the preference type, and each eᵢ is a preference element. The rule body B has the same

removed in future releases. By now, as a workaround, these elements can be represented as follows: in the weighted formula, write a variable (e.g., \( \bar{V} \)) instead of the prohibited element (e.g., \( a(X) \)), and in the body (see below) assign the variable to the element (e.g., \( \bar{V} = a(X) \)).

29This is a pragmatic restriction, enabling asprin to only include the relevant parts of its library.
10.2 Preference Handling with asprin

form and purpose as above. That is, the body $B$ of a preference statement is used to instantiate the variables of $s$, $t$ and each $e_i$. For safety, all variables appearing in $s$ must also appear in a positive literal in $B$.

**Example 10.3.** Given the logic program

$$\text{dom}(1..2).$$

{ $a(X,Y) : \text{dom}(X), \text{dom}(Y)$}.

the preference statement

$$\text{#preference}(p(X), \text{subset}) \{ a(X,Y) : \text{dom}(Y) \} : \text{dom}(X).$$

stands for the following two ground preference statements:

$$\text{#preference}(p(1), \text{subset}) \{ a(1,1) ; a(1,2) \}.$$

$$\text{#preference}(p(2), \text{subset}) \{ a(2,1) ; a(2,2) \}.$$

Preference statements are accompanied by *optimization directives* such as

$$\text{#optimize}(s) : B.$$  

where $B$ is as above, telling asprin to restrict its reasoning mode to the preference relation declared by $s$.

A preference specification is a set of preference statements along with an optimization directive. It is valid, if grounding results in acyclic and closed naming dependencies along with a single optimization directive (see [11] for details). Whenever these conditions do not hold, asprin reports an error and exits. As mentioned, the purpose of such a specification is to define the optimal answer sets of an underlying base logic program.

**Example 10.4.** Consider a preference specification about leisure activities (without base program).

```plaintext
1  #preference(costs,less(weight)){
2    C :: sauna : cost(sauna,C);
3    C :: dive : cost(dive,C)
4 }.
5  #preference(fun,superset){ sauna; dive; hike; not bunji }.
6  #preference(temps,aso){
7    dive > sauna || hot;
8    sauna > dive || not hot
9 }.
10 #preference(all,pareto){name(costs); name(fun); name(temps)}.
11 #optimize(all).
```


10.2 Preference Handling with \textit{asprin}

Intuitively, the relation expressed by the preference statement \texttt{costs} in Line 1 aims at optimizing the sum of weights of its preference elements, viz. \texttt{C::sauna:cost(sauna,C)} and \texttt{C::dive:cost(dive,C)}. The preference type \texttt{less(weight)} is very similar to the one used by native minimization directives (cf. Section 3.1.13). The preference type \texttt{superset} provides a set inclusion based relation and the one refereed to as \texttt{aso} amounts to answer set optimization as put forward in [13]. These three basic preference relations are combined according to the \texttt{pareto} principle in Line 10. And this combined preference relation is declared subject to optimization in Line 12.

\textbf{Remark 10.5.} All four preference types in Example 10.4 are predefined in \textit{asprin}'s preference library and take different syntactic restrictions of preference elements as arguments.

### 10.2.4 Preference relations and preference types

A ground preference statement declares a strict partial order over answer sets.\textsuperscript{30} This order is called a \textit{preference relation}.

\textbf{Example 10.5.} The preference statement of Example 10.1 stands for the following ground preference statement:

\begin{verbatim}
#preference(p1,subset){ a(1); a(2); a(3) }.
\end{verbatim}

It declares the following preference relation:

\[ X \succ_{p1} Y \iff \{ e \in \{a(1), a(2), a(3)\} \mid X \models e \} \subset \{ e \in \{a(1), a(2), a(3)\} \mid Y \models e \} \]

In Example 10.1, we get \( X_1 \succ_{p1} X_2 \) because \( \{a(1)\} \subset \{a(1), a(2), a(3)\} \) and \( X_3 \succ_{p1} X_2 \) given that \( \{a(2), a(3)\} \subset \{a(1), a(2), a(3)\} \); however, we have \( X_1 \not\succ_{p1} X_3 \) since \( \{a(1)\} \not\subset \{a(2), a(3)\} \).

An answer set \( X \) of a base program \( P \) is \textit{optimal} with respect to a preference relation \( \succ \) if there is no other answer set \( Y \) of \( P \) such that \( Y \succ X \). In Example 10.1, \( X_1 \) and \( X_3 \) are optimal, whereas \( X_2 \) is not because \( X_1 \succ_{p1} X_2 \) (and \( X_3 \succ_{p1} X_2 \)). \textit{asprin} computes answer sets of the base program that are optimal with respect to the preference relation defined by the preference statement selected for optimization. Hence, in the example it produces \( X_1 \) and \( X_3 \).

But how does a preference statement declare a preference relation? This is accomplished by the \textit{preference type} that maps a set \( E \) of ground preference elements into a preference relation. For example, the type \texttt{subset} maps \( E \) into

\[ X \succ Y \iff \{ e \in E \mid X \models e \} \subset \{ e \in E \mid Y \models e \} \]

And when applied to the preference elements of \texttt{p1} in Example 10.1, we obtain \( \succ_{p1} \).

\textsuperscript{30}A strict partial order is a transitive and irreflexive relation.
The full generality of preference elements is not always needed. For example, for `subset` we are only interested in preference elements that are Boolean formulas. For this reason, we specify for each preference type its `domain`, i.e., the ground preference elements for which the preference type is well defined. Hence, the domain of `subset` consists of Boolean formulas. Furthermore, a ground preference statement

\[
\text{#preference}(s,t)\{e_1; \ldots; e_n\}.
\]

is `admissible` if every \( e_i \) belongs to the domain of \( t \). If a ground preference statement is not admissible, `asprin` reports an error and exits.

**Example 10.6.** In Example 10.1, the preference statement \( p1 \) is admissible because \( a(1), a(2), \) and \( a(3) \) are Boolean formulas and thus belong to the domain of `subset`. If we added the preference elements \( 1::a(1) \) or `name(p2)`, the statement would not be admissible any more.

### 10.2.5 asprin library

`asprin`'s preference library implements the following basic preference types:

- **subset** and **superset**
- **less(cardinality)** and **more(cardinality)**
- **less(weight)** and **more(weight)**
- **aso** (Answer Set Optimization, [13])
- **poset** (Qualitative Preferences, [17])

We have already given the definition of `subset`. The preference types `superset`, `less(cardinality)`, and `more(cardinality)` share the domain of `subset`. Given a set of ground preference elements \( E \), their semantics is defined as follows:

- **superset** maps \( E \) to the preference relation
  \[
  X > Y \text{ iff } \{e \in E \mid X \models e\} \supset \{e \in E \mid Y \models e\}
  \]

- **less(cardinality)** maps \( E \) to the preference relation
  \[
  X > Y \text{ iff } \{e \in E \mid X \models e\} < \{e \in E \mid Y \models e\}
  \]

- **more(cardinality)** maps \( E \) to the preference relation
  \[
  X > Y \text{ iff } \{e \in E \mid X \models e\} > \{e \in E \mid Y \models e\}
  \]

An example of preference type `superset` is given in Line 5 of Example 10.4.
Example 10.7. We can use the type \texttt{less(cardinality)} to minimize the cardinality of the atoms of predicate \texttt{b/1} in Example 10.1 as follows.

\begin{verbatim}
#preference(p3,less(cardinality)){ b(X) : dom(X) }.
#optimize(p3).
\end{verbatim}

This yields the unique optimal answer set \(X_2\).

Preference types \texttt{less(weight)} and \texttt{more(weight)} are similar to \#\texttt{minimize} and \#\texttt{maximize} statements. However, they do not comprise priorities but apply to general Boolean formulas. Their common domain consists of sets of ground preference elements of the form:

\[ w, t :: F \]

Here, \(w\) is an integer, \(t\) a term tuple, and \(F\) a Boolean formula. Their meaning is defined with respect to a set \(E\) of ground preference elements:

- \texttt{less(weight)} maps \(E\) to the preference relation \(X \succ Y\) iff

\[ \sum_{(w,t) \in E} w < \sum_{(w,t) \in E, X \models F} w \]

- \texttt{more(weight)} maps \(E\) to the preference relation \(X \succ Y\) iff

\[ \sum_{(w,t) \in E} w > \sum_{(w,t) \in E, Y \models F} w \]

For illustrating the similarity to optimization statements, consider the following \#\texttt{minimize} statement from Section 5.2.2.

\begin{verbatim}
#minimize { C,X,Y : cycle(X,Y), cost(X,Y,C) }.
\end{verbatim}

With preference type \texttt{less(weight)}, this can be expressed as follows.

\begin{verbatim}
#preference(myminimize,less(weight))
  { C, (X,Y) :: cycle(X,Y) : cost(X,Y,C) }.
#optimize(myminimize).
\end{verbatim}

The similarity between preference type \texttt{more(weight)} and \#\texttt{maximize} statements is analogous. Recall also Remark 10.2 from above. Another example of preference type \texttt{less(weight)} is given in Lines 1-4 of Example 10.4.

The preference type \texttt{aso} implements answer set optimization [13] and relies upon ground preference elements of the form:

\[ F_1 > \ldots > F_m \mid \mid B. \]

where each \(F_i\) and \(B\) are Boolean formulas. Preference elements of this form are called \texttt{aso} rules. The semantics of \texttt{aso} is based on satisfaction degrees. In a nutshell, the satisfaction degree of an \texttt{aso} rule \(r\) in an answer set \(X\), written \(v_X(r)\),
10.2 Preference Handling with asprin

<table>
<thead>
<tr>
<th>74x768</th>
</tr>
</thead>
<tbody>
<tr>
<td>10.2 Preference Handling with asprin</td>
</tr>
</tbody>
</table>

is 1 if \( X \) does not satisfy the body \( B \), or if \( X \) does not satisfy any \( F_i \), and it
is \( \min\{k \mid X \models F_k, 1 \leq k \leq n\} \) otherwise. Then, a set of \( \text{aso} \) rules \( E \)
mapped to the preference relation defined as follows: \( X \geq Y \) if for all rules \( r \in E \),
\( v_X(r) \leq v_Y(r) \), and \( X > Y \) if \( X \geq Y \) but \( Y \not\geq X \). See [13] for a more detailed
introduction.

**Example 10.8.** The following preference statement of type \( \text{aso} \) expresses a preference for atoms of predicate \( a/1 \) over atoms of predicate \( b/1 \).

\[
\text{#preference(p4,aso)} \{ \ a(X) > b(X) \ : \ \text{dom}(X) \ \}.  
\]

Together with the base program in Example 10.1, this yields the unique optimal answer set \( X_2 \).

Another example of preference type \( \text{aso} \) is given in Lines 6-9 of Example 10.4.

The preference type \( \text{poset} \) implements the approach to qualitative preferences put forward in [52]. Such preferences are modeled as a strict partially ordered set \((S, \succ)\) of literals. The literals in \( S \) represent propositions that are preferably satisfied and the strict partial order \( \succ \) on \( S \) gives their relative importance. The \text{asprin} implementation of \( \text{poset} \) extends the original approach by allowing preferences over Boolean formulas. The domain of \( \text{poset} \) consists of the sets \( E \) of ground preference elements of the form

\[ F, \]

or

\[ F \succ F', \]

where \( F \) and \( F' \) are Boolean formulas. To give a glimpse of the formal underpinnings, consider a set \( E \) of such ground preference statements. The set \( S_E \) consists of all Boolean formulas appearing in \( E \) and the partial order \( \succ_E \) is the transitive closure of the order specified by the preference elements of the second type. Then, \( X \succ Y \) holds if there is some formula \( F \in S_E \) such that \( X \models F \) and \( Y \not\models F \), and for every formula \( F \in S_E \) such that \( Y \models F \) and \( X \models F \), there is a formula \( F' \in S \) such that \( F' \succ F \) and \( X \models F' \) but \( Y \not\models F' \). The interested reader is referred to [52] for full details.

**Example 10.9.** We apply the preference type \( \text{poset} \) to the preference statement of Example 10.8:

\[
\text{#preference(p5, poset)} \{ \ a(X) > b(X) \ : \ \text{dom}(X) \ \}.  
\]

This expresses a preference for the truth of both \( a/1 \) and \( b/1 \), preferring \( a/1 \) over \( b/1 \). With the base program in Example 10.1, we obtain three optimal answer sets \( X_1, X_2 \) and \( X_3 \).

The library of \text{asprin} implements furthermore the following composite preference types, which amount to the ones defined in [77]:

To inspect the output, invoke:

```
asprin base.lp  \ 
asprin.lib
```
• neg
• and
• pareto
• lexico

Preference types and pareto deal with sets of ground naming atoms

name(s)

For neg, these sets must be singleton. And for lexico, each naming atom has an attached tuple \( w \):

\( w :: name(s) \)

Given a naming atom \( name(s) \), let \( >_s \), \( \geq_s \), \( =_s \), \( \leq_s \), \( <_s \) be the strict, non-strict, equal, and inverse preference relations associated with preference statement \( s \). Then, the semantics of each composite preference type is defined as follows:

- neg maps \( E = \{ name(s) \} \) to the preference relation
  \[ X > Y \iff Y <_s X \]

- and maps \( E = \{ name(s_1); \ldots; name(s_n) \} \) to the preference relation
  \[ X > Y \iff \bigwedge_{name(s) \in E} (X >_s Y) \]

- pareto maps \( E = \{ name(s_1); \ldots; name(s_n) \} \) to the preference relation
  \[ X > Y \iff \bigwedge_{name(s) \in E} (X \geq_s Y) \land \bigvee_{name(s) \in E} (X >_s Y) \]

- lexico maps \( E = \{ w_1 :: name(s_1); \ldots; w_n :: name(s_n) \} \) to the preference relation
  \[ X > Y \iff \bigvee_{w :: name(s) \in E} ((X \geq_s Y) \land \bigwedge_{v :: name(s') \in E, v < w} (X =_s Y)) \]

Example 10.10. Consider the following preference specification, where \( p_1 \) and \( p_3 \) are defined as before:

\#preference(p1,subset) { a(X) : dom(X) }.
\#preference(p3,less(cardinality)) { b(X) : dom(X) }.
\#preference(p6,neg) { name(p1); name(p3) }.
\#preference(p7,and) { name(p1); name(p3) }.
\#preference(p8,pareto) { name(p1); name(p3) }.
\#preference(p9,lexico) { 1::name(p1); 2::name(p3) }.

Along with the base program of Example 10.1 and adding the fact \#optimize(p6), asprin produces answer sets \( X_2 \) and \( X_3 \). If instead we optimize over \( p_7 \) or \( p_8 \), we obtain the three answer sets \( X_1 \), \( X_2 \) and \( X_3 \), while optimizing on \( p_9 \) we get only \( X_2 \).
Implementing preference types

In asprin, preference types are implemented by logic programs called preference programs. In a nutshell, a preference type decides if, given a preference statement \( s \), an answer set \( X \) is better than another answer set \( Y \). To represent that decision by a preference program, the three involved elements \( s \), \( X \), and \( Y \) are translated into facts and rules. Let us first look at some simple translations of preference statements.

**Example 10.11.** Recall the preference statement \( p_1 \) of Example 10.1:

\[
\#\text{preference}(p_1, \text{subset})\{ a(X) : \text{dom}(X) \}.
\]

This is translated into:

1. \( \text{preference}(p_1, \text{subset}) \).
2. \( \text{preference}(p_1, (1, (X)), 1, \text{for}(a(X)), ()) :- \text{dom}(X) \).

Line 1 states the name and the type of the preference statement. Line 2 can be read as follows: in the preference statement \( p_1 \), the first element has variables \( t \ X \ u \), and in the first position of the element there is a Boolean formula \( a(X) \) that has an empty list of associated weights. The translation of

\[
\#\text{preference}(p_2, \text{subset})\{ a(X) : \text{dom}(X); b(X) : \text{dom}(X) \}.
\]

replaces \( p_1 \) with \( p_2 \) in Line 1 and 2, and adds:

\( \text{preference}(p_2, (2, (X)), 1, \text{for}(b(X)), ()) :- \text{dom}(X) \).

Number 2 in \( (2, (X)) \) tells us that this is the second preference element.

**Example 10.12.** The preference statement of Example 10.3:

\[
\#\text{preference}(p(X), \text{subset})\{ a(X,Y) : \text{dom}(Y) \} : \text{dom}(X).
\]

is translated into the rules:

\[
\text{preference}(p(X), \text{subset}) :- \text{dom}(X).
\]
\[
\text{preference}(p(X), (1, (X,Y)), 1, \text{for}(a(X,Y)), ()) :- \text{dom}(Y), \text{dom}(X).
\]

Observe how \( \text{dom}(X) \) is appended to both rules.

In general, a weighted formula \( t : F \) occurs in some set

\[
F_i = \{F_1; \ldots; F_m\}
\]

of a preference element \( e_j \) of the form

\[
F_1 > \ldots > F_n \mid F_0 : B_j
\]

that belongs itself to a preference statement \( s \) of the form

\[
\#\text{preference}(s, t)\{e_1; \ldots; e_o\} : B.
\]
Accordingly, the weighted formula is translated into a rule of the form

\[
\text{preference}(s, (j, v), i, \text{for}(t_F), t) :- B_j, B.
\]

where \(i\) and \(j\) are the indexes of \(F_i\) and \(e_j\), respectively, \(v\) is a term tuple containing all variables appearing in the rule, and \(t_F\) is a term representing the Boolean formula \(F\) by using function symbols \textit{not/1, and/2, and or/2} in prefix notation. For example, the formula \((\text{not a}(X) \mid b(X)) \& c(X)\) is translated into \(\text{and}(c(X), \text{or}(_\text{not}(a(X)), b(X)))\). For representing the condition \(F_0\), \(i\) is set to 0. A naming atom \textit{name(s)} is represented analogously, except that \(\text{for}(t_F)\) is replaced by \textit{name(s)}. The translation of a preference statement of the form mentioned above comprises the translation of all weighted formulas appearing in it along with the rule:

\[
\text{preference}(s, t) :- B.
\]

Optimization directives are translated similarly:

\[
\text{#optimize}(s) : B.
\]

becomes:

\[
\text{optimize}(s) :- B.
\]

**Remark 10.6.** All bodies \(B_j\) and \(B\) consist of domain predicates or built-ins. Hence, after grounding, all rules generated in the translation become facts.

**Example 10.13.** The preference specification of Example 10.4 is translated into the following rules:

\[
\begin{align*}
\text{preference}(\text{costs}, (1, (C)), 1, \text{for}(\text{sauna}), (C)) & :- \text{cost}(\text{sauna},C). \\
\text{preference}(\text{costs}, (2, (C)), 1, \text{for}(\text{dive}), (C)) & :- \text{cost}(\text{dive},C). \\
\text{preference}(\text{costs}, \text{less}(\text{weight})). & \\
\text{preference}(\text{fun}, (1, ()), 1, \text{for}(\text{sauna}), ()). & \\
\text{preference}(\text{fun}, (2, ()), 1, \text{for}(\text{dive}), ()). & \\
\text{preference}(\text{fun}, (3, ()), 1, \text{for}(\text{hike}), ()). & \\
\text{preference}(\text{fun}, (4, ()), 1, \text{for}(\_\text{not}(\text{bunji})), ()). & \\
\text{preference}(\text{fun}, \text{superset}). & \\
\text{preference}(\text{temps}, (1, ()), 0, \text{for}(\text{hot}), ()). & \\
\text{preference}(\text{temps}, (1, ()), 1, \text{for}(\text{dive}), ()). & \\
\text{preference}(\text{temps}, (1, ()), 2, \text{for}(\text{sauna}), ()). & \\
\text{preference}(\text{temps}, (2, ()), 0, \text{for}(\_\text{not}(\text{hot})), ()). & \\
\text{preference}(\text{temps}, (2, ()), 1, \text{for}(\text{sauna}), ()). & \\
\text{preference}(\text{temps}, (2, ()), 2, \text{for}(\text{dive}), ()). & \\
\text{preference}(\text{temps}, \text{aso}). & 
\end{align*}
\]
preference(all, (1, ()), 1, name(costs), ()).
preference(all, (2, ()), 1, name(fun), ()).
preference(all, (3, ()), 1, name(temps), ()).
preference(all, pareto).

optimize(all).

A preference program implementing a preference type \( t \) compares two answer sets \( X \) and \( Y \) given a preference statement \( s \) of type \( t \). To allow for this comparison, asprin provides for every term \( \text{for}(F) \) appearing in the translation of \( s \) the fact \( \text{holds}(F) \) whenever \( X \) satisfies the Boolean formula \( F \). Analogously, asprin provides the fact \( \text{holds}'(F) \), if \( Y \) satisfies \( F \).

**Example 10.14.** For the preference statement \( p_1 \) of Example 10.1, translated in Example 10.11, asprin provides the following facts. For deciding whether \( X_1 \succeq p_1 X_2 \) is true, asprin adds the facts \( \text{holds}(a(1)) \), \( \text{holds}'(a(1)) \), \( \text{holds}'(a(2)) \), and \( \text{holds}'(a(3)) \). Similarly, for testing \( X_3 \succeq p_1 X_2 \), asprin provides \( \text{holds}(a(2)) \), \( \text{holds}(a(3)) \), \( \text{holds}'(a(1)) \), \( \text{holds}'(a(2)) \) and \( \text{holds}'(a(3)) \). And for \( X_1 \succeq p_1 X_3 \), atoms \( \text{holds}(a(1)) \), \( \text{holds}'(a(2)) \) and \( \text{holds}'(a(3)) \) are established.

We have seen how asprin provides the translation of the preference statement \( s \) of type \( t \) and the facts of predicates \( \text{holds}/1 \) and \( \text{holds}'/1 \) for every pair of answer sets \( X \) and \( Y \) that may be compared. Then the preference program implementing \( t \) has two parts. In the first part, we define the predicate \( \text{better}/1 \) in such a way that \( \text{better}(s) \) is true iff \( X \succeq s Y \). In the second part, we add a constraint stating that if \( s \) is optimized then \( \text{better}(s) \) must be true.

**Example 10.15.** The preference type \( \text{subset} \) can be implemented as follows (see file \text{subset.lp}).

```
#program preference(subset).
better(S) :- preference(S,subset),
           not holds(A), holds'(A), preference(S,_,_,for(A),_),
           not holds(B) : not holds'(B), preference(S,_,_,for(B),_).
```

Consider that we want to compare two answer sets \( X \) and \( Y \) for which we have the corresponding \( \text{holds}/1 \) and \( \text{holds}'/1 \) facts. Intuitively, \( \text{better}(s) \) is true if \( X \) better than \( Y \) with respect to a preference statement \( s \) of type \( \text{subset} \). More formally, \( \text{better}(s) \) is true if there is one formula \( A \) appearing in \( s \) that is false in \( X \) and true in \( Y \), and every formula \( B \) in \( s \) that is false in \( Y \) is also false in \( X \).

In addition, the following integrity constraint enforces the optimization with respect to a given optimization directive: (included in file \text{basic.lp}):

```
#program preference.
:- not better(P), optimize(P).
```
This cardinality constraint makes sure that \( \text{better}(P) \) holds whenever \( P \) is optimized. Given that this rule is shared by many preference programs, it is included in a preference-type independent program named \textit{preference}.

Instead of using \textit{asprin}'s library, viz. \textit{asprin.lib}, we can now directly use the above preference program as follows:

\[
\text{asprin base.lp preference1.lp subset.lp basic.lp 0}
\]

As before, we obtain \( X_1 \) and \( X_3 \) as optimal answer sets.

\[\square\]

**Remark 10.7.** \textit{asprin} relies on the correctness of preference programs. In other words, if a preference program correctly implements the corresponding preference type, then \textit{asprin} also functions correctly. Otherwise, the behavior of \textit{asprin} is undefined.

\[\square\]

For implementing composite preference types we also define predicate \textit{better/1}, but in this case the implementation relies on predicates that must be defined by other preference types.

**Example 10.16.** The preference type \textit{pareto} is implemented by the following preference program:

\[
\#\text{program preference(pareto)}. \\
\text{better}(P) :- \text{preference}(P,\text{pareto}), \\
\text{better}(Q), \text{preference}(P,\_,\_,\text{name}(Q),\_), \\
\text{bettereq}(R) : \text{preference}(P,\_,\_,\text{name}(R),\_). \\
\]

The program uses predicates \textit{better/1} and \textit{bettereq/1}, representing the relations \( > \) and \( \geq \), respectively. They must be defined by the implementations of the preference types of the named preference statements. To illustrate this, recall the preference statements \textit{p1}, \textit{p3} and \textit{p8} of Example 10.10, that we put together in the file \textit{preference8.lp} along with an optimization directive for \textit{p8}:

\[
\#\text{preference(p1,subset)}\{ \text{a}(X) : \text{dom}(X) \}. \\
\#\text{preference(p3,less(cardinality))}\{ \text{b}(X) : \text{dom}(X) \}. \\
\#\text{preference(p8,pareto)}\{ \text{name}(p1); \text{name}(p3) \}. \\
\#\text{optimize(p8)}. \\
\]

Given that \textit{p8} refers to \textit{p1} and \textit{p3}, which are of type \textit{subset} and \textit{less(cardinality)}, the implementations of these preference types must define \textit{better/1} and \textit{bettereq/1}. For \textit{subset}, we already have the definition of \textit{better/1} from Example 10.15, so we just have to add to the program \textit{preference(subset)} the following rule:

\[
\text{bettereq}(S) :- \text{preference}(S,\text{subset}), \\
\text{not holds}(B) : \text{not holds}'(B), \text{preference}(S,\_,\_,\text{for}(B),\_). \\
\]

For \textit{less(cardinality)}, the following preference program provides the implementation:
10.2 Preference Handling with asprin

```prolog
#program preference(less(cardinality)).
better(P) :- preference(P,less(cardinality)),
   1 #sum { -1,X : holds(X), preference(P,_,_,for(X),_); 
      1,X : holds'(X), preference(P,_,_,for(X),_)).

bettereq(P) :- preference(P,less(cardinality)),
   0 #sum { -1,X : holds(X), preference(P,_,_,for(X),_); 
      1,X : holds'(X), preference(P,_,_,for(X),_)).
```

Putting all this together, we can compute the optimal answer sets of the program with the following command:

```
asprin base.lp preference8.lp basic.lp 
   subset.lp less-cardinality.lp pareto.lp 0
```

As in Example 10.10 (optimizing p8), we obtain $X_1$, $X_2$ and $X_3$.

**Remark 10.8.** The correctness of the implementation of a composite preference type relies on the correctness of the implementations of the preference types to which it relates via naming atoms. For the preference types that the asprin library implements, it provides correct definitions of predicates better/1, bettereq/1, eq/1, worseq/1, and worse/1, representing relations $\succ$, $\preceq$, $=$, $\le$, and $<$, respectively.

**Remark 10.9.** Preference programs of type $t$ start with a directive

```prolog
#program preference(t).
```

and end with another program directive or at the end of a file (cf. Section 3.1.15). For every preference type $t$ appearing in a preference statement $s$, there must be a preference program block starting with:

```prolog
#program preference(t).
```

If there is no such block, asprin prints an error and exits. Additionally, “generic” preference program blocks starting with the directive

```prolog
#program preference.
```

can be used. These are intended to provide rules shared by all preference programs. asprin loads all “generic” preference programs along with the preference programs for the types appearing in the preference statements of the program.

**Remark 10.10.** There are some syntax restrictions to the form of preference programs:

- They may not contain atoms starting with an underscore ‘_’.

- Atoms of predicates `preference/2`, `preference/5`, `holds/1`, `holds’` and `optimize/1` may not appear in the heads of rules.
Remark 10.11. When using \textit{asprin} to compute many optimal answer sets the syntax of preference programs is limited to that of stratified logic programs \cite{3}. This excludes preference programs with disjunctions, choices, or aggregates in rule heads.

Recall from the previous section that we define a set of admissible preference elements for each preference type. The respective notion of admissibility is defined in \textit{asprin} using predicate \texttt{_error/1}. That is, \texttt{_error(X)} is true whenever the preference statement is not admissible. In this case, \textit{asprin} exits and prints the error message bound to \texttt{X}.\footnote{In the current \textit{asprin} implementation, the system only exits after finding a first answer set of the base program.} For coherence, the corresponding rules are also included in the corresponding preference program.

Example 10.17. The preference program (\texttt{subset.lp}) includes the following rules to define the admissibility of \texttt{subset} preference statements:

\begin{verbatim}
_error(M) :-
  M = @cat("subset: 'name("X",")' is not allowed."),
  preference(S,subset),
  preference(S, (N,_,_), name(X), _).

_error(M) :-
  M = @cat("subset: weight "W"," is not allowed."),
  preference(S,subset),
  preference(S, (N,_,_), X, W), W != ().

_error(M) :-
  M = @cat("subset: element "X"," is not allowed."),
  preference(S,subset),
  preference(S, (N,_,_), for(X), _, R != 1).
\end{verbatim}

The first rule checks preference elements containing naming atoms, the second the ones containing weights, and the third one those that contain more than one Boolean formula. Function \texttt{@cat(*args)} is provided by \textit{asprin} and simply concatenates the terms passed as arguments.

Remark 10.12. The predicate \texttt{_error/1} must be defined using domain predicates, built-ins or the special predicates \texttt{preference/2, preference/5} and \texttt{optimize/1} (not using \texttt{holds/1} or \texttt{holds'/1}).
11 Constraint Programming

11.1 ASP modulo CSP solving with clingcon

This section is not yet ready for publishing and will be included in one of the forthcoming editions of this guide.

Information on constraint programming with clingcon can be obtained at the following references.

- http://potassco.sourceforge.net#clingcon
- [45, 69] (clingcon, based upon gringo 3.0.92 and clasp 1.3.10)

11.2 Solving CSPs with aspartame

This section is not yet ready for publishing and will be included in one of the forthcoming editions of this guide.

Information on constraint programming with aspartame can be obtained at the following references.

- http://potassco.sourceforge.net/labs.html
- http://www.cs.uni-potsdam.de/aspartame
- [5] describes aspartame and how it allows for solving finite linear CSPs in ASP

11.3 Constraint Programming with gringo

Grounder gringo features some experimental means for expressing finite linear constraint satisfaction problems within ASP’s modeling language. The linear constraints are compiled into normal rules following the order encoding [79, 5]. Hence, off-the-shelf ASP solvers like clasp can be used to solve such problems.

CSP constraints in gringo are build over constraint terms, which have form

\[ c_1 \$* \$ v_1 \$+ \cdots \$+ \ c_n \$* \$ v_n \]

where \( n > 0 \), and each \( c_i \) (integer factor) and \( v_i \) (name of a constraint variable) are terms. If a factor is one, then the ‘\( c_i \$* \$’ part can be omitted. Similarly, it is possible to just add a factor in which case the ‘\( \$* \$ v_i \$’ part can be omitted.

Linear constraints in gringo are syntactically similar to built-in comparison predicates (cf. Section 3.1.8) but relation symbols have to be preceded with a \( \$ \) symbol

\[ t_0 \$<_1 \cdots \$<_n \ t_n \]

where \( n > 0 \), each \( <_i \) is a comparison predicate, and each \( t_i \) is a constraint term.
In addition, there is the global \textit{disjoint constraint}
\begin{verbatim}
#disjoint { t_1 : c_1 : L_1 ; \ldots ; t_n : c_n : L_n }
\end{verbatim}
where \( n \geq 0 \), \( t_i \) and \( L_i \) are given as in Section 3.1.12, and each \( c_i \) is a constraint term. The idea is that sets of values labeled with the same term(s) must be disjoint.

\textbf{Example 11.1.} For illustration, consider the following encoding of the \textit{n}-queens puzzle:
\begin{verbatim}
1 1 $<=$ $\underline{\text{queen(1..n)}}$ $<=$ n.
3 $\text{queen(X)}$ $\neq$ $\text{queen(Y)}$ :- X=1..n, Y=1..n, X<Y.
4 X $+$ $\text{queen(Y)}$ $\neq$ Y $+$ $\text{queen(X)}$ :- X=1..n, Y=1..n, X<Y.
5 X $+$ $\text{queen(X)}$ $\neq$ Y $+$ $\text{queen(Y)}$ :- X=1..n, Y=1..n, X<Y.
\end{verbatim}
The first line fixes the domain of the integer variables $\text{queen(1)}$ to $\text{queen(n)}$. Line 3 forbids queens on the same columns and the last two lines address queens on the same diagonals.

\textbf{Remark 11.1.} The current implementation of constraints in \textit{gringo} requires that all constraint variables appearing in a program must have finite domains inferable from the grounded program. Hence, rules like in Line 1 of Example 11.1 fixing the domain of a constraint variable have to be added for each constraint variable.
12 Solver Configuration

clasp has more than 80 performance relevant parameters, some of which are shown in Section 6.3. Even if only a discrete subset of all possible parameter settings is considered, this amounts to approximately $10^{59}$ configurations. In such a huge configuration space, it is a tedious and time-consuming task to manually determine a well-performing configuration. Two complementary ways to automatically address this issue for clasp are the automatic configuration selection solver claspfolio [58] and the automatic configuration tool piclasp.

The following description conforms with claspfolio 2.2 and piclasp 1.2, respectively.

12.1 Portfolio-Solving with claspfolio

The targeted use-case of claspfolio is to solve a set of heterogeneous problem instances. In such a case, there is no single well-performing configuration for all instances but a well-performing configuration has to be selected for each individual instance. Therefore, claspfolio should be used either in scenarios involving instances with different characteristics, e.g., due to different encodings, different sizes or changing constraints, or simply to get a first impression of a well-performing configuration on a (homogeneous) benchmark set.

The basic idea of claspfolio consists of using numerical characteristics of instances to select a well-performing configuration from a given set of pre-selected configurations by using machine learning techniques in order to solve a given (ground) logic program at hand. These so-called instance features are computed by claspre.

For illustration, consider to use claspfolio to solve an instance of the ricochet robots problem [27], i.e., examples/ricochet_robots.lp.gz. To invoke claspfolio, we have simply to pass the instance via stdin and tell claspfolio to read from stdin (-I -).

```
$ zcat examples/ricochet_robots.lp.gz | \
  python ./src/claspfolio.py -I -
[[...]
Time : 3.288s
```

Comparing the performance of clasp’s default configuration and the configuration selected by claspfolio shows a 9.1-fold speedup.

```
$ zcat examples/ricochet_robots.lp.gz | clasp
[[...]
Time : 30.080s
```

Another way to use claspfolio is to select a configuration for a given set of instances. In such a setting, claspfolio scores each configuration on each instance and averages over the scores of each configuration. Such a robust and
12.2 Problem-oriented Configuration of clasp with piclasp

well-performing configuration of clasp can than be used without further use of claspfolio which saves some overhead produced by claspfolio (e.g., computing the instance features).

```bash
$ python ./src/claspfolio.py --oracle_dir <INSTANCE_DIR>
% [...] % >>> Algorithm Scores <<< % % 1-th ranked solver: <CONFIGURATION NAME> % Call: <CMD CALL> % Score: <SCORE> % ...
```

clapfolio lists all configuration sorted by its performance score — starting with predicted best-performing configuration. Please note that claspfolio minimizes <SCORE>.

**Remark 12.1.** claspfolio is trained for a runtime cutoff of 600 seconds. It will most likely perform well for smaller runtime cutoffs but performance could get worse with larger runtime cutoffs.

**Remark 12.2.** claspfolio is trained only on decision problems. Therefore, claspfolio does not cover enumeration and optimization related parameters in its selected configurations.

clapfolio provides also an interface to retrain machine learning models on other problem instances (e.g., to get an claspfolio for enumeration applications). To this end, claspfolio supports the Algorithm Selection Library format. To determine a well-performing training configuration of claspfolio, we recommend the use of autofolio [64].

12.2 Problem-oriented Configuration of clasp with piclasp

piclasp allows for identifying a single well-performing parameter configuration in the complete parameter configuration space of clasp. To this end, piclasp optimizes clasp's configuration with the automatic algorithm configuration framework SMAC [59]. In the process of determining a configuration, piclasp has to assess the performance of different clasp configurations on different instances. Therefore, piclasp needs a lot more computational resources than claspfolio but has the advantage of adapting clasp even better to a given application.

piclasp has two required parameters:

```bash
--instances, -I a directory containing a set of grounded instances on which the performance of clasp will be optimized.
```

---

www.aslib.net
Problem-oriented Configuration of clasp with piclasp

--cutoff, -c defines the runtime cutoff of each run of clasp.

We recommend that clasp’s default configuration solves at least 50% of the given instances with this cutoff. The runtime of piclasp (i.e., the configuration budget) will be approx. 200 times this runtime cutoff to determine a well-performing configuration of clasp.

To install all required packages of piclasp, please run ‘bash install.sh’. This locally installs clasp, SMAC, runsolver and claspre.

For illustration, consider to use piclasp to determine a well-performing configuration again for the ricochet robots problem, you have to provide a directory with the grounded instances, e.g., a directory with examples/ricochet_robots.lp.gz

Running piclasp with a budget of 3300 seconds (-b 3300) and a runtime cutoff of 33 seconds per clasp run on this one instance yields the following result.

$ python piclasp.py -b 3300 -c 33 -I <INSTANCE_DIR>
Found 1 instances
[...]
Result of piclasp:
Performance: 0.094000
--backprop --eq=0 --no-gamma --trans-ext=all
--sat-prepro=0 --init-watches=2
--heuristic=Domain --score-other=1
--sign-def=0 --rand-freq=0.05
--strengthen=local,1 --lookahead=no
--otfs=2 --reverse-arcs=3 --dom-mod=5,0
--save-progress=129 --restarts=no
--partial-check=50 --score-res=1
--update-lbd=0 --deletion=no
--loops=common --del-grow=0
--init-moms --contraction=no

Comparing the performance of clasp’s default configuration and the configuration determined by piclasp shows a 295-fold speedup.

$ zcat examples/ricochet_robots.lp.gz | clasp
[...]
Time : 30.080s

$ zcat examples/ricochet_robots.lp.gz | \ clasp <PICLASP CONFIGURATION>
[...]
Time : 0.102s

Interestingly, the configuration determined by piclasp changes nearly all parameters of clasp. However, we do not know which of these changes (resp. which combination) is necessary for the performance improvement.
Remark 12.3. To improve the performance of piclasp, we recommend to run piclasp with at least 10 independent SMAC runs (option \texttt{--repetition,-R}). More SMAC runs or a larger configuration budget (option \texttt{--budget,-B}) should always lead to better results.

Remark 12.4. Algorithm configuration and hence also piclasp works especially well on homogeneous instance sets (e.g., [27]), that is, there is one configuration that performs well on all given instances. On heterogeneous instance sets, piclasp will most likely need a lot more SMAC runs and a larger configuration budget, and it will still find only configurations with small performance improvements, since clasp’s default configuration is already optimized to have a robust performance on a large variety of instances.

Remark 12.5. Using piclasp, the performance of clasp on the given instance set will improve. However ultimately, the performance of clasp should improve on new (unseen) instances. Therefore, we strongly recommend to use another (disjoint) set of instances to assess the performance of the obtained clasp configuration.

Remark 12.6. We recommend that piclasp optimizes the performance of clasp on at least 100 instances (in contrast to our mini example above). On smaller instance sets, the determined configuration may not perform well on yet unseen instances.
13 Future Work

We conclude this guide with a brief outlook on the future development of gringo [46], clasp [43], and clingo [33]. An important goal of future releases will be improving usability by adding functionalities that make some errors and warnings obsolete. In particular, we consider adding support for arbitrary positive loops as well as language constructs that allow for redefining atoms in incremental logic programs. Aggregates in clasp that are involved in non-HCF components are currently compiled into normal rules, adding support for native treatment of such aggregates is an interesting topic. Systems like clingcon [45, 69] support multi-valued variables and constraints that cannot be encoded efficiently in plain ASP in a straightforward manner. This will be addressed in the near future by including constraint processing capacities into grounding as well as solving. The asprin system [11] supports complex preferences that go beyond simple cardinality or subset minimization. We are planning to extend the input language of gringo to be able to express general aggregate-like language constructs, which allow for representing the multitude of constraints available in Constraint Programming [73] as well as complex preferences, as treated by asprin. For the representation of ground programs, we are working on a new intermediate language format to fix some shortcomings of the smodels format and also to represent general language constructs.
A  Complementary Resources

Books [7, 32, 48]

Language Standard [15]

Semantics of gringo’s input language [26, 57]

Potassco publications

Articles http://www.cs.uni-potsdam.de/wv/publications/index.html


Potassco mailing lists

potassco-users http://sourceforge.net/p/potassco/mailman/potassco-users

potassco-announce http://sourceforge.net/p/potassco/mailman/potassco-announce

Potassco videos http://potassco.sourceforge.net/videos.html

Potassco teaching material http://potassco.sourceforge.net/teaching.html

Potassco FAQ http://potassco.sourceforge.net/faq.html

Potassco sourceforge resources

Bug tracker http://sourceforge.net/p/potassco/bugs

Feature requests http://sourceforge.net/p/potassco/bugs

Wiki http://sourceforge.net/p/potassco/wiki

Further resources http://potassco.sourceforge.net/links.html
B Differences to the Language of gringo 3

This section is not yet ready for publishing and will be included in one of the forthcoming editions of this guide.

Information on differences between the languages of gringo 3 and 4 can be obtained here:

- NOTES in gringo/clingo distribution

Removed features

- #hide statements
- #domain statements
- #compute statements
- aggregates
  - multiset semantics
  - #avg
  - #even/#odd
C Differences to the Language of iclingo and oclingo

This section is not yet ready for publishing and will be included in one of the forthcoming editions of this guide.
References


References


References


ings of the Sixth International Conference on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems (CPAIOR’09), volume 5547 of Lecture Notes in Computer Science, pages 71–86. Springer-Verlag, 2009. 64


References


Index

aggregate, 30
  average, #avg, 109
  body, 31
  condition, 28
  count, #count, 31
  even, #even, 109
  head, 32
  maximum, #max, 31
  minimum, #min, 31
  odd, #odd, 109
  sum plus, #sum+, 31
  sum, #sum, 31

arithmetic function, 24
  absolute value, |·|, 24
  addition, +, 24
  bitwise and, &·, 24
  bitwise complement, ¬·, 24
  bitwise or, ?·, 24
  bitwise xor, ^·, 24
  division, /·, 24
  exponentiation, ·*, 24
  modulo, \·, 24
  multiplication, ·*, 24
  subtraction, −·, 24
  unary minus, −·, 24

atom, see also literal, 19
  see aggregate
  Boolean constant, 24
  see comparison predicate
  see pooling

comparison predicate, 25
  greater or equal, ≥·, 25
  greater, >·, 25
  inequality, !·=, 25
  less or equal, ≤·, 25
  less, <·, 25

conditional literal, 28

domain predicate, 30

eexternal function, 39

heuristic predicate, 75
  factor, 79
  false, 77
  init, 79
  level, 76
  sign, 75
  true, 77

literal, see also atom, 19
  conditional, 28
  head, 23

logic program, 19
  disjunctive, 22
  normal, 19

lua, 39

meta-statement, 43
  comment, 43
  const, #const, 44
  external, #external, 45
  see optimization
  program part, #program, 45
  show, #show, 43

negation, see also literal
  classical, 22
  default, 19
  double, 23

optimization, 37
  maximize, #maximize, 37
  minimize, #minimize, 37
  optimization directive, 87
  preference program, 95
    preference specification, 87
    preference element, 87
    preference statement, 87
    weighted formula, 87
  see preference type
    weak constraint, 37

pooling, ;·, 28
Index

preference type, 90
  and, 93
  aso, 91
  less(cardinality), 91
  less(weight), 91
  lexico, 93
  more(cardinality), 91
  more(weight), 91
  neg, 93
  pareto, 93
  poset, 91
  subset, 91
  superset, 91
python, 39

rule
  disjunctive, 22
  fact, 19
  integrity constraint, 19
  normal, 19

safety, 21
  aggregate, 34
  arithmetic function, 25
  comparison predicate, 27
  external, #external, 45
  interval, 27
  maximize, #maximize, 38
  minimize, #minimize, 38
  preference statement, 88
  show, #show, 44
  weak constraint, 38

term, 17
  #inf, 17
  #sup, 17
  anonymous variable, 17
  see arithmetic function
  constant, 17
  see external function
  function, 17
  integer, 17, 24
  interval, 27
  see pooling
  string, 17

variable, 17