On the Input Language of ASP Grounder *Gringo*

Martin Gebser, Roland Kaminski, Max Ostrowski, Torsten Schaub*, and Sven Thiele

Universität Potsdam, Institut für Informatik, August-Bebel-Str. 89, D-14482 Potsdam

Abstract. We report on recent advancements in the development of grounder *Gringo* for logic programs under answer set semantics. Like its relatives, *DLV* and *Lparse*, *Gringo* has in the meantime reached maturity and offers a rich modeling language to program developers. The attractiveness of *Gringo* is fostered by the fact that it significantly extends the input language of *Lparse* while supporting a compatible output format, recognized by many state-of-the-art ASP solvers.

1 Introduction

Answer Set Programming (ASP; [1]) is an attractive paradigm for knowledge representation and reasoning. On the one hand, its popularity is due to the availability of efficient off-the-shelf solvers (cf. [2]). But equally or even more important under the aspect of usability is its rich modeling language, including first-order variables, function symbols, aggregates, etc. In fact, search problems are in ASP usually modeled in a uniform way by means of a data part, called instance, and a general part, called encoding (cf. [3–7]). The computation of answer sets, corresponding to problem solutions, is then typically performed in two phases: first, *grounding* the encoding on the problem instance, and second, solving the resulting propositional program.

In contrast to the multitude of available solvers, the field of ASP grounders is still underrepresented. To the best of our knowledge, there are only three popular grounders, namely, (the grounding component of) *DLV* [8], *Lparse* [9], and *Gringo* [10]. While *DLV* processes the grounding result internally or prints it as text, the numerical output format of *Lparse* and *Gringo* is recognized by many state-of-the-art ASP solvers. In view of this transparency from the solver side and the progress made since the first description of *Gringo* [11], *Gringo* has become a real alternative to *Lparse*. In particular, the attractiveness of *Gringo* is fostered by the fact that it significantly extends the input language of *Lparse*, providing advanced modeling features to ASP programmers. This paper reports on such new features of *Gringo*, potentially useful for ASP applications.

2 Modeling Features

The input language of *Gringo* is designed to be in large parts compatible to the one of *Lparse* [9], so that the majority of *Lparse* programs can still be grounded with *Gringo*. Assuming basic familiarity with *Lparse*, we focus our description on extensions available in *Gringo* and also mainly take *Lparse* as the grounder to contrast with.

* Affiliated with Simon Fraser University, Canada, and Griffith University, Australia.

λ-Restricted Programs [11]. The class of programs processable with *Gringo* is a proper superclass of ω-restricted programs [12] accepted by *Lparse*. The underlying idea is that all relevant ground instances of a rule (that is, ground instances whose bodies can potentially be true wrt an answer set) are implicitly given if, for each variable in the rule, we find some atom in the positive rule body such that its predicate's relevant ground instances are known. As the basic grounding algorithm of *Gringo* works rule-wise, the latter is the case when all rules with the predicate occurring in the head have already been instantiated. In fact, before beginning with instantiation, *Gringo* computes an ordering such that all rules with a predicate in the head are completely processed before the predicate is used to restrict variable domains in other rules where it occurs in the positive body. Notably, *Gringo* imposes no additional restrictions, such as being definite or stratified, on the rules to be ordered. To see this, consider the following example [10]:

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

Here, *Gringo* first looks at the (ground) rules with `zig/1` and `zag/1` in the head, and so it determines that `0` and `1` are all argument values for which the predicates can hold wrt answer sets. This is used to restrict ground instances of `X` and `Y` in the rule with `zigzag/2` in the head, which in turn restricts `X` and `Y` in the rule with `zagzig/2`. Hence, *Gringo* grounds the above program without complaints, while *Lparse* rejects it because of not being ω-restricted. In order to use *Lparse*, we would have to add a domain predicate, saying that `X` and `Y` must be `0` or `1`, to the bodies of the last two rules. Of course, such information would be redundant, and thus λ-restrictedness helps to write more focused programs, concentrating on the relevant information within rules.

Uninterpreted Functions. The input language of *Gringo* allows for using functions in the heads and bodies of rules, and unification is applied for instantiating variables in uninterpreted functions. For instance, this enables *Gringo* to ground the program:

```
parent(joan,mother(jane)).  female(Y) :- parent(X,mother(Y)).
parent(joan,father(john)).    male(Y) :- parent(X,father(Y)).
```

Though *Lparse* also tolerates uninterpreted functions, it internally handles them like interpreted (arithmetic) functions, and so it refuses to instantiate `Y` in the above program. For modeling, the full support of uninterpreted functions by *Gringo* can be beneficial.

Conditions. Conditions are indicated by "`:`" in the input languages of *Lparse* and *Gringo*. Their purpose is to instantiate "local" variables on the left-hand side with values for which (a set of) literals over domain predicates on the right-hand side holds (cf. [9, 10]). For illustration, consider the following program:

```
od(1).              ne(1).
      ev(2). pr(2).
od(3).       pr(3).
and_1 :-     pr(X) : od(X).              % and_1 :- pr(1), pr(3).
and_2 :-     ne(X) : od(X) : not pr(X). % and_2 :- ne(1).
and_3 :- not ev(X) : ev(X) : not pr(X). % and_3.
or(X) : od(X).                           % or(1) | or(3).
```

For comparison, the ground rules qualified by the rules with conditions are provided in comments. First, observe that *Gringo* expands conditions in the bodies of rules into conjunctions of the required length, while disjunction is used for conditions in rule heads. Furthermore, default negation via `not` can be used on the right-hand side and, in rule bodies, also on the left-hand side of a condition. A particular case is illustrated by the rule with head `and_3`, where the left-hand side is the negation of an atom on the right-hand side. In this situation, the set of literals on the right-hand side must be unsatisfied by all of its ground instances, as it happens with the above rule, or the expansion of the condition is immediately unsatisfied. This phenomenon can be exploited for testing whether certain properties do not hold wrt all ground instances of a set of literals. Comparing with conditions in *Lparse* yields that it accepts only the rule with head `and_1`, while neither negative literals nor occurrences in rule heads are supported. To illustrate the usefulness of the latter, let us consider a disjunctive encoding of N-Coloring:

```
#const n=3.
col(X,C) : C = 1..n :- node(X).
% col(X,1) | ... | col(X,n) :- node(X).
:- col(X,C), col(Y,C), edge(X,Y).
```

To keep the encoding general, we make use of a constant `n` for defining the number of available colors. All values from `1` to `n` can successively be assigned to `C` in the condition of the first rule. Hence, we obtain a disjunction ranging over all colors for each node `X`. Without this opportunity, it is more involved to make use of disjunction for arbitrary N. In fact, as the length of the required disjunction is open (illustrated also by the uncommented rule), other ways of encoding it would have to be used instead.

Aggregates. Aggregates (and associated comparison operations), like the ones supported by *DLV* [13] or cardinality and weight constraints of *Lparse* [9], permit a compact representation of (numerical) constraints on sets of literals. The aggregates currently supported by *Gringo* are: `#count`, `#sum`, `#times`, `#avg`, `#min`, `#max`, `#even`, and `#odd`. Each aggregate applies to either a set of literals, enclosed in curly brackets, or a multiset of literals with associated weights, enclosed in square brackets, where `1` is used as a default for omitted weights. The result of applying an aggregate can be compared to a lower bound ($-\infty$ if omitted) and an upper bound (∞ if omitted) in order to obtain a truth value. The only exceptions to this are `#even` and `#odd` whose meanings are fixed independently of bounds. Before we illustrate individual aggregates, we note that [14] provides a general semantics for them. An objective of *Gringo* is to respect this semantics as far as possible, with the modification of applying "choice semantics" [15, 16] instead of minimization to atoms occurring positively in an aggregate being the head of a rule. However, some compromises are needed for compatibility to the output format of *Lparse*, supporting only `#count` and `#sum` (all other aggregates are compiled into them), and only non-negative weights in `#sum` aggregates (negative weights are eliminated by translation [15]). As a consequence, compliance with the "choice version" of the semantics in [14] is only guaranteed if dependencies through `#avg` as well as `#sum` and `#times` aggregates with negative weights are not subject to (positive) recursion, i.e., an atom appearing in such an aggregate in a rule body should not be defined (directly or indirectly) by any atom occurring positively in the rule head. Now illustrating the available aggregates, we begin with the ones familiar from *Lparse*:

```
1 #count {a,   not b, c} 2.   % 1 {a,   not b,   c} 2.
1 #count {a,a, not b, c} 2.   % 1 {a,   not b,   c} 2.

2 #sum [a=1, not b=1, c=2] 3. % 2 [a=1, not b=1, c=2] 3.
2 #sum [a,   not b,   c,c] 3. % 2 [a=1, not b=1, c=2] 3.
```

The above (ground) facts specify `#count` and `#sum` aggregates. In comments, we provide their notations in terms of cardinality and weight constraints, also accepted by *Gringo* for compatibility to *Lparse*. Note that *Gringo* properly deals with the set semantics of `#count` and multisets of `#sum`, while *Lparse* turns `a,a` as in the second fact into `a=2`. However, given that the above facts contain negative literal `not b` (in the head), *Lparse* would not accept them either. Such restrictions do not apply to *Gringo*, capable of handling negative literals in aggregates occurring as rule heads. The next examples demonstrate the use of the further aggregates supported by *Gringo*:

```
2 #times [a=1,     not b=2, c=3] 3.
2 #times [a, not b,not b, c,c,c] 3.
  % 2 #times [a=1, not b=1, c=1] 3.

2 #avg [a=3, not b=1, c=0] 2.
2 #avg [a,a, not b=2].        % 2 #avg [a=1, a=1, not b=2].

2 #min [a=1,     not b=2, c=3] 2.
2 #min [a, not b,not b, c,c,c] 2.
  % 2 #min [a=1, not b=1, c=1] 2.

2 #max [a=1,     not b=2, c=3] 2.
2 #max [a, not b,not b, c,c,c] 2.
  % 2 #max [a=1, not b=1, c=1] 2.

  #even {a,   not b, c}.      #odd  {a,   not b, c}.
  #even {a,a, not b, c}.      #odd  {a,a, not b, c}.
  % #even {a, not b, c}.      % #odd  {a, not b, c}.
```

For aggregates over repeated literals and omitted weights, semantically equivalent counterparts are provided in comments. With multisets, repeated literals appear also repeatedly in the output of *Gringo*, while the effect of such repetitions depends on the aggregate at hand. As regards the `#avg` aggregate, `a=2` contributes one and `a,a` two addends to the numerator and denominator, respectively, in the average calculation. Also note that the meanings of `a=2` and `a,a` are different from one another in `#times`, `#min`, and `#max` aggregates. Finally, as `#even` and `#odd` determine the parity of the number of (true) literals in sets, repeated literals are collapsed into one. Though not demonstrated above, aggregates (and associated comparison operations) can also be used in rule bodies. In addition to comparing aggregate results to bounds, *Gringo* supports assigning the result to a variable, as exemplified in the following program:

```
q(X) :- X = #sum [p(Y) : p(Y) : Y #mod 2 != 0 = Y].   p(1..3).
```

Such assignments of aggregate results are also possible with *DLV* (cf. [17]), but not in *Lparse*. Their main application is to identify deterministic properties of instances that can be calculated from the stratified part of a program [10].

Runtime Options. The default output format of *Gringo* is the same as the one of *Lparse* [9]. Via (experimental) option `--aspils`, the output is printed in one of the normal forms of ASPils, an intermediate format proposed in [18]. As with *Lparse*, option `--text` (or `-t`) makes *Gringo* print ground rules in human-readable text format. In addition, option `--debug` can be provided to investigate internal representations of (non-ground) rules during grounding. Via `--const` (or `-c`), also available in *Lparse*, occurrences of a constant can be replaced with another term, e.g., beneficial with N-Coloring as encoded above. For disjunctive programs, in particular, "head-cycle-free" ones, option `--shift` replaces disjunction in rule heads with default negation in rule bodies, so that solvers for non-disjunctive ASP can be applied. To improve efficiency, if an input program is already ground, it can be signaled to *Gringo* via option `--ground`. This allows *Gringo* to avoid unnecessary yet non-negligible overhead, which is useful, e.g., for running in a mode similar to category SCore of the first ASP system competition [2]. Note that any occurrences of variables are considered as syntax errors if *Gringo* expects an input program to be ground. The binder-splitting technique [11], applied by default, can be switched off via option `--bindersplit`; this is mainly to admit experimental comparisons. Finally, options `--ifixed` and `--ibase` enable *Gringo* to ground incremental programs, written for *iClingo* [19] and containing meta-directives `#base`, `#cumulative`, and `#volatile` [10].

3 Discussion

We have presented relevant features of grounder *Gringo* (version 2.0.3), significantly extending the functionalities of *Lparse*. *Gringo* constitutes an integral part of *Potassco*, the Potsdam Answer Set Solving Collection bundling tools for ASP, for which sources (and binaries) are publicly available at `http://potassco.sourceforge.net`. In addition to its executable, *Gringo* comes as library. As such, it is used inside ASP systems *Clingo*, *iClingo*, and *Clingcon*, all belonging to the Potassco suite. However, due to supporting the output format of *Lparse*, *Gringo* is not limited to work only in integrated tools, but can be used as a front-end for many state-of-the-art ASP solvers. Recent applications modeled and grounded with *Gringo* include [20–24]. For the future, we plan to integrate grounding techniques beyond rule-wise working ones and, accordingly, to relax the required input program properties (currently λ-restrictedness).

Acknowledgments. This work was partially funded by DFG under Grant SCHA 550/8-1 and by the GoFORSYS project under Grant 0313924.

References

1. Baral, C.: Knowledge Representation, Reasoning and Declarative Problem Solving. Cambridge University Press (2003)
2. Gebser, M., Liu, L., Namasivayam, G., Neumann, A., Schaub, T., Truszczyński, M.: The first answer set programming system competition. [25] 3–17
3. Schlipf, J.: The expressive powers of the logic programming semantics. Journal of Computer and System Sciences **51** (1995) 64–86
4. Marek, V., Truszczyński, M.: Stable models and an alternative logic programming paradigm. In: The Logic Programming Paradigm: a 25-Year Perspective. Springer (1999) 375–398

5. Niemelä, I.: Logic programs with stable model semantics as a constraint programming paradigm. Annals of Mathematics and Artificial Intelligence **25**(3-4) (1999) 241–273
6. Gelfond, M., Leone, N.: Logic programming and knowledge representation — the A-Prolog perspective. Artificial Intelligence **138**(1-2) (2002) 3–38
7. Lifschitz, V.: Answer set programming and plan generation. Artificial Intelligence **138**(1-2) (2002) 39–54
8. Leone, N., Pfeifer, G., Faber, W., Eiter, T., Gottlob, G., Perri, S., Scarcello, F.: The DLV system for knowledge representation and reasoning. ACM Transactions on Computational Logic **7**(3) (2006) 499–562
9. Syrjänen, T.: Lparse 1.0 user's manual. http://www.tcs.hut.fi/Software/smodels/lparse.ps.gz
10. Gebser, M., Kaminski, R., Kaufmann, B., Ostrowski, M., Schaub, T., Thiele, S.: A user's guide to `gringo`, `clasp`, `clingo`, and `iclingo`. http://potassco.sourceforge.net
11. Gebser, M., Schaub, T., Thiele, S.: GrinGo: A new grounder for answer set programming. [25] 266–271
12. Syrjänen, T.: Omega-restricted logic programs. In: Proceedings of the Sixth International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'01). Springer (2001) 267–279
13. Faber, W., Pfeifer, G., Leone, N., Dell'Armi, T., Ielpa, G.: Design and implementation of aggregate functions in the DLV system. Theory and Practice of Logic Programming **8**(5-6) (2008) 545–580
14. Ferraris, P.: Answer sets for propositional theories. In: Proceedings of the Eighth International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'05). Springer (2005) 119–131
15. Simons, P., Niemelä, I., Soininen, T.: Extending and implementing the stable model semantics. Artificial Intelligence **138**(1-2) (2002) 181–234
16. Ferraris, P., Lifschitz, V.: Weight constraints as nested expressions. Theory and Practice of Logic Programming **5**(1-2) (2005) 45–74
17. Terracina, G., De Francesco, E., Panetta, C., Leone, N.: Experiencing ASP with real world applications. In: Proceedings of the Fifteenth RCRA Workshop on Experimental Evaluation of Algorithms for Solving Problems with Combinatorial Explosion (RCRA'08). (2008)
18. Gebser, M., Janhunen, T., Ostrowski, M., Schaub, T., Thiele, S.: A versatile intermediate language for answer set programming. In: Proceedings of the Twelfth International Workshop on Nonmonotonic Reasoning (NMR'08). (2008) 150–159
19. Gebser, M., Kaminski, R., Kaufmann, B., Ostrowski, M., Schaub, T., Thiele, S.: Engineering an incremental ASP solver. [26] 190–205
20. Mileo, A., Merico, D., Bisiani, R.: A logic programming approach to home monitoring for risk prevention in assisted living. [26] 145–159
21. Boenn, G., Brain, M., de Vos, M., Fitch, J.: Automatic composition of melodic and harmonic music by answer set programming. [26] 160–174
22. Gebser, M., Schaub, T., Thiele, S., Usadel, B., Veber, P.: Detecting inconsistencies in large biological networks with answer set programming. [26] 130–144
23. Kim, T., Lee, J., Palla, R.: Circumscriptive event calculus as answer set programming. In: Proceedings of the Twenty-first International Joint Conference on Artificial Intelligence (IJCAI'09). AAAI Press (2009) To appear
24. Thielscher, M.: Answer set programming for single-player games in general game playing. In: Proceedings of the Twenty-fifth International Conference on Logic Programming (ICLP'09). Springer (2009) To appear
25. Proceedings of the Ninth International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'07). Springer (2007)
26. Proceedings of the Twenty-fourth International Conference on Logic Programming (ICLP'08). Springer (2008)