GrinGo: A New Grounder for Answer Set Programming

Martin Gebser, Torsten Schaub[⋆], and Sven Thiele

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

Abstract. We describe a new grounder system for logic programs under answer set semantics, called *GrinGo*. Our approach combines and extends techniques from the two primary grounding approaches of *lparse* and *dlv*. A major emphasis lies on an extensible design that allows for an easy incorporation of new language features in an efficient system environment.

1 Motivation, Features, and System Architecture

A major advantage of Answer Set Programming (ASP; [1]) is its rich modeling language. Paired with high-performance solver technology, it has made ASP a popular tool for declarative problem solving. As a consequence, all ASP solvers rely on sophisticated preprocessing techniques for dealing with the rich input language. The primary purpose of preprocessing is to accomplish an effective variable substitution in the input program. This is why these preprocessors are often referred to as *grounders*.

Although there is meanwhile quite a variety of ASP solvers, there are merely two major grounders, namely *lparse* [2] and *dlv*'s grounding component [3]. We enrich this underrepresented area and present a new grounder, called *GrinGo*, that combines and extends techniques from both aforementioned systems. A salient design principle of *GrinGo* is its *extensibility* that aims at facilitating the incorporation of additional language constructs. In more detail, *GrinGo* combines the following features:

– its input language features normal logic program rules, cardinality constraints, and further *lparse* constructs,
– its parser is implemented by appeal to *flex* and *bison++* paving an easy way for language extensions,
– it offers the new class of λ-*restricted programs* (detailed in Section 2) that extends *lparse*'s ω-restricted programs [4],
– its instantiation procedure uses back-jumping and improves on the technique used in *dlv*'s grounder [5] by introducing *binder-splitting* (see Section 3),
– its primary output language currently is textual, as with *dlv*'s grounding component; *lparse* format will be supported soon.

We identify four phases in the grounding process and base the core components of *GrinGo* upon them. The primary *GrinGo* architecture is shown in Figure 1. First, the *parser* checks the syntactical correctness of an input program and creates an internal representation of it. Subsequently, the *checker* verifies that the input program is λ-restricted, so that the existence of a finite equivalent ground instantiation is guaranteed.

[⋆] Affiliated with Simon Fraser University, Canada, and Griffith University, Australia.

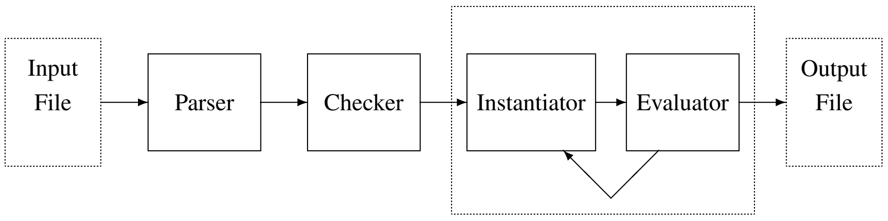


Fig. 1. The *GrinGo* architecture

From this analysis, the checker also schedules the grounding tasks. The *instantiator* computes ground instances of rules as scheduled. Note that the grounding procedure of the instantiator is based on an enhanced version of *dlv*'s back-jumping algorithm. The generated ground rules are then passed to the *evaluator* which identifies newly derived ground instances of predicates. The evaluator also checks for potential program simplifications and finally decides whether a ground rule is output or not.

2 λ-Restricted Programs

For simplicity, we confine ourselves to normal logic programs with function symbols and first-order variables. Let $\mathcal{F}$ and $\mathcal{V}$ be disjoint sets of *function* and *variable* symbols, respectively. As usual, a *term* is defined inductively: Each variable $v \in \mathcal{V}$ is a term, and $f(t_1, \dots, t_k)$ is a term if $f/k \in \mathcal{F}$ and $t_1, \dots, t_k$ are terms. Note that the arity k of f/k can be zero. For a term t, we let $V(t)$ denote the set of all variables occurring in t.

A *rule* r over $\mathcal{F}$ and $\mathcal{V}$ has the form

$$\begin{aligned} p_0(t_{1_0}, \dots, t_{k_0}) \leftarrow\ & p_1(t_{1_1}, \dots, t_{k_1}), \dots, p_m(t_{1_m}, \dots, t_{k_m}), \\ & not\ p_{m+1}(t_{1_{m+1}}, \dots, t_{k_{m+1}}), \dots, not\ p_n(t_{1_n}, \dots, t_{k_n})\ , \end{aligned} \qquad (1)$$

where $p_0/k_0, \dots, p_n/k_n$ are *predicate* symbols, $p_0(t_{1_0}, \dots, t_{k_0}), \dots, p_n(t_{1_n}, \dots, t_{k_n})$ are *atoms*, and t_{j_i} is a term for $0 \leq i \leq n$ and $1 \leq j \leq k_i$. For an atom $p(t_1, \dots, t_k)$, we let $P(p(t_1, \dots, t_k)) = p/k$ be its predicate, and $V(p(t_1, \dots, t_k)) = (V(t_1) \cup \dots \cup V(t_k))$ be the set of its variables. For r as in (1), we define the head as $H(r) = p_0(t_{1_0}, \dots, t_{k_0})$. The sets of atoms, positive body atoms, predicates, and variables, respectively, in r are denoted by $A(r) = \{p_i(t_{1_i}, \dots, t_{k_i}) \mid 0 \leq i \leq n\}$, $B(r) = \{p_i(t_{1_i}, \dots, t_{k_i}) \mid 1 \leq i \leq m\}$, $P(r) = \{P(a) \mid a \in A(r)\}$, and $V(r) = \bigcup_{a \in A(r)} V(a)$. For a rule r and a variable $v \in \mathcal{V}$, we let $B(v, r) = \{P(a) \mid a \in B(r), v \in V(a)\}$ be the set of *binders* for v in r. Note that the set of binders is empty if v does not occur in any positive body atom of r.

A *normal logic program* Π over $\mathcal{F}$ and $\mathcal{V}$ is a finite set of rules over $\mathcal{F}$ and $\mathcal{V}$. We let $P(\Pi) = \bigcup_{r \in \Pi} P(r)$ be the set of predicates in Π. For a predicate $p/k \in P(\Pi)$, we let $R(p/k) = \{r \in \Pi \mid P(H(r)) = p/k\}$ be the set of *defining* rules for p/k in Π. Program Π is *ground* if $V(r) = \emptyset$ for all $r \in \Pi$. The semantics of ground programs is given by their *answer sets* [1]. We denote by $AS(\Pi)$ the set of all answer sets of Π.

We now introduce the notion of λ-*restrictedness* for normal logic programs.

Definition 1. *A normal logic program Π over $\mathcal{F}$ and $\mathcal{V}$ is λ-restricted if there is a level mapping $\lambda : P(\Pi) \rightarrow \mathbb{N}$ such that, for every predicate $p/k \in P(\Pi)$, we have*

$$max\{\ \overbrace{max\{\overbrace{min\{\underbrace{\lambda(p'/k') \mid p'/k' \in B(v,r)}\} \mid v \in V(r)}\} \mid r \in R(p/k)}\ \} < \lambda(p/k)\,.$$

(We added over- and underbraces for the sake of easier readability.) Intuitively, λ-restrictedness means that all variables in rules defining p/k are bound by predicates p'/k' such that $\lambda(p'/k') < \lambda(p/k)$. If this is the case, then the domains of rules in $R(p/k)$, i.e., their feasible ground instances, are completely determined by predicates from lower levels than the one of p/k.

We now provide some properties of λ-restricted programs and compare them with the program classes handled by *lparse* and *dlv*. Recall that *lparse* deals with ω-restricted programs [2], while programs have to be *safe* with *dlv* [3].

Theorem 1. *If a normal logic program Π is ω-restricted, then Π is λ-restricted.*

Note that the converse of Theorem 1 does not hold. To see this, observe that the rules

$$\begin{array}{lll} a(1) \qquad & b(X) \leftarrow a(X), c(X) \qquad & c(X) \leftarrow a(X) \\ & & c(X) \leftarrow b(X) \end{array}$$

constitute a λ-restricted program, but not an ω-restricted one. The cyclic definition of $b/1$ and $c/1$ denies both predicates the status of a domain predicate (cf. [2]). This deprives rule $c(X) \leftarrow b(X)$ from being ω-restricted. Unlike this, the λ-restrictedness of the above program is witnessed by the level mapping $\lambda = \{a \mapsto 0, b \mapsto 1, c \mapsto 2\}$.

On the one hand, the class of λ-restricted programs is more general than that of ω-restricted ones. On the other hand, there are safe programs (that is, all variables occurring in a rule are bound by positive body atoms) that are not λ-restricted. In contrast to safe programs, however, every λ-restricted program has a finite equivalent ground instantiation, even in the presence of functions with non-zero arity.

Theorem 2. *For every λ-restricted normal logic program Π, there is a finite ground program Π' such that $AS(\Pi') = AS(\Pi)$.*

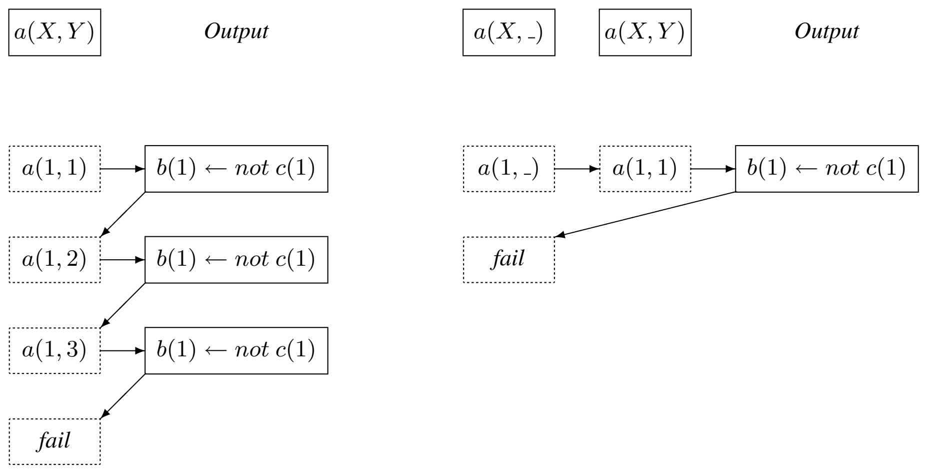
To see the difference between safe and λ-restricted programs, consider the following program, which is safe, but not λ-restricted:

$$a(1) \qquad\qquad a(Y) \leftarrow a(X), Y = X + 1$$

This program has no finite equivalent ground instantiation, which is tolerated by the safeness criterion. To obtain a finite ground instantiation, *dlv* insists on the definition of a maximum integer value `maxint` (in the presence of arithmetic operations, like $+$) for restricting the possible constants to a finite number.

3 Back-Jumping Enhanced by Binder-Splitting

GrinGo's grounding procedure is based on *dlv*'s back-jumping algorithm [5,6]. To avoid the generation of redundant rules, this algorithm distinguishes atoms binding *relevant*



$a(X,Y)$
Output
$a(1,1)$
$b(1) \leftarrow not\ c(1)$
$a(1,2)$
$b(1) \leftarrow not\ c(1)$
$a(1,3)$
$b(1) \leftarrow not\ c(1)$
fail
$a(X,\_)$
$a(X,Y)$
Output
$a(1,\_)$
$a(1,1)$
$b(1) \leftarrow not\ c(1)$
fail

Fig. 2. Back-jumping in (a) *dlv* and (b) *GrinGo*

and *irrelevant* variables. A variable is relevant in a rule, if it occurs in a literal over an unsolved predicate; and a predicate is *solved*, if the truth value of each of its ground instances is known. *dlv*'s back-jumping algorithm avoids revisiting binders of irrelevant variables, whenever different substitutions for these variables result in rule instantiations that only differ in solved literals.

The back-jumping algorithm of *GrinGo* goes a step further and distinguishes between relevant and irrelevant variables within the same binder. The instantiator internally splits such binders into two new binders, the first one binding the relevant variables and the second one binding the irrelevant ones. While the original *dlv* algorithm necessitates that a binder is revisited whenever it contains some relevant variables to find all substitutions for these variables, the *GrinGo* approach allows us to jump over the binder of the irrelevant variables, directly to the binder of the relevant ones. This technique allows us to further reduce the generation of redundant rules.

To illustrate this, consider the rules

$$a(1,1..3) \qquad b(X) \leftarrow a(X,Y), not\ c(X) \qquad c(X) \leftarrow b(X)\ .$$

The predicate $a/2$ is solved before the ground instantiations of the second rule are computed; the atom $a(X,Y)$ acts as binder for the relevant variable X and the irrelevant variable Y. Figure 2 illustrates on the left how *dlv*'s back-jumping algorithm works; it revisits the binder $a(X,Y)$ three times to create all possible substitutions and thus outputs three times the same rule. The scheme on the right in Figure 2 exemplifies *GrinGo*'s binder-splitting. The binder $a(X,Y)$ is replaced with a binder for the relevant variable $a(X,\_)$ plus a second binder $a(X,Y)$ accounting for the bindings of the irrelevant variable Y, depending on the substitution of X. Due to this binder-splitting, it is now possible to jump directly from a solution back to the binder of the relevant variable X, avoiding any further substitutions of Y. As no further substitutions of X are found, the algorithm terminates and does not generate redundant ground rules.

Table 1. *GrinGo*'s back-jumping versus *dlv*'s back-jumping and *lparse*'s backtracking

Sudoku

board	*lparse*	*GrinGo*
1	584.28	5.27
2	190.82	5.48
3	1878.91	5.44
4	1.29	5.40
5	42.13	5.14
6	94.78	5.40
7	10901.35	5.32
8	118.75	5.53
9	165.50	5.42
10	1.58	5.32
SUM	13979.39	53.72

Graph 3-Colorability

graph	*dlv*	*lparse*	*GrinGo*
g40_05_0	0.00	57.30	0.01
g40_05_1	0.00	62.27	0.01
g40_05_2	0.24	39.82	4.01
g40_05_3	0.07	3.49	0.04
g40_05_4	0.00	4.49	0.01
g40_05_5	0.01	21.24	0.03
g40_05_6	0.02	159.69	0.00
g40_05_7	0.62	0.81	57.50
g40_05_8	0.00	1.36	1.12
g40_05_9	4.70	71.38	3.48
SUM	5.66	421.85	66.21

4 Experiments

We tested *GrinGo* [7] (V 0.0.1) together with *dlv*'s grounder (build BEN/Jul 14 2006) and *lparse* (V 1.0.17) on benchmarks illustrating the computational impact of back-jumping and binder-splitting. All tests were run on an Athlon XP 2800+ with 1024 MB RAM; each result shows the average of 3 runs.

For demonstrating the effect of back-jumping, we use logic programs encoding *Sudoku* games. An encoding consists of a set of facts, representing the numbers in Sudoku board coordinates, viz. `number(1..9)`, and a single rule that encodes all constraints on a solution of the given Sudoku instance. All Sudoku instances are taken from newspapers and have a single solution, the corresponding logic programs are available at [7]. The major rule contains 81 variables, which exceeds the maximum number of variables that *dlv* allows in a single rule. We thus only compare our results with *lparse*,[1] the latter relying on systematic backtracking. However, given that *dlv* uses back-jumping as well, we would expect it to perform at least as good as *GrinGo* (if it would not restrict the number of allowed variables below the threshold of 81).

Further, we tested logic programs that encode *Graph 3-Colorability* as grounding problem on a set of random graphs such that a valid coloring corresponds to the ground instantiation of a program. We tested 10 randomly generated graphs, each having 40 nodes and a 5% probability that two nodes are connected by an edge.

Table 1 shows the run times of *lparse*, *dlv*, and *GrinGo* in seconds. Due to its back-jumping technique, *GrinGo*'s performance is almost constant on the Sudoku examples. In addition, *GrinGo* on most instances is faster than *lparse*, the latter showing a great variance in run times. Also on the Graph 3-Colorability examples the grounders using back-jumping techniques turn out to be more robust. The outlier of *GrinGo* on graph 'g40_05_7' however shows that also back-jumping needs a good heuristics for the instantiation order among binders; this is a subject to future improvement.

[1] Note that the grounding procedures of *lparse* and *GrinGo* actually solve the Sudokus, and could in principle be (ab)used for solving other constraint satisfaction problems as well.

Table 2. The effect of *GrinGo*'s binder-splitting

	dlv		*lparse*		*GrinGo*	
n	time	rules	time	rules	time	rules
50	4.13	252500	0.95	252500	0.09	7500
75	24.77	849375	3.14	849375	0.18	16875
100	72.91	1020000	7.80	1020000	0.37	30000
125	166.14	3921875	15.86	3921875	0.68	46875
150	332.40	6772500	26.29	6772500	0.99	67500
175	—	—	42.20	10749375	1.44	91875
200	—	—	65.89	16040000	1.87	120000

For showing the effect of *GrinGo*'s binder-splitting, we use a suite of examples that have a solved predicate with a large domain (viz. `b/2`) and rules in which this predicate is used as the binder of both relevant and irrelevant variables:

```
b(1..n, 1..n).
p(X,Z) :- b(X,Y), b(Y,Z), not q(X,Z).
q(X,Z) :- b(X,Y), b(Y,Z), not p(X,Z).
```

The programs in this suite mainly aim at comparing *dlv* and *GrinGo*, both using backjumping but differing in binder-splitting; *lparse* is included as a reference. The results for parameter `n` varying from 50 to 200 are provided in Table 2. It shows the run time in seconds and the number of generated rules for *dlv*, *lparse*, and *GrinGo*. In fact, all three systems output the same set of rules, differing only in the number of duplicates. Interestingly, both *dlv* and *lparse* even produce the same collection of $n^2 + 2n^3$ rules (ignoring `compute` statements in *lparse*'s output). A hyphen "—" indicates a (reproducible) system failure. The results clearly show that *dlv* and *lparse* generate many (duplicate) rules, avoided by *GrinGo*, and therefore perform poorly on these (artificial) examples. This gives an indication on the computational prospect of binder-splitting.

A more general evaluation of all three grounder systems is an ongoing yet difficult effort, given the small common fragment of the input languages.

References

1. Baral, C.: Knowledge Representation, Reasoning and Declarative Problem Solving. Cambridge University Press (2003)
2. Syrjänen, T.: Lparse 1.0 user's manual. (http://www.tcs.hut.fi/Software/smodels/lparse.ps.gz)
3. Leone, N., Pfeifer, G., Faber, W., Eiter, T., Gottlob, G., Perri, S., Scarcello, F.: The DLV system for knowledge representation and reasoning. ACM TOCL **7**(3) (2006) 499–562
4. Syrjänen, T.: Omega-restricted logic programs. Proceedings of LPNMR'01. Springer (2001) 267–279
5. Leone, N., Perri, S., Scarcello, F.: Backjumping techniques for rules instantiation in the DLV system. Proceedings of NMR'04. (2004) 258–266
6. Perri, S., Scarcello, F.: Advanced backjumping techniques for rule instantiations. Proceedings of the Joint Conference on Declarative Programming. (2003) 238–251
7. (`http://www.cs.uni-potsdam.de/~sthiele/gringo`)