

## **Advances in gringo Series 3**

### Author

Gebser, Martin, Kaminski, Roland, K"onig, Arne, Schaub, Torsten

### Published

2011

### Conference Title

Logic Programming and Nonmonotonic Reasoning

### DOI

[10.1007/978-3-642-20895-9\\_39](https://doi.org/10.1007/978-3-642-20895-9_39)

### Rights statement

© 2011 Springer Berlin/Heidelberg. This is the author-manuscript version of this paper. Reproduced in accordance with the copyright policy of the publisher. The original publication is available at [www.springerlink.com](http://www.springerlink.com)

### Downloaded from

<http://hdl.handle.net/10072/42746>

### Griffith Research Online

<https://research-repository.griffith.edu.au>

# Advances in *gringo* series 3

Martin Gebser, Roland Kaminski, Arne König, and Torsten Schaub\*

Institut für Informatik, Universität Potsdam

**Abstract.** We describe the major new features emerging from a significant redesign of the grounder *gringo*, building upon a grounding algorithm based on semi-naive database evaluation. Unlike previous versions, rules only need to be *safe* rather than domain-restricted.

## 1 Introduction

A distinguishing feature of Answer Set Programming (ASP; [1]) is its highly declarative modeling language along with its domain-independent grounding systems, like *lparse* [2], *dlv* [3], and *gringo* [4]. This paper is dedicated to the features of the new major release of *gringo*, starting with version 3. Most notably, this series only stipulates rules to be *safe* (cf. [5]) rather than  $\lambda$ -restricted [6], as in previous versions of *gringo*. Hence, programs are no longer subject to any restriction guaranteeing a finite grounding. Rather, this responsibility is left with the user in order to provide her with the greatest flexibility. This general setting is supported by a grounding algorithm based on semi-naive database evaluation (cf. [5]), closely related to that of *dlv*. In what follows, we elaborate upon the new features emerging from this significant redesign. For a thorough introduction of *gringo*'s language, please consult the manual available at [7].

## 2 Advances in *gringo* series 3

The most significant change from *gringo* 2 to 3 is that the basic grounding procedure of *gringo* 3 no longer instantiates the rules of a logic program strictly along a predefined order. This enables more convenient predicate definitions in terms of (positive) recursion. E.g., consider a  $\lambda$ -restricted “connected graph design” program:

```
node(1..5).
{ edge(1,X) } :- node(X).
{ edge(X,Y) } :- reached(X), node(Y).
reached(Y)   :- edge(X,Y), node(X;Y).
              :- node(X),   not reached(X).
```

In *gringo* 3, `reached/1` can be defined more conveniently as follows:

```
reached(Y)   :- edge(X,Y).
```

---

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

In fact, additional domain information via `node/1`, needed in rule-wise grounding to “break” the cyclic definition of `edge/2` and `reached/1`, is not anymore required in view of semi-naive evaluation.

Since *gringo* features built-in arithmetics and uninterpreted functions, it deals with potentially infinite Herbrand universes, and the safeness condition does not guarantee termination. For instance, *gringo 3* does not terminate on the following program:

```
succ(X,X+1) :- succ(X-1,X). succ(X,X+1) :- zero(X). zero(0).
```

In fact, the program has an infinite answer set, and it is not  $\lambda$ -restricted in view of the first (recursive) rule. Although it may appear disturbing that the termination of *gringo 3* is not always guaranteed, we deliberately chose to not reintroduce any syntactic finiteness check, and rather leave it to the user to include appropriate stop conditions limiting the “relevant” ground instances of rules. E.g., one may replace the first rule by:

```
succ(X,X+1) :- succ(X-1,X), X < 42.
```

Since *gringo 3* evaluates built-ins while instantiating a rule, it stops grounding at `succ(41,42)`, so that only finitely many relevant ground rules are produced.

The design decision to not enforce (syntactic) conditions guaranteeing termination gives users the freedom to write “clever” encodings, where syntactic checks are bound to fail. To see this, consider the following encoding of a universal Turing Machine:

```
tm(S, L, A, R)           :- init(S),           tape(L, A, R).
tm(SN, L, AL, r(AN, R)) :- tm(S, l(L, AL), A, R), d(S, A, AN, SN, l).
tm(SN, n, 0, r(AN, R))  :- tm(S, n, A, R),     d(S, A, AN, SN, l).
tm(SN, l(L, AN), AR, R) :- tm(S, L, A, r(AR, R)), d(S, A, AN, SN, r).
tm(SN, l(L, AN), 0, n)  :- tm(S, L, A, n),     d(S, A, AN, SN, r).
```

The idea is to represent configurations of the universal Turing Machine by instances of `tm(State, LTape, Symbol, RTape)`, where `State` is a state of the machine that is run, `Symbol` is the tape contents at the current read/write-head position, and `LTape` and `RTape` are (usually) functions representing the tape contents to the left and right, respectively, of the current position. Starting from an initial state and tape, successor configurations are calculated relative to a transition table given by instances of `d(State, Symbol, NSymbol, NState, Direction)`. For instance, if the direction is `l` for “left,” the contents of `NSymbol` is appended to the tape contents to the right, and the next symbol to the left is taken as the contents at the new position, while also removing it from the left-hand side’s tape contents. Hereby, we use `n` to indicate infinitely many blanks `0` to the left or right of the current position, and dedicated rules take care of “generating” a blank on demand when a tape position is visited first. A machine to run, e.g., a 3-state Busy Beaver machine, can then be given by facts like:

```
d(a,0,1,b,r). d(b,0,1,a,l). d(c,0,1,b,l). init(a).
d(a,1,1,c,l). d(b,1,1,b,r). d(c,1,1,h,r). tape(n,0,n).
```

If we run *gringo 3* on the universal Turing Machine encoding along with the facts specifying the 3-state Busy Beaver machine, it generates all traversed configurations, where the final one is as follows:

```
tm(h,l(l(l(l(n,1),1),1),1),1,r(1,n))
```

The fact that *gringo 3* terminates tells us that the 3-state Busy Beaver machine halts after writing six times the symbol 1 to the tape. However, given that *gringo 3* can be used to simulate any machine and the halting problem, in general, is undecidable, it is also undecidable whether semi-naive evaluation yields a finite grounding, i.e., whether *gringo 3* eventually terminates.

The language of *gringo 2* [4] already included a number of **aggregates**, most of which are still supported by *gringo 3*: `#count`, `#sum`, `#min`, `#max`, `#avg`, `#even`, and `#odd`.<sup>1</sup> The support also includes backward compatibility to the traditional notation of *cardinality* and *weight constraints* [2], `1{...}u` and `1[...]u`, respectively, rather than `1#count{...}u` and `1#sum[...]u`. As with *gringo 2*, the condition connective ‘:’ allows for qualifying local variables within an aggregate (or variable-sized conjunctions/disjunctions). Hence, *gringo 2* and *3* both accept the next program:

```
d(1;2;3). { p(X) : d(X) }.
all :- S = #sum[ d(X) : d(X) = X ], S #sum[ p(X) : d(X) = X ].
```

Note that all local variables (named *X*) within aggregates are bound via a domain predicate [2], `d/1`, on the right-hand side of ‘:’. While such binding via domain predicates (or built-ins) had been mandatory with *gringo 2*, it can often be omitted with *gringo 3*. E.g., the last rule above can also be written shorter as follows:

```
all :- S = #sum[ d(X) = X ], S #sum[ p(X) = X ].
```

After investigating the rules with atoms of the predicates `d/1` and `p/1` in the head and collecting their ground instances, *gringo 3* notices that no further rule can derive any instances, so that both domains are limited to 1, 2, and 3. Hence, explicit domain information is not needed to identify all eligible values for the local variables *X* in the remaining rule. In fact, since `d/1` is a domain predicate, *gringo 3* (deterministically) calculates `S = 6`, which is then taken as the lower bound in `6#sum[p(1), p(2), p(3)]`. However, note that a similar omission of ‘: `d(X)`’ is not admissible in the head of the second rule above, since it would violate the safeness requirement. Finally, *gringo 3* does currently not support implicit domains of local variables if an aggregate is involved in (positive) recursion; e.g., the following modified rule is safe, but not yet supported:

```
p(S) :- S = #sum[ d(X) = X ], S #sum[ p(X) = X ].
```

Implicit domains in recursive aggregates are subject to future work (cf. Section 3).

**Optimization** statements, which can be specified via the directives `#minimize` and `#maximize`, are syntactically very similar to aggregates, and *gringo 3* fully supports implicit domains for local variables in them. (Since optimization statements are not part of logic program rules, they cannot be “applied” to derive any atom.) With *lparse* and *gringo 2*, it is possible to provide multiple optimization statements with implicit priorities depending on their order in the input: by convention [2], the last statement is more significant than the second last one, which in turn is more significant than the one before, etc. This convention is also adopted by *gringo 3*, which must be taken into account when writing a sequence of optimization statements like the following one:

```
#minimize[ p(X) = X ].
#maximize[ p(X) = X ].
```

<sup>1</sup> The `#times` aggregate is currently not supported by *gringo 3*; while it requires an involved compilation to “Smodels Internal Format” [2], we are not aware of any application using it.

According to the order, the `#maximize` objective takes precedence over `#minimize`. Since such implicit prioritization necessitates a lot of care to be used properly and also undermines the idea of declarative programming, *gringo* 3 supports explicit priorities via *precedence levels*, provided via the connective '@'. This can be used to override default prioritization, e.g., as follows:

```
#minimize[ p(X) = X @ X ].
#maximize[ p(X) = X @ -X ].
```

If we assume the domain of `p/1` to contain the values 1, 2, and 3, the corresponding ground optimization statements include the following weighted literals (sorted by their precedence levels):

```
#minimize[ p(3) = 3 @ 3, p(2) = 2 @ 2, p(1) = 1 @ 1 ].
#maximize[ p(1) = 1 @ -1, p(2) = 2 @ -2, p(3) = 3 @ -3 ].
```

These optimization statements involve six precedence levels, and a greater level is more significant than a smaller one. Accordingly, our main objective is `p(3)` to be false, followed by `p(2)`, and then `p(1)`. Furthermore, the three negative levels (which are superseded by the positive ones that are greater) express that we would also like `p(1)` to be true, then `p(2)`, and finally `p(3)`. Observe that the optimization priorities are fully determined by precedence levels (and weights), so that there are no implicit priorities based on ordering anymore. We note that prioritization of optimization objectives via precedence levels and weights is also supported by *dlv*, which offers weak constraints [3]. In fact, extending *gringo* by weak constraints is subject to future work.

For selectively **displaying** atoms, *lparse* and *gringo* 2 support `#hide` and `#show` statements to, at the predicate level, decide which atoms in an answer set ought to be presented or suppressed, respectively. Albeit such output restriction mainly serves user convenience, there are also profound application scenarios, such as the enumeration of projected answer sets [8] offered by *clasp* (option `--project`). In fact, the following methodology had been used to, at the predicate level, project Hamiltonian cycles in a clumpy graph down to edges in an underlying “master graph”:

```
% Derive "mc" from "hc"
mc(C1,C2) :- hc(C1,V1,C2,V2), C1 != C2.
% Output PROJECTION to "mc"
#hide.
#show mc(C1,C2).
```

To support *output projection* not only at the predicate but also *at the atom level*, *gringo* 3 allows for conditions, connective ':' followed by domain predicates and/or built-ins, within `#hide` and `#show` statements. Given this, defining a predicate `mc/2` can be omitted and output projection be accomplished more conveniently as follows:

```
% Output PROJECTION
#hide.
#show hc(C1,V1,C2,V2) : C1 != C2.
```

As with precedence levels for optimization, the possibility to distinguish outputs qualified via `#hide` and `#show` at the atom level, rather than at the level of predicates, contributes to declarativeness, as it abolishes the need to define auxiliary predicates within a logic program only for the sake of projecting the displayed output to them.

A number of **built-in** (arithmetic) comparison **predicates**, viz., ‘==’, ‘!=’, ‘<=’, ‘>=’, ‘<’, ‘>’, and (variable) assignments, via ‘=’, were already included in the input language of *gringo* 2. In *gringo* 3, respective comparison predicates are generalized to *term comparisons*, that is, they do not anymore raise an error like “comparing different types,” as encountered with (some versions of) *gringo* 2 when writing, e.g.,  $2 < f(a)$  or alike. Furthermore, while the left-hand side of ‘=’ must be a variable, the *generalized assignment operator* ‘:=’ offered by *gringo* 3 admits composite terms (including variables) on its left-hand side. Hence, it is possible to simultaneously assign multiple variables in a rule like the following one:

$$p(X, Y, Z) :- (X, f(Y, a, Z)) := (a, f(b, a, c)).$$

As with ‘=’, we still require a right-hand side of ‘:=’ to be instantiable before ‘:=’ is evaluated. E.g., a rule like the following one is currently not supported by *gringo*:

$$p(X) :- (X, a) := (a, X).$$

Sometimes, the built-ins offered by a grounder may be too spartan to accomplish sophisticated calculations, and encoding them may likewise be involved and possibly too space-consuming. To nonetheless allow users to accomplish application-specific calculations during grounding, *gringo* 3 comes along with an **embedded scripting language**, viz., *lua* [9]. For instance, the greatest common divisor of numbers given by instances of a predicate  $p/1$  can be calculated via *lua* and then be “saved” in the third argument of a predicate  $q/3$ , as done in the following program:

```
#begin_lua
function gcd(a,b)
  if a == 0 then return b else return gcd(b % a,a) end
end
#end_lua

q(X,Y,@gcd(X,Y)) :- p(X;Y), X < Y. p(2*3*5;2*3*7;2*5*7).
```

When passing this program to *gringo* 3, it for one calculates the numbers being arguments of predicate  $p/1$ , 30, 42, and 70, while the implementation of the `gcd` function in *lua* is used to derive the following facts over predicate  $q/3$ :

$$q(30, 42, 6). \quad q(30, 70, 10). \quad q(42, 70, 14).$$

Beyond sophisticated arithmetics, *lua* also allows for environment interaction. E.g., it provides interfaces to read off values from a database. In the following example, we use `sqlite3`, embedded into the precompiled *gringo* 3 binaries available at [7]:

```
#begin_lua
local env = luasql.sqlite3()
local conn = env:connect("db.sqlite3")
function query()
  local cur = conn:execute("SELECT * FROM test")
  local res = {}
  while true do
    local row = {}
    row = cur:fetch(row, "n")
    if row == nil then break end
  end
end
```

```

        res[#res + 1] = Val.new(Val.FUNC, row)
    end
    cur:close()
    return res
end
#end_lua.

p(X,Y) :- (X,Y) := @query().

```

Here, a *lua* function `query` is used to read data from a table called `test`. Although we do here not delve into the details of *lua*, there is one line that deserves attention:

```
res[#res + 1] = Val.new(Val.FUNC, row)
```

If `test` contains the tuples  $\langle 1,a \rangle$ ,  $\langle 2,b \rangle$ , and  $\langle 3,c \rangle$ , they are successively inserted into the array `res`. The collected tuples are then taken to construct the following facts:

```
p("1", "a"). p("2", "b"). p("3", "c").
```

We note that the generation of terms via *lua* is similar to “value invention” [10], allowing for custom built-in predicates evaluated during grounding.

### 3 Discussion

The redesign of *gringo* is an important step in consolidating the distinct grounding approaches originated by *lparse* and *dlv*. A common ASP language unifying the constructs of both approaches is already envisaged as a joint effort of the teams at Calabria and Potsdam. Although *dlv* and *gringo* now share many commonalities, like safety, semi-naive database evaluation, function symbols, and Turing completeness, they still differ in aspects like finiteness criteria, indexing, connectivity, incrementality, recursive aggregates, backtracking and -jumping. Hence, it is interesting future work to further investigate the different designs and consolidate them wherever possible.

*Acknowledgments.* This work was partly funded by DFG grant SCHA 550/8-2.

### 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. Gebser, M., Kaminski, R., Ostrowski, M., Schaub, T., Thiele, S.: On the input language of ASP grounder *gringo*. In *LPNMR’09*. Springer (2009) 502–508
5. Abiteboul, S., Hull, R., Vianu, V.: Foundations of Databases. Addison-Wesley (1995)
6. Gebser, M., Schaub, T., Thiele, S.: Gringo: A new grounder for answer set programming. In *LPNMR’07*. Springer (2007) 266–271
7. <http://potassco.sourceforge.net>
8. Gebser, M., Kaufmann, B., Schaub, T.: Solution enumeration for projected Boolean search problems. In *CPAIOR’09*. Springer (2009) 71–86
9. Ierusalimschy, R.: Programming in Lua. <http://www.lua.org> (2006)
10. Calimeri, F., Cozza, S., Ianni, G.: External sources of knowledge and value invention in logic programming. *AMAI* **50**(3-4) (2007) 333–361