Advances in *gringo* series 3

Martin Gebser   Roland Kaminski   Arne König
Torsten Schaub

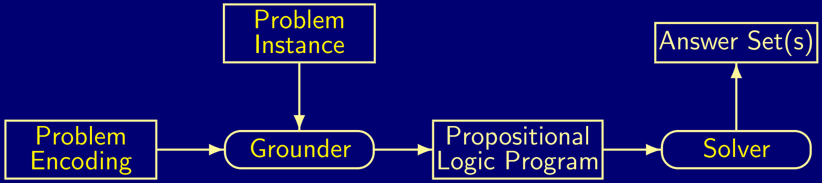
University of Potsdam

Outline

1. Introduction
2. (New) Features
3. Summary

Outline

Answer Set Programming (ASP) =
Knowledge Representation + Instantiation + Search

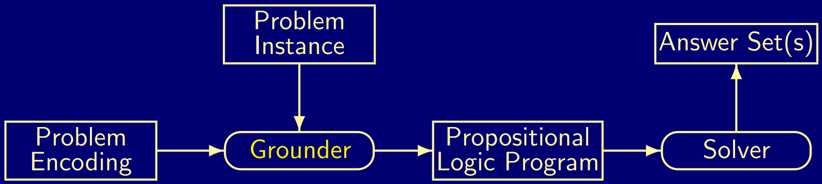


Grounders

lparse	stratified domain predicates
gringo 2.x.x	non-recursive domain predicates
dlv	safe programs w/o domain predicates
gringo 3.x.x	safe programs + rich language of *gringo* 2.x.x **!**
	☞ semi-naive bottom-up evaluation

Motivation

Answer Set Programming (ASP) =
Knowledge Representation + Instantiation + Search

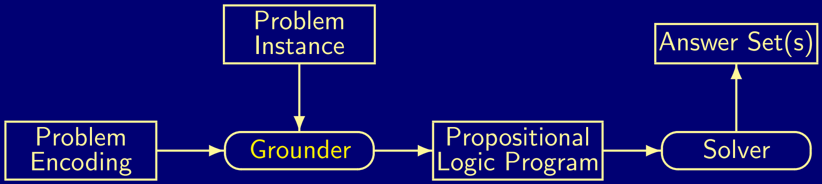


Grounders

lparse	stratified domain predicates
gringo 2.x.x	non-recursive domain predicates
dlv	safe programs w/o domain predicates
gringo 3.x.x	safe programs + rich language of *gringo* 2.x.x **!**
	☞ semi-naive bottom-up evaluation

Motivation

Answer Set Programming (ASP) =
Knowledge Representation + Instantiation + Search



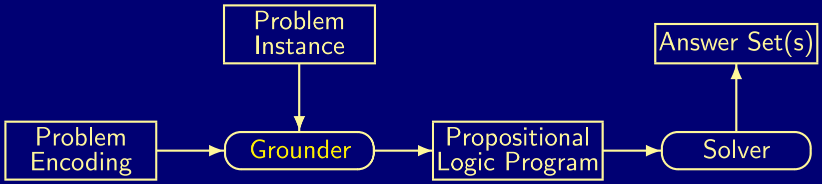
Grounders

lparse	stratified domain predicates
gringo 2.x.x	non-recursive domain predicates
dlv	safe programs w/o domain predicates
gringo 3.x.x	safe programs + rich language of *gringo* 2.x.x **!**

☞ semi-naive bottom-up evaluation

Motivation

Answer Set Programming (ASP) = Knowledge Representation + Instantiation + Search



Grounders

lparse	stratified domain predicates
gringo 2.x.x	non-recursive domain predicates
dlv	safe programs w/o domain predicates
gringo 3.x.x	safe programs + rich language of *gringo* 2.x.x **!**

☞ semi-naive bottom-up evaluation

Outline

(Non-Recursive) Domain Predicates

Connected Graph Design

```
 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).
```

Finite Grounding ?

- Sources of infinity
  1. (built-in) arithmetics
  2. (nested) uninterpreted functions

  ☞ allow for representing the Halting problem (see paper)
- Finiteness guarantees up to the user

  ☞ provide built-in or domain predicates (if needed)

Safe Programs

Connected Graph Design

```
 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).
```

Finite Grounding ?

- Sources of infinity
  1. (built-in) arithmetics
  2. (nested) uninterpreted functions

  ☞ allow for representing the Halting problem (see paper)
- Finiteness guarantees up to the user

  ☞ provide built-in or domain predicates (if needed)

Safe Programs

Connected Graph Design

```
 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).
```

Finite Grounding ?

- Sources of infinity
  1. (built-in) arithmetics
  2. (nested) uninterpreted functions

  ☞ allow for representing the Halting problem (see paper)
- Finiteness guarantees up to the user

  ☞ provide built-in or domain predicates (if needed)

Safe Programs

Connected Graph Design

```
 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).
```

Finite Grounding ?

- Sources of infinity
  1. (built-in) arithmetics
  2. (nested) uninterpreted functions

  ☞ allow for representing the Halting problem (see paper)
- Finiteness guarantees up to the user

  ☞ provide built-in or domain predicates (if needed)

Aggregate Operations

- #count, #sum, #min, #max, #avg, #even, #odd, e.g., in

  l #count{...} u   and   l #sum[...] u

- "traditional" cardinality and weight constraints, looking like

  l #count{...} u   and   l #sum[...] u

A *gringo* 2.x.x program

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

☞ Implicit domains in (body) aggregates

Representing Aggregates

Aggregate Operations

- #count, #sum, #min, #max, #avg, #even, #odd, e.g., in
  `l #count{...} u` and `l #sum[...] u`
- "traditional" cardinality and weight constraints, looking like
  `l #count{...} u` and `l #sum[...] u`

A *gringo* 2.x.x program

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

☞ Implicit domains in (body) aggregates

Representing Aggregates

Aggregate Operations

- `#count`, `#sum`, `#min`, `#max`, `#avg`, `#even`, `#odd`, e.g., in
  `l #count{...} u` and `l #sum[...] u`
- "traditional" cardinality and weight constraints, looking like
  `l #count{...} u` and `l #sum[...] u`

A *gringo* 3.x.x program

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

☞ Implicit domains in (body) aggregates

Optimization Statements (1)

Declarativeness ?

By *lparse* convention

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

map to

```
#minimize[ p(X) = X, not p(Y) = Y' ].
```

where `X` $\ll$ `Y'`, but

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

map to

```
#minimize[ p(X) = X', not p(Y) = Y ].
```

where `Y` $\ll$ `X'`.

Optimization Statements (1)

Declarativeness ?

By *lparse* convention

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

map to

```
#minimize[ p(X) = X, not p(Y) = Y' ].
```

where `X` $\ll$ `Y'`, but

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

map to

```
#minimize[ p(X) = X', not p(Y) = Y ].
```

where `Y` $\ll$ `X'`.

Optimization Statements (2)

Precedence Levels in *gringo* 3.x.x

```
#minimize[ p(X) = X @ 1 ].
#maximize[ p(Y) = Y @ 2 ].
```

and

```
#maximize[ p(Y) = Y @ 2 ].
#minimize[ p(X) = X @ 1 ].
```

each map to

```
#minimize[ p(X) = X, not p(Y) = Y' ].
```

where `X` $\ll$ `Y'`.

☞ Precendence levels can also be specified via variables, e.g., in

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

Optimization Statements (2)

Precedence Levels in *gringo* 3.x.x

```
#minimize[ p(X) = X @ 1 ].
#maximize[ p(Y) = Y @ 2 ].
```

and

```
#maximize[ p(Y) = Y @ 2 ].
#minimize[ p(X) = X @ 1 ].
```

each map to

```
#minimize[ p(X) = X, not p(Y) = Y' ].
```

where `X` $\ll$ `Y'`.

☞ Precendence levels can also be specified via variables, e.g., in

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

Embedded Scripting Language (lua.org)

Greatest Common Divisor

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

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

Ground Instantiation

```
p(30).        p(42).         p(70).
q(30,42,6).   q(30,70,10).   q(42,70,14).
```

☞ More advanced features include, e.g., database connectivity

Embedded Scripting Language (lua.org)

Greatest Common Divisor

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

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

Ground Instantiation

```
p(30).         p(42).          p(70).
q(30,42,6).    q(30,70,10).    q(42,70,14).
```

☞ More advanced features include, e.g., database connectivity

Embedded Scripting Language (lua.org)

Greatest Common Divisor

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

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

Ground Instantiation

```
p(30).         p(42).          p(70).
q(30,42,6).    q(30,70,10).    q(42,70,14).
```

☞ More advanced features include, e.g., database connectivity

Convenience Features

Output Projection at Atom Level

```
mc(C1,C2) :- hc(C1,V1,C2,V2), C1 != C2.
#hide.
#show mc(C1,C2).
```

Generalized Assignment Operator

```
#const m = 4.  #const n = 2.
product(X*Y) :- .
```

Reification

```
$ gringo --reify   1{ p(X) : X = 1..3 }2 :- not p(3).
rule(pos(sum(1,0,2)),pos(conjunction(0))).   set(0,neg(atom(p(3)))).
wlist(0,0,pos(atom(p(1))),1).  wlist(0,1,pos(atom(p(2))),1).  wlist(0,2,pos(atom(p(3))),1).
```

Convenience Features

Output Projection at Atom Level

```
mc(C1,C2) :- hc(C1,V1,C2,V2), C1 != C2.
#hide.
#show hc(C1,V1,C2,V2) : C1 != C2.
```

Generalized Assignment Operator

```
#const m = 4.  #const n = 2.
product(X*Y) :- .
```

Reification

```
$ gringo --reify   1{ p(X) : X = 1..3 }2 :- not p(3).
rule(pos(sum(1,0,2)),pos(conjunction(0))).   set(0,neg(atom(p(3)))).
wlist(0,0,pos(atom(p(1))),1).  wlist(0,1,pos(atom(p(2))),1).  wlist(0,2,pos(atom(p(3))),1).
```

Convenience Features

Output Projection at Atom Level

```
mc(C1,C2) :- hc(C1,V1,C2,V2), C1 != C2.
#hide.
#show hc(C1,V1,C2,V2) : C1 != C2.
```

Generalized Assignment Operator

```
#const m = 4.  #const n = 2.
product(X*Y) :- X = 1..m, Y = 1..n.
```

Reification

```
$ gringo --reify   1{ p(X) : X = 1..3 }2 :- not p(3).
rule(pos(sum(1,0,2)),pos(conjunction(0))).   set(0,neg(atom(p(3)))).
wlist(0,0,pos(atom(p(1))),1).  wlist(0,1,pos(atom(p(2))),1).  wlist(0,2,pos(atom(p(3))),1).
```

Convenience Features

Output Projection at Atom Level

```
mc(C1,C2) :- hc(C1,V1,C2,V2), C1 != C2.
#hide.
#show hc(C1,V1,C2,V2) : C1 != C2.
```

Generalized Assignment Operator

```
#const m = 4.  #const n = 2.
product(X*Y) :- (X,Y) := (1..m,1..n).
```

Ground Program Reification

```
$ gringo --reify   1{ p(X) : X = 1..3 }2 :- not p(3).
rule(pos(sum(1,0,2)),pos(conjunction(0))).   set(0,neg(atom(p(3)))).
wlist(0,0,pos(atom(p(1))),1).  wlist(0,1,pos(atom(p(2))),1).  wlist(0,2,pos(atom(p(3))),1).
```

Convenience Features

Output Projection at Atom Level

```
mc(C1,C2) :- hc(C1,V1,C2,V2), C1 != C2.
#hide.
#show hc(C1,V1,C2,V2) : C1 != C2.
```

Generalized Assignment Operator

```
#const m = 4.  #const n = 2.
product(X*Y) :- (X,Y) := (1..m,1..n).
```

Ground Program Reification

```
$ gringo --reify   1{ p(X) : X = 1..3 }2 :- not p(3).
rule(pos(sum(1,0,2)),pos(conjunction(0))).   set(0,neg(atom(p(3)))).
wlist(0,0,pos(atom(p(1))),1).  wlist(0,1,pos(atom(p(2))),1).  wlist(0,2,pos(atom(p(3))),1).
```

Convenience Features

Output Projection at Atom Level

```
mc(C1,C2) :- hc(C1,V1,C2,V2), C1 != C2.
#hide.
#show hc(C1,V1,C2,V2) : C1 != C2.
```

Generalized Assignment Operator

```
#const m = 4.  #const n = 2.
product(X*Y) :- (X,Y) := (1..m,1..n).
```

Ground Program Reification

```
$ gringo --reify   1{ p(X) : X = 1..3 }2 :- not p(3).
rule(pos(sum(1,0,2)),pos(conjunction(0))).   set(0,neg(atom(p(3)))).
wlist(0,0,pos(atom(p(1))),1).  wlist(0,1,pos(atom(p(2))),1).  wlist(0,2,pos(atom(p(3))),1).
```

Convenience Features

Output Projection at Atom Level

```
mc(C1,C2) :- hc(C1,V1,C2,V2), C1 != C2.
#hide.
#show hc(C1,V1,C2,V2) : C1 != C2.
```

Generalized Assignment Operator

```
#const m = 4.  #const n = 2.
product(X*Y) :- (X,Y) := (1..m,1..n).
```

Ground Program Reification

```
$ gringo --reify   1{ p(X) : X = 1..3 }2 :- not p(3).
rule(pos(sum(1,0,2)),pos(conjunction(0))).   set(0,neg(atom(p(3)))).
wlist(0,0,pos(atom(p(1))),1).  wlist(0,1,pos(atom(p(2))),1).  wlist(0,2,pos(atom(p(3))),1).
```

Outline

Discussion

gringo 3.x.x

1. Safe programs
2. Rich language of *gringo* 2.x.x (+ extensions)
3. It's free and for free **!**

   ☞ `http://potassco.sourceforge.net`

Work in Progress

- Extension to *dlv*-style aggregates and weak constraints
- Common standard for ASP input languages
  - more than normal logic programs
  - less than ASP-RfC (@ ASP Competition 2011)

  ☞ backward compatibility + coherent semantics
- Comprehensive documentation . . . sorry for that **!**

Discussion

gringo 3.x.x

1. Safe programs
2. Rich language of *gringo* 2.x.x (+ extensions)
3. It's free and for free **!**

   ☞ `http://potassco.sourceforge.net`

Work in Progress

- Extension to *dlv*-style aggregates and weak constraints
- Common standard for ASP input languages
  - more than normal logic programs
  - less than ASP-RfC (@ ASP Competition 2011)

  ☞ backward compatibility + coherent semantics
- Comprehensive documentation . . . sorry for that **!**