Grounding: Overview

1 Background

2 Bottom Up Grounding

3 Semi-naive Evaluation Based Grounding

4 On-the-fly Simplifications

5 Rule Instantiation

Outline

1 Background

2 Bottom Up Grounding

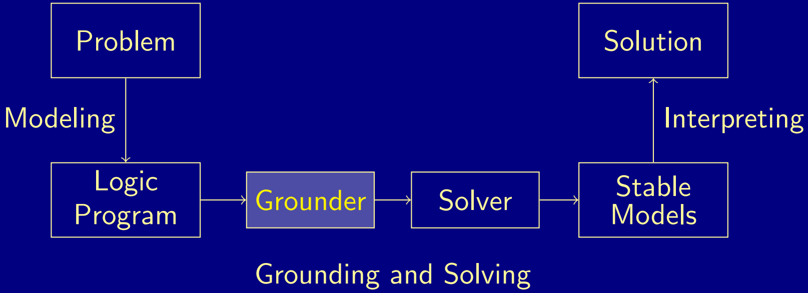
3 Semi-naive Evaluation Based Grounding

4 On-the-fly Simplifications

5 Rule Instantiation

August 3, 2015

Introduction



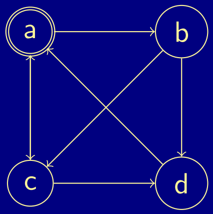
- some grounders (in chronological order)
  - *lparse* (grounding using domain predicates)
  - *dlv* (semi-naive evaluation based grounding)
  - *gringo* (semi-naive evaluation based since version 3)

Hamiltonian Cycle Instance

```
% vertices
node(a).  node(b).
node(c).  node(d).

% edges
edge(a,b).  edge(a,c).
edge(b,c).  edge(b,d).
edge(c,a).  edge(c,d).
edge(d,a).

% starting point (for presentation purposes)
start(a).
```



Hamiltonian Cycle Encoding

```
% generate path
path(X,Y) :- not omit(X,Y), edge(X,Y).
omit(X,Y) :- not path(X,Y), edge(X,Y).

% at most one incoming/outgoing edge
:- path(X,Y), path(X',Y), X < X'.
:- path(X,Y), path(X,Y'), Y < Y'.

% at least one incoming/outgoing edge
on_path(Y) :- path(X,Y), path(Y,Z).
:- node(X), not on_path(X).

% connectedness
reach(X) :- start(X).
reach(Y) :- reach(X), path(X,Y).
:- node(X), not reach(X).
```

Grounding

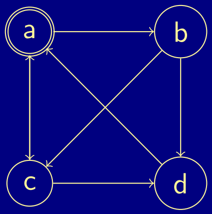
- Safety
  - each variable has to occur in a positive body element
  - consider: `p(X) :- not q(X).`
- Herbrand universe
  - all constants in program and
    all functions over function symbols in program
- Herbrand base
  - all atoms over predicates in program
    with terms from Herbrand universe
- Instance of a rule
  - all variables replaced with elements from Herbrand universe
- Grounding of a program
  - $ground(P)$ is the union of all instances of rules in P

Example: Size of Grounding

```
% Herbrand Universe: {a,b,c,d}
12 facts from instance
% path(X,Y) :- not omit(X,Y), edge(X,Y).
% omit(X,Y) :- not path(X,Y), edge(X,Y).
% reach(Y) :- reach(X), path(X,Y).
16 rules + 16 rules + 16 rules
% on_path(Y) :- path(X,Y), path(Y,Z).
% :- path(X,Y), path(X',Y), X < X'.
% :- path(X,Y), path(X,Y'), Y < Y'.
64 rules + 64 rules + 64 rules
% reach(X) :- start(X).
% :- node(X), not on_path(X).
% :- node(X), not reach(X).
4 rules + 4 rules + 4 rules
```

Example: Unnecessary Rules I

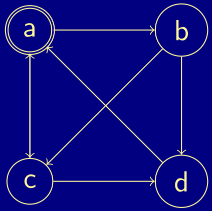
```
% path(X,Y) :- not omit(X,Y), edge(X,Y).
path(a,a) :- not omit(a,a), edge(a,a).
path(a,b) :- not omit(a,b), edge(a,b).
path(a,c) :- not omit(a,c), edge(a,c).
path(a,d) :- not omit(a,d), edge(a,d).
             ⋮
path(d,a) :- not omit(d,a), edge(d,a).
path(d,b) :- not omit(d,b), edge(d,b).
path(d,c) :- not omit(d,c), edge(d,d).
path(d,d) :- not omit(d,d), edge(d,d).
```



Example: Unnecessary Rules II

```
% :- path(X,Y), path(X',Y), X < X'.
:- path(a,a), path(a,a), a < a.
:- path(a,b), path(a,b), a < a.
:- path(a,c), path(a,c), a < a.
:- path(a,d), path(a,d), a < a.

:- path(a,a), path(b,a), a < b.
:- path(a,b), path(b,b), a < b.
:- path(a,c), path(b,c), a < b.
:- path(a,d), path(b,d), a < b.
              ⋮
:- path(d,d), path(d,d), d < d.
```



Outline

1 Background

2 Bottom Up Grounding

3 Semi-naive Evaluation Based Grounding

4 On-the-fly Simplifications

5 Rule Instantiation

Bottom Up Grounding

- ground relevant rules by incrementally extending the Herbrand base
- $ground_D(P) = \{r \in ground(P) \mid body(r)^+ \subseteq D,$ all comparison literals in $body(r)$ are satisfied$\}$

```
function GROUND_BOTTOM_UP(P, D)
    G ← ground_D(P)
    if head(G) ⊈ D then
        return GROUND_BOTTOM_UP(P, D ∪ head(G))
    return G
```

- given safe program P and set of ground facts I (typically corresponds to encoding and instance), $P \cup I$ is equivalent to GROUND_BOTTOM_UP$(P, head(I)) \cup I$

Example: Bottom Up Grounding Step 1

```
% Step 1
path(a,b) :- not omit(a,b), edge(a,b).
          ⋮ % 7 rules total
path(d,a) :- not omit(d,a), edge(d,a).

omit(a,b) :- not path(a,b), edge(a,b).
          ⋮ % 7 rules total
omit(d,a) :- not path(d,a), edge(d,a).

:- node(a), not on_path(a).  :- node(b), not on_path(b).
:- node(c), not on_path(c).  :- node(d), not on_path(d).

:- node(a), not reach(a).  :- node(b), not reach(b).
:- node(c), not reach(c).  :- node(d), not reach(d).

reach(a) :- start(a).
```

Example: Bottom Up Grounding Step 2

```
% Step 2 and rules of Step 1
:- path(a,c), path(b,c), a < b.
:- path(b,d), path(c,d), b < c.
:- path(c,a), path(d,a), c < d.

:- path(a,b), path(a,c), b < c.
:- path(c,a), path(c,d), a < d.
:- path(b,c), path(b,d), c < d.

on_path(a) :- path(a,b), path(c,a).
             ⋮ % 12 rules total
on_path(d) :- path(d,a), path(c,d).

reach(b) :- reach(a), path(a,b).
reach(c) :- reach(a), path(a,c).
```

Example: Bottom Up Grounding Step 3 and 4

```
% Step 3 and rules of Step 2
reach(c) :- reach(b), path(b,c).
reach(d) :- reach(b), path(b,d).
reach(a) :- reach(c), path(c,a).
reach(d) :- reach(c), path(c,d).

% Step 4 and rules of Step 3
reach(a) :- reach(d), path(d,a).
```

Properties of Bottom Up Grounding

- grounds only relevant rules
  - each positive body literal has a non-cyclic derivation (ignoring negative literals)
- regrounds rules from previous steps

---

function GROUND_BOTTOM_UP(P, D)
    $G \leftarrow ground_D(P)$
    if $head(G) \not\subseteq D$ **then**
        return GROUND_BOTTOM_UP($P, D \cup head(G)$)
    return G

- does not perform simplifications

Improving Bottom Up Grounding

- use dependencies to focus grounding
  - begin with partial Herbrand base given by facts
  - use rule dependency graph of program to obtain components that can be grounded successively
- adapt semi-naive evaluation put forward in the database field
  - avoids redundancies when grounding
- perform simplifications during grounding
  - remove literals from rule bodies if possible
  - omit rules if body cannot be satisfied

Program Dependencies

- dependency graph of program P
  - rule r_2 depends on rule r_1
    if $b \in body(r_2)^+ \cup body(r_2)^-$ unifies with $h \in head(r_1)$
  - $G_P = (P, E)$ where $E = \{(r_1, r_2) \mid r_2 \text{ depends on } r_1\}$
- positive dependency graph of program P
  - rule r_2 positively depends on rule r_1
    if $b \in body(r_2)^+$ unifies with $h \in head(r_1)$
  - $G_P^+ = (P, E)$ where $E = \{(r_1, r_2) \mid r_2 \text{ positively depends on } r_1\}$
- let $L_P = (C_{1,1}, \dots, C_{1,m_1}, \dots, C_{n,1}, \dots, C_{n,m_n})$ where
  - $(C_1, \dots, C_n)$ is a topological ordering of G_P
  - $(C_{i,1}, \dots, C_{i,m_i})$ is a topological ordering of each $G_{C_i}^+$

Example: Dependencies

Component$_{1,1}$: `omit(X,Y) :- not path(X,Y), edge(X,Y).`

Component$_{1,2}$: `path(X,Y) :- not omit(X,Y), edge(X,Y).`

Component$_{2,1}$: `:- path(X,Y), path(X',Y), X < X'.`

Component$_{3,1}$: `:- path(X,Y), path(X,Y'), Y < Y'.`

Component$_{4,1}$: `on_path(Y) :- path(X,Y), path(Y,Z).`

Component$_{5,1}$: `:- node(X), not on_path(X).`

Component$_{6,1}$: `reach(X) :- start(X).`

Component$_{7,1}$: `reach(Y) :- reach(X), path(X,Y).`

Component$_{8,1}$: `:- node(X), not reach(X).`

Grounding With Dependencies

```
function GROUND_WITH_DEPENDENCIES(P, D)
    G ← ∅
    foreach C in L_P do
        G′ ← GROUND_BOTTOM_UP(C, D)
        (G, D) ← (G ∪ G′, D ∪ head(G′))
    return G
```

- given safe program P and set of facts I, $P \cup I$ is equivalent to $\textsc{ground\_with\_dependencies}(P, head(I)) \cup I$

Example: Grounding with Dependencies

```
% Component_{1,1}
omit(a,b) :- not path(a,b), edge(a,b).
          ⋮ % 7 rules total
omit(d,a) :- not path(d,a), edge(d,a).

% Component_{1,2}
path(a,b) :- not omit(a,b), edge(a,b).
          ⋮ % 7 rules total
path(d,a) :- not omit(d,a), edge(d,a).

...
```

- no regrounding if there is no positive recursion in a component

Example: Grounding Component$_{7,1}$

```
% Step 1
reach(b) :- reach(a), path(a,b).
reach(c) :- reach(a), path(a,c).

% Step 2 and rules of Step 1
reach(c) :- reach(b), path(b,c).
reach(d) :- reach(b), path(b,d).
reach(a) :- reach(c), path(c,a).
reach(d) :- reach(c), path(c,d).

% Step 3 and rules of Step 2
reach(a) :- reach(d), path(d,a).

% less regrounding but still...
```

Outline

1 Background

2 Bottom Up Grounding

3 Semi-naive Evaluation Based Grounding

4 On-the-fly Simplifications

5 Rule Instantiation

Recursive Atoms

- given $L_P = (C_1, \ldots, C_n)$, an atom a_1 is recursive in component C_i if a_1 unifies a_2 such that
  - $r_1 \in C_i$ and $r_2 \in C_j$ with $i \leq j$,
  - $a_1 \in body(r_1)^+ \cup body(r_1)^-$, and
  - $a_2 \in head(r_2)$

Example: Recursive Atoms

Component$_{1,1}$: `omit(X,Y) :- not path(X,Y), edge(X,Y).`

Component$_{1,2}$: `path(X,Y) :- not omit(X,Y), edge(X,Y).`

Component$_{2,1}$: `:- path(X,Y), path(X',Y), X < X'.`

Component$_{3,1}$: `:- path(X,Y), path(X,Y'), Y < Y'.`

Component$_{4,1}$: `on_path(Y) :- path(X,Y), path(Y,Z).`

Component$_{5,1}$: `:- node(X), not on_path(X).`

Component$_{6,1}$: `reach(X) :- start(X).`

Component$_{7,1}$: `reach(Y) :- reach(X), path(X,Y).`

Component$_{8,1}$: `:- node(X), not reach(X).`

Preparing Components

- the set of prepared rules for $r \in C$ is

$$\left\{\begin{array}{llllllll} h :- n(b_1), & a(b_2), & a(b_3), & \dots, & a(b_{i-2}), & a(b_{i-1}), & a(b_i), & B \\ h :- o(b_1), & n(b_2), & a(b_3), & \dots, & a(b_{i-2}), & a(b_{i-1}), & a(b_i), & B \\ \vdots & \vdots & & \ddots & & & \vdots & \vdots \\ h :- o(b_1), & o(b_2), & o(b_3), & \dots, & o(b_{i-2}), & n(b_{i-1}), & a(b_i), & B \\ h :- o(b_1), & o(b_2), & o(b_3), & \dots, & o(b_{i-2}), & o(b_{i-1}), & n(b_i), & B \end{array}\right\}$$

or $\{h \;:-\; n(b_{i+1}), \dots, n(b_j), b_{j+1}, \dots, b_n\}$ if $i = 0$

where $body(r) = \{b_1, \dots, b_i, b_{i+1} \dots, b_j, b_{j+1}, \dots, b_n\}$,
$b_k \in body(r)^+$ for $1 \leq k \leq i$ is recursive,
$b_k \in body(r)^+$ for $i < k \leq j$ is not recursive, and
$B = a(b_{i+1}), \dots, a(b_j), b_{j+1}, \dots, b_n$

- a prepared component is the union of all its prepared rules

Example: Preparing Components

```
% prepared Component_{1,1}
omit(X,Y) :- n(edge(X,Y)), not path(X,Y).
% prepared Component_{1,2}
path(X,Y) :- n(edge(X,Y)), not omit(X,Y).
% prepared Component_{2,1}
:- n(path(X,Y)), n(path(X',Y)), X < X'.
...
% prepared Component_{7,1}
reach(Y) :- n(reach(X)), a(path(X,Y)).
...
```

Semi-naive Evaluation-based Grounding

function GROUND_SEMI_NAIVE(P, A)
  $G \leftarrow \emptyset$
  foreach C **in** L_P **do**
    $(O, N) \leftarrow (\emptyset, A)$
    repeat
      let $D_p = \{p(a) \mid a \in D\}$ for set D of atoms
      $G' \leftarrow ground_{O_o \cup N_n \cup A_a}(\text{prepared } C)$
      $N \leftarrow head(G') \setminus A$
      $(G, O, A) \leftarrow (G \cup G', A, N \cup A)$
    until $N = \emptyset$
  return G with $o/1$, $n/1$, $a/1$ stripped from positive bodies

- given safe program P and set of facts I, $P \cup I$ is equivalent to GROUND_SEMI_NAIVE$(P, head(I)) \cup I$

Example: Grounding Component$_{7,1}$

```
% grounding of
% reach(Y) :- n(reach(X)), a(path(X,Y)).

% Step 1 with N = A from previous step (reach(a) ∈ A)
reach(b) :- n(reach(a)), a(path(a,b)).
reach(c) :- n(reach(a)), a(path(a,c)).

% Step 2 with N = { reach(b), reach(c) }
reach(c) :- n(reach(b)), a(path(b,c)).
reach(d) :- n(reach(b)), a(path(b,d)).
reach(a) :- n(reach(c)), a(path(c,a)).
reach(d) :- n(reach(c)), a(path(c,d)).

% Step 3 with N = { reach(d) }
reach(a) :- n(reach(d)), a(path(d,a)).
```

Example: Grounding Component$_{7,1}$

```
% grounding of
% reach(Y) :- n(reach(X)), a(path(X,Y)).

% Step 1 with N = A from previous step (reach(a) ∈ A)
reach(b) :- reach(a), path(a,b).
reach(c) :- reach(a), path(a,c).

% Step 2 with N = { reach(b), reach(c) }
reach(c) :- reach(b), path(b,c).
reach(d) :- reach(b), path(b,d).
reach(a) :- reach(c), path(c,a).
reach(d) :- reach(c), path(c,d).

% Step 3 with N = { reach(d) }
reach(a) :- reach(d), path(d,a).

% without n/1 and a/1 of course
```

Example: Nonlinear Programs

```
trans(U,V) :- edge(U,V).
trans(U,W) :- trans(U,V), trans(V,W).
```

```
% prepared Component 1:
trans(U,V) :- n(edge(U,V)).

% prepared Component 2:
trans(U,W) :- n(trans(U,V)), a(trans(V,W)).
trans(U,W) :- o(trans(U,V)), n(trans(V,W)).
```

Example: Nonlinear Programs

```
trans(U,V) :- edge(U,V).
% trans(U,W) :- trans(U,V), trans(V,W).
% better written as:
trans(U,W) :- trans(U,V), edge(V,W).

% prepared Component 1:
trans(U,V) :- n(edge(U,V)).

% prepared Component 2:
trans(U,W) :- n(trans(U,V)), a(edge(V,W)).
```

Outline

1 Background

2 Bottom Up Grounding

3 Semi-naive Evaluation Based Grounding

4 On-the-fly Simplifications

5 Rule Instantiation

Propagation of Facts

- simplifications are performed on-the-fly
  (rules are printed immediately but not stored in *gringo*)
- maintain a set of fact atoms
- remove facts from positive body
- discard rules with negative literals over a fact
- discard rules whenever the head is a fact
- gather new facts whenever a rule body is empty

Example: Propagation of Facts

```
...
path(a,b) :- not omit(a,b), edge(a,b).
...
reach(a) :- start(a).
```

Example: Propagation of Facts

```
...
path(a,b) :- not omit(a,b).
...
reach(a). % reach(a) is added as fact
```

Example: Propagation of Facts

```
...
path(a,b) :- not omit(a,b).
...
reach(a). % reach(a) is added as fact

...

:- node(a), not reach(a).
...
```

Example: Propagation of Facts

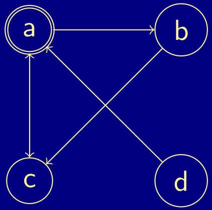
```
...
path(a,b) :- not omit(a,b).
...
reach(a). % reach(a) is added as fact

...

~~:- node(a), not reach(a).~~ % rule is discarded
...
```

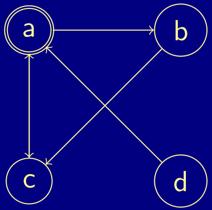
Propagation of Negative Literals

- non-recursive negative literals not in the current base A can be removed from rule bodies
- stratified logic programs are completely evaluated during grounding
- consider the instance where node `d` is not reachable



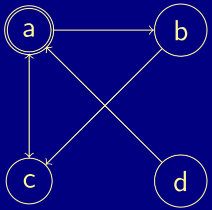
Example: Propagation of Negative Literals

```
path(a,b) :- not omit(a,b).
path(a,c) :- not omit(a,c).
path(b,c) :- not omit(b,c).
path(c,a) :- not omit(c,a).
path(d,a) :- not omit(d,a).
...
reach(a).
reach(b) :- path(a,b).
reach(c) :- path(a,c).
reach(c) :- path(b,c), reach(b).
...
% reach(X) is not recursive and reach(d) ∉ A
:- not reach(b).
:- not reach(c).
:- not reach(d). % remove not reach(d) from body
```



Example: Propagation of Negative Literals

```
path(a,b) :- not omit(a,b).
path(a,c) :- not omit(a,c).
path(b,c) :- not omit(b,c).
path(c,a) :- not omit(c,a).
path(d,a) :- not omit(d,a).
...
reach(a).
reach(b) :- path(a,b).
reach(c) :- path(a,c).
reach(c) :- path(b,c), reach(b).
...
% reach(X) is not recursive and reach(d) ∉ A
:- not reach(b).
:- not reach(c).
:- . % inconsistency detected during grounding
```



Conclusion/Summary

- grounding algorithms for normal logic programs (with integrity constraints)
- language features not covered here
  - (recursive) aggregates
  - conditional literals
  - optimization statements
  - disjunctions
  - arithmetic functions
  - syntactic sugar to write more compact encodings
  - safety of = relation (for aggregates and terms)
  - python/lua integration
    - external functions
    - control over grounding and solving

Outline

1 Background

2 Bottom Up Grounding

3 Semi-naive Evaluation Based Grounding

4 On-the-fly Simplifications

5 Rule Instantiation

Rule Instantiation

- the following slides show how to ground individual rules
- I am probably not going to show them

Safe Body Order

- given safe rule r, the tuple $(b_1, \ldots, b_n)$ is a safe body order if
  - $\{b_1, \ldots, b_n\} = body(r)$
  - the body $\{b_1, \ldots, b_i\}$ is safe for each i
- for example given rule `:- node(X), not reach(X).`
  - $(\texttt{node(X)}, \texttt{not reach(X)})$ is a safe body order
  - $(\texttt{not reach(X)}, \texttt{node(X)})$ is not a safe body order

Safe Body Order

- given safe rule r, the tuple $(b_1, \ldots, b_n)$ is a safe body order if
  - $\{b_1, \ldots, b_n\} = body(r)$
  - the body $\{b_1, \ldots, b_i\}$ is safe for each i
- for example given rule `:- node(X), not reach(X).`
  - (`node(X)`, `not reach(X)`) is a safe body order
  - (`not reach(X)`, `node(X)`) is not a safe body order

Matching Body Literals

- $match_{F,D}(\sigma, b)$ is the set of all matches for literal b
  - σ is a substitution
  - F are facts (set of ground atoms)
  - D is the domain (set of ground atoms)
  - $\sigma' \in match_{F,D}(\sigma, b)$ if
    - $\sigma \subseteq \sigma'$ and $vars(b) \subseteq vars(\sigma') \subseteq vars(b) \cup vars(\sigma)$,
    - $b\sigma'$ holds if b is a comparison literal,
    - $b\sigma' \in D$ if b is an atom, and
    - $a\sigma' \notin F$ if b is a symbolic literal of form `not` a
- for example given body: `p(X),q(X,Y),not r(Y)`
  - $F = \{r(3)\}$ and $D = \{p(1), q(1,2), q(1,3), r(3)\}$
  - $match_{F,D}(\emptyset, \texttt{p(X)}) = \{\{X \mapsto 1\}\}$
  - $match_{F,D}(\{X \mapsto 1\}, \texttt{q(X,Y)}) = \{\{X \mapsto 1, Y \mapsto 2\}, \{X \mapsto 1, Y \mapsto 3\}\}$
  - $match_{F,D}(\{X \mapsto 1, Y \mapsto 2\}, \texttt{not r(Y)}) = \{\{X \mapsto 1, Y \mapsto 2\}\}$
  - $match_{F,D}(\{X \mapsto 1, Y \mapsto 3\}, \texttt{not r(Y)}) = \emptyset$

Matching Body Literals

- $match_{F,D}(\sigma, b)$ is the set of all matches for literal b
  - σ is a substitution
  - F are facts (set of ground atoms)
  - D is the domain (set of ground atoms)
  - $\sigma' \in match_{F,D}(\sigma, b)$ if
    - $\sigma \subseteq \sigma'$ and $vars(b) \subseteq vars(\sigma') \subseteq vars(b) \cup vars(\sigma)$,
    - $b\sigma'$ holds if b is a comparison literal,
    - $b\sigma' \in D$ if b is an atom, and
    - $a\sigma' \notin F$ if b is a symbolic literal of form $\texttt{not}\ a$
- for example given body: `p(X),q(X,Y),not r(Y)`
  - $F = \{r(3)\}$ and $D = \{p(1), q(1,2), q(1,3), r(3)\}$
  - $match_{F,D}(\emptyset, \texttt{p(X)}) = \{\{X \mapsto 1\}\}$
  - $match_{F,D}(\{X \mapsto 1\}, \texttt{q(X,Y)}) = \{\{X \mapsto 1, Y \mapsto 2\}, \{X \mapsto 1, Y \mapsto 3\}\}$
  - $match_{F,D}(\{X \mapsto 1, Y \mapsto 2\}, \texttt{not r(Y)}) = \{\{X \mapsto 1, Y \mapsto 2\}\}$
  - $match_{F,D}(\{X \mapsto 1, Y \mapsto 3\}, \texttt{not r(Y)}) = \emptyset$

Rule Grounding by Backtracking

function $\text{GROUND\_BACKTRACK}_{r,R,D}(\sigma, F, (b_1, \ldots, b_n))$
  if $n = 0$ **then**
    let $H = head(r\sigma)$
      $B = body(r\sigma)^+ \setminus F \cup$
        $\{\texttt{not}\ a\sigma \mid a \in body(r)^- \setminus R, a \in D\} \cup$
        $\{\texttt{not}\ a\sigma \mid a \in body(r)^- \cap R\}$
    if $B = \emptyset$ **then** $F \leftarrow F \cup H$
    return $(\{H\ \texttt{:-}\ B \mid B^- \cap F = \emptyset, H \cap F = \emptyset\}, F)$
  else
    $G \leftarrow \emptyset$
    foreach $\sigma' \in match_{F,D}(\sigma, b_1)$ **do**
      $(G, F) \leftarrow (G, F) \sqcup \text{GROUND\_BACKTRACK}_{r,R,D}(\sigma', F, (b_2, \ldots, b_n))$
    return (G, F)