# Finite Groundings for ASP with Functions

## A Journey through Consistency
## (Extended Abstract)

Lukas Gerlach[1] David Carral[2] Markus Hecher[3]

[1]Knowledge-Based Systems Group, TU Dresden, Germany
[2]LIRMM, Inria, University of Montpellier, CNRS, France
[3]Massachusetts Institute of Technology, United States

02.11.2024

TECHNISCHE UNIVERSITÄT DRESDEN

International Center for Computational Logic

# Outline for this Talk

- ⚽ Goal: Why do functions make ASP so hard and how can we address this?

# Outline for this Talk

- ⚽ Goal: Why do functions make ASP so hard and how can we address this?
- 🥳 Motivating Example: Programs often use “artificial bounds”

# Outline for this Talk

- ⚽ Goal: Why do functions make ASP so hard and how can we address this?
- 🥳 Motivating Example: Programs often use “artificial bounds”
- 📋 Summary of main results

# Outline for this Talk

- ⚽ Goal: Why do functions make ASP so hard and how can we address this?
- 🥳 Motivating Example: Programs often use “artificial bounds”
- 📋 Summary of main results
- 🤓 Elaboration on proofs for high level of undecidability

# Outline for this Talk

- ⚽ Goal: Why do functions make ASP so hard and how can we address this?
- 🥳 Motivating Example: Programs often use “artificial bounds”
- 📋 Summary of main results
- 🤓 Elaboration on proofs for high level of undecidability
- 🗄️ Two Classes of Programs that eliminate key problems

# Outline for this Talk

- ⚽ Goal: Why do functions make ASP so hard and how can we address this?
- 🥳 Motivating Example: Programs often use “artificial bounds”
- 📋 Summary of main results
- 🤓 Elaboration on proofs for high level of undecidability
- 🗄 Two Classes of Programs that eliminate key problems
- ⚙ Proposal of a grounding procedure

# How are Functions used in ASP?

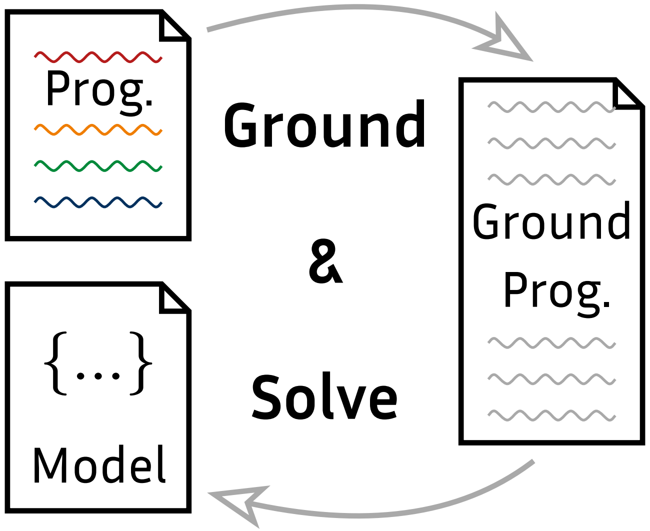
**General Procedure:**

ASP systems like Clingo and (i)DLV work as follows.

# How are Functions used in ASP?

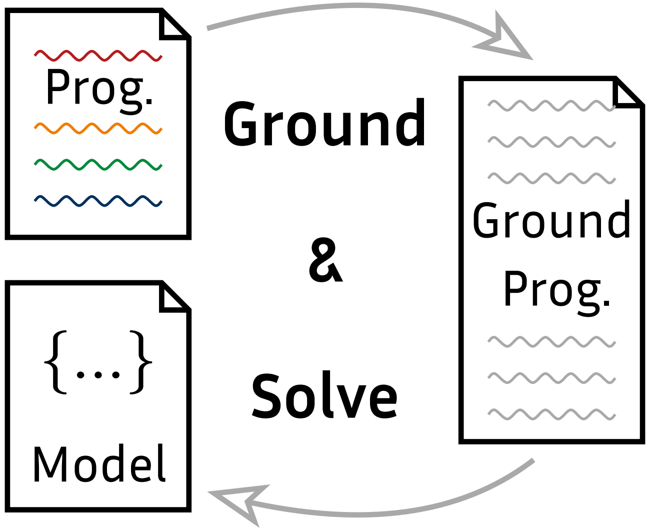
**General Procedure:**

ASP systems like Clingo and (i)DLV work as follows.



# How are Functions used in ASP?

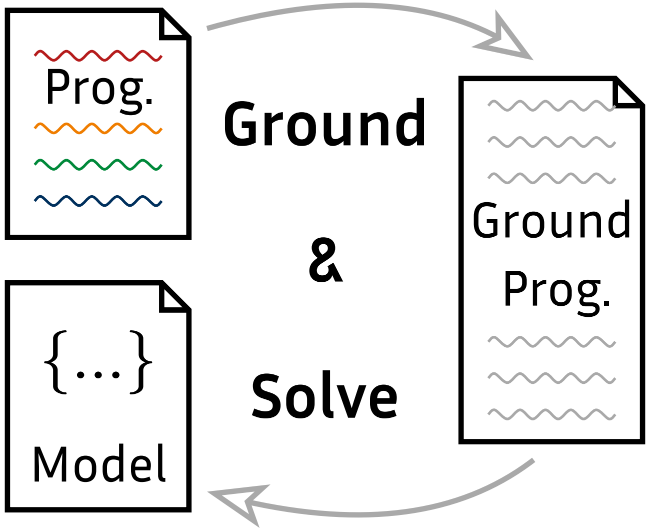
**General Procedure:** ASP systems like Clingo and (i)DLV work as follows.



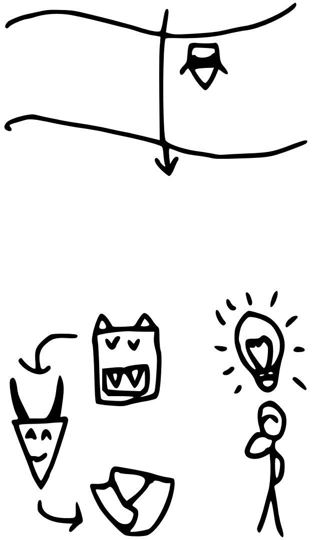

**Example:** Bring wolf, goat, and cabbage over river.

# How are Functions used in ASP?

**General Procedure:** ASP systems like Clingo and (i)DLV work as follows.

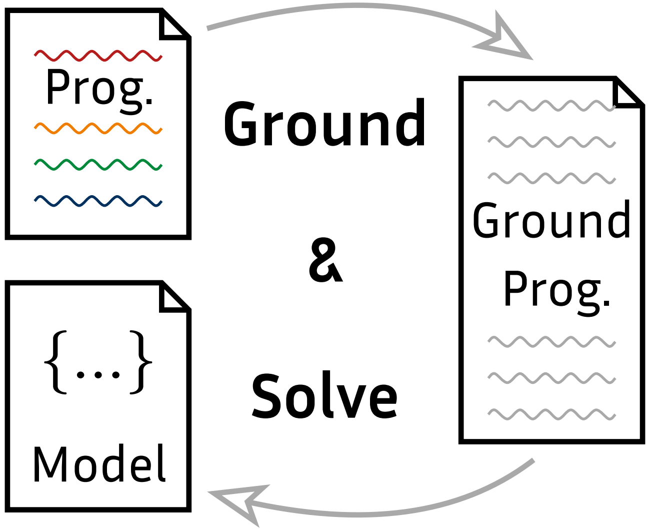



**Example:** Bring wolf, goat, and cabbage over river.

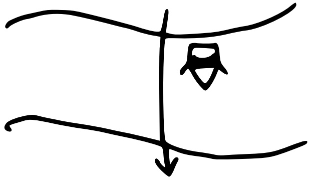


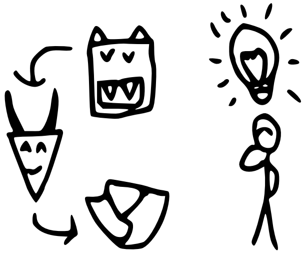
# How are Functions used in ASP?

**General Procedure:** ASP systems like Clingo and (i)DLV work as follows.




**Example:** Bring wolf, goat, and cabbage over river.





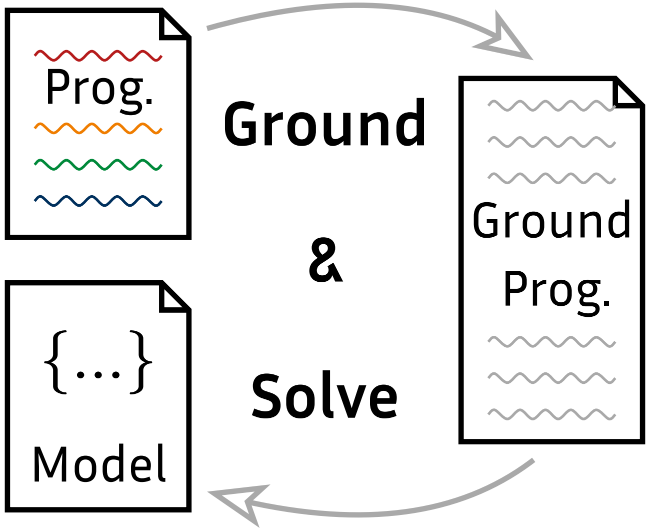
WolfGoatCabbage-GameRules.asp

```
bank(east). bank(west).
opposite(east, west). opposite(west,east).
passenger(wolf). passenger(goat). passenger(cabbage).
position(wolf, west, 0). position(goat, west, 0).
position(cabbage, west, 0). position(farmer, west, 0). eats(wolf,
goat). eats(goat, cabbage).

win(N) :- position(wolf, east, N),
  position(goat, east, N),
  position(cabbage, east, N).
winEnd :- win(N).
lose :- position(X, B, N),
  position(Y, B, N), eats(X, Y),
  position(farmer, C, N), opposite(B, C).
:- not winEnd. % we must win eventually
:- lose.       % we must not lose
```

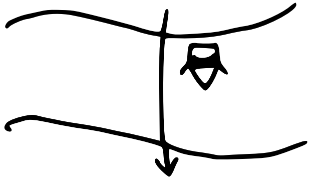
Encode Basic Game Rules

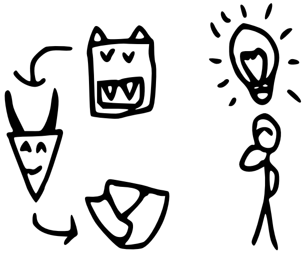
# How are Functions used in ASP?

**General Procedure:** ASP systems like Clingo and (i)DLV work as follows.




**Example:** Bring wolf, goat, and cabbage over river.





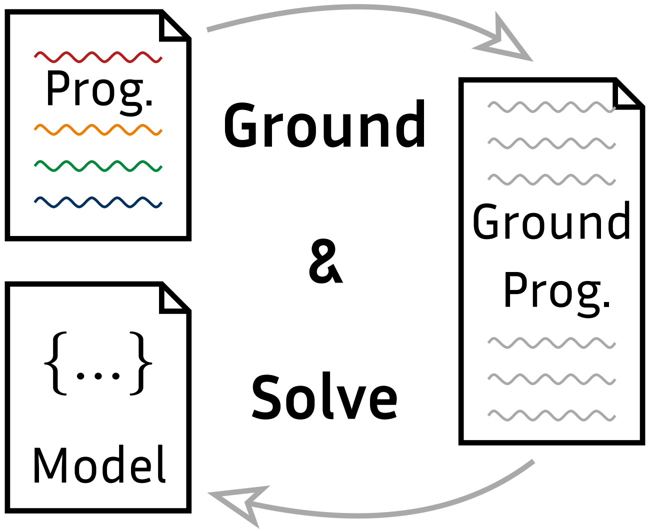
WolfGoatCabbage-ChooseMove.asp

```
% farmer either goes alone ...
goAlone(N) :- position(farmer, B, N),
  not takeSome(N), not win(N).
% ... or takes some passenger ...
takeSome(N) :- position(farmer, B, N),
  passenger(Y), position(Y, B, N),
  not goAlone(N), not win(N).
% ... and needs to pick exactly one
transport(X, N) :- takeSome(N),
  position(X, B, N), position(farmer, B, N),
  passenger(X), not othertransport(X, N).
othertransport(X, N) :- position(X, B, N),
  transport(Y, N), X != Y.
```

Choose whom to transport
(in each step)

# How are Functions used in ASP?

**General Procedure:** ASP systems like Clingo and (i)DLV work as follows.



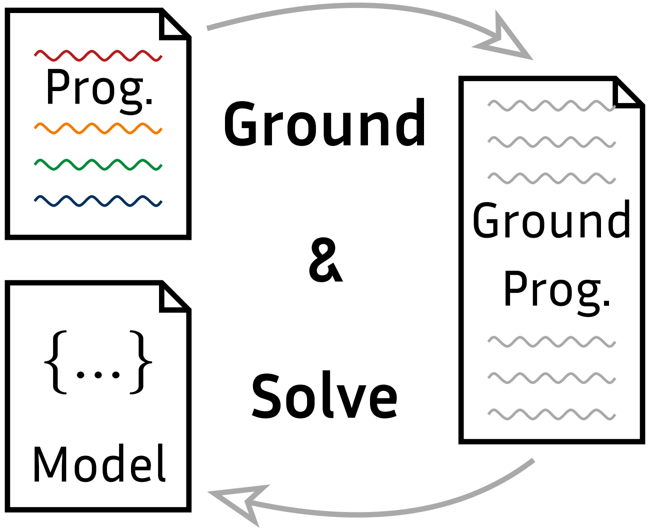

**Example:** Bring wolf, goat, and cabbage over river.

WolfGoatCabbage-UpdatePositions-LimitSteps.asp

```
% Numbers are functions! e.g. 2 = s(s(0)); N+1 = s(N)
steps(0..100). % Common Hack to contain Ground program

% based on the choice, we update positions
position(X, C, N+1) :- transport(X, N), position(X, B, N),
  opposite(B, C), steps(N+1).

position(X, B, N+1) :- position(X, B, N), passenger(X),
  not transport(X, N), not win(N), steps(N+1).

position(farmer, C, N+1) :- position(farmer, B, N),
  opposite(B, C), not win(N), steps(N+1).
```

# How are Functions used in ASP?

**General Procedure:** ASP systems like Clingo and (i)DLV work as follows.




**Example:** Bring wolf, goat, and cabbage over river.

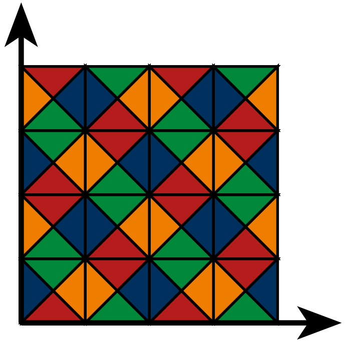
WolfGoatCabbage-AvoidRedundancies.asp

```
% we forbid configurations that already occurred
change(N, M) :- position(X, B, N), position(X, C, M),
  opposite(B, C), N < M.

redundant :- position(X, B, N), position(X, B, M),
  N < M, not change(N, M).

:- redundant.
```

# Why are Functions so hard and what to do about it?

**Understand:**

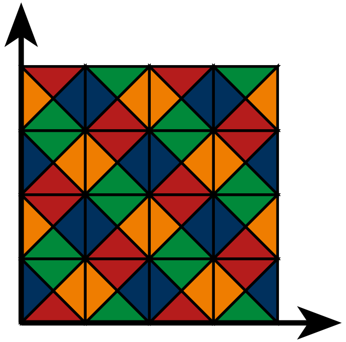
- Consistency is $\Sigma_1^1$-complete. [Dan+01, MNR94]
- We reprove e.g. hardness by reduction from a variant of the tiling problem. [Har86]
- We characterize **frugal** and **non-proliferous** programs.



# Why are Functions so hard and what to do about it?

## Understand:

- Consistency is $\Sigma_1^1$-complete. [Dan+01, MNR94]
- We reprove e.g. hardness by reduction from a variant of the tiling problem. [Har86]
- We characterize **frugal** and **non-proliferous** programs.



## Overcome:

We propose `GroundNotForbidden` as a grounding procedure ignoring **forbidden atoms** that yields finite grounding for frugal and non-proliferous programs.

GroundNotForbidden.pseudo; Output: $P_g$

1. Set $i := 1, A_0 := \emptyset, P_g := \emptyset$.
2. Set $A_i := A_{i-1}$. For each ground rule $r = H_r \leftarrow B_r^+, B_r^-$ with $B_r^+ \subseteq A_{i-1}$, (a) if $H_r$ is *forbidden* add $\leftarrow B_r^+, B_r^-$ to $P_g$, (b) otherwise add $r$ to $P_g$ and $H_r$ to $A_i$.
3. Stop if $A_i = A_{i-1}$; else inc $i$, go to 2.

# Consistency is (very) hard, i.e. $\Sigma_1^1$-complete

**Hardness:** Reduction
from "Recurring Tiling"

# Consistency is (very) hard, i.e. $\Sigma^1_1$-complete

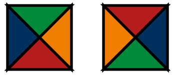
**Hardness:** Reduction

from "Recurring Tiling"

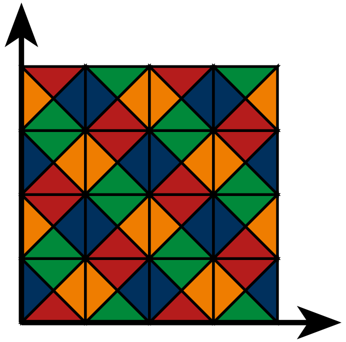
Given:  

# Consistency is (very) hard, i.e. $\Sigma_1^1$-complete

**Hardness:** Reduction
from "Recurring Tiling"

Given: 

Wanted:



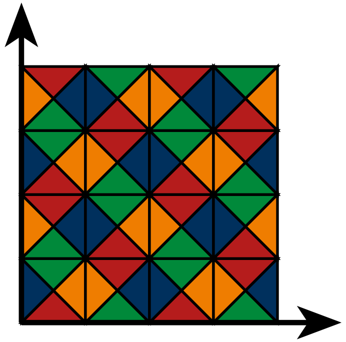
with tile  infinitely
often in first column.

# Consistency is (very) hard, i.e. $\Sigma_1^1$-complete

**Hardness:** Reduction from "Recurring Tiling"

Given:  

Wanted:



with tile  infinitely often in first column.

RecurringTiling.asp

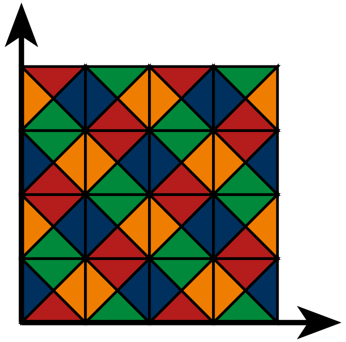
```
dom(c0).
dom(s(X)) :- dom(X).
tile0(X, Y) :- dom(X), dom(Y), not tile1(X, Y)
tile1(X, Y) :- dom(X), dom(Y), not tile0(X, Y)
:- tile0(X, Y), tile0(s(X), Y).
:- tile0(X, Y), tile0(X, s(Y)).
:- tile1(X, Y), tile1(s(X), Y).
:- tile1(X, Y), tile1(X, s(Y)).
below0(Y) :- tile0(c0, s(Y)). % each tile in first
below0(Y) :- below0(s(Y)).    % column is below a
:- dom(Y), not below0(Y).     % tile of type 0
```

# Consistency is (very) hard, i.e. $\Sigma_1^1$-complete

**Hardness:** Reduction from “Recurring Tiling”

Given:  

Wanted:



with tile  infinitely often in first column.

RecurringTiling.asp

```
dom(c0).
dom(s(X)) :- dom(X).
tile0(X, Y) :- dom(X), dom(Y), not tile1(X, Y)
tile1(X, Y) :- dom(X), dom(Y), not tile0(X, Y)
:- tile0(X, Y), tile0(s(X), Y).
:- tile0(X, Y), tile0(X, s(Y)).
:- tile1(X, Y), tile1(s(X), Y).
:- tile1(X, Y), tile1(X, s(Y)).
below0(Y) :- tile0(c0, s(Y)). % each tile in first
below0(Y) :- below0(s(Y)).    % column is below a
:- dom(Y), not below0(Y).     % tile of type 0
```

*“Eventually Quantification” is typical for* $\Sigma_1^1$.

# Consistency is (very) hard, i.e. $\Sigma_1^1$-complete

**Membership:**
Reduction to NTM that admits a run that visits the start state infinitely many times iff the program is consistent.

# Consistency is (very) hard, i.e. $\Sigma_1^1$-complete

**Membership:**
Reduction to NTM that admits a run that visits the start state infinitely many times iff the program is consistent.

Rule Shape: $[H_r] \leftarrow B_1^+, \ldots, B_n^+, \neg B_1^-, \ldots, \neg B_m^-$

NTM-for-Consistency.pseudo; Input: Program $P$

1. Initialize empty set $L_0$ of literals, and counters $i := 0$ and $j := 0$.
2. If $L_i^+$ and $L_i^-$ are not disjoint, halt.
3. If $L_i^+$ is an answer set of P , loop on the start state.
4. Initialize $L_{i+1} := L_i \cup \{H_r\} \cup \{\neg a \mid a \in B_r^-\}$ where $r$ is some non-deterministically chosen rule in $\mathrm{Active}_{L_i^+}(P)$.
5. If $L_i$ satisfies all of the rules in $\mathrm{Active}_{L_j^+}(P)$, then set $j := j + 1$ and visit the start state once.
6. Set $i := i + 1$ and go to Step 2.

$\mathrm{Active}_I(P)$ is the set of ground rules that are unsatisfied in $I$.

# Two Characterizations of Programs

**Frugal:** Only finite answer sets. $\Pi^1_1$-complete. (Membership: Use `NTM-for-Consistency.pseudo` but halt instead of loop in step 3. Hardness: `RecurringTiling.asp` is frugal iff the tiling problem has no solution.)

# Two Characterizations of Programs

**Frugal:** Only finite answer sets. $\Pi^1_1$-complete. (Membership: Use `NTM-for-Consistency.pseudo` but halt instead of loop in step 3. Hardness: `RecurringTiling.asp` is frugal iff the tiling problem has no solution.)

**Non-proliferous:** Only finitely many finite answer sets (infinite ones allowed). $\Sigma^0_2$-complete.

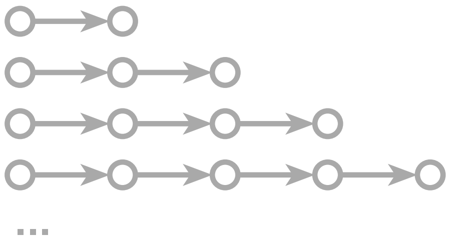
# Two Characterizations of Programs

**Frugal:** Only finite answer sets. $\Pi^1_1$-complete. (Membership: Use `NTM-for-Consistency.pseudo` but halt instead of loop in step 3. Hardness: `RecurringTiling.asp` is frugal iff the tiling problem has no solution.)

**Non-proliferous:** Only finitely many finite answer sets (infinite ones allowed). $\Sigma^0_2$-complete.

FrugalButProliferous.asp

```
next(c,d).
next(Y, f(Y)) :- next(X, Y), not last (Y).
last(Y) :- next(X, Y), not next(Y, f(Y)).
done :- last(Y).
:- not done.
```



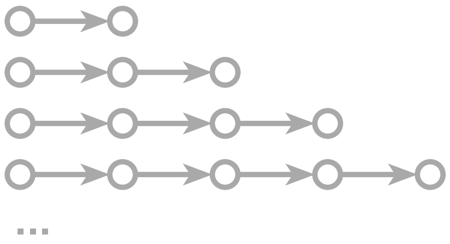
# Two Characterizations of Programs

**Frugal:** Only finite answer sets. $\Pi^1_1$-complete. (Membership: Use `NTM-for-Consistency.pseudo` but halt instead of loop in step 3. Hardness: `RecurringTiling.asp` is frugal iff the tiling problem has no solution.)

**Non-proliferous:** Only finitely many finite answer sets (infinite ones allowed). $\Sigma^0_2$-complete.

FrugalButProliferous.asp

```
next(c,d).
next(Y, f(Y)) :- next(X, Y), not last (Y).
last(Y) :- next(X, Y), not next(Y, f(Y)).
done :- last(Y).
:- not done.
```



*Even when frugal and non-proliferous, consistency is (only) semi-decidable.*

# Finite Groundings using Forbidden Atoms

- **Forbidden Atoms**: Not in any answer set
- Ignoring them during grounding yields a **finite grounding** for every frugal and non-proliferous programs.
- Checking if an atom is forbidden is **undecidable**.
- `redundant` is forbidden in first example.

GroundNotForbidden.pseudo; Output: $P_g$

1. Set $i := 1, A_0 := \emptyset, P_g := \emptyset$.
2. Set $A_i := A_{i-1}$. For each ground rule $r = H_r \leftarrow B_r^+, B_r^-$ with $B_r^+ \subseteq A_{i-1}$, (a) if $H_r$ is *forbidden* add $\leftarrow B_r^+, B_r^-$ to $P_g$, (b) otherwise add $r$ to $P_g$ and $H_r$ to $A_i$.
3. Stop if $A_i = A_{i-1}$; else inc $i$, go to 2.

# Detecting Forbidden Atoms: Toy Example

ToyExample.asp

```
r(a, b).
r(Y, f(Y)) :- r(X, Y), not stop(X).
stop(Y) :- r(X, Y).
```

*We have a sufficient check for forbidden atoms that combines backtracking of atom origins with "obvious" inferences.*

**Grounding:**

# Detecting Forbidden Atoms: Toy Example

ToyExample.asp

```
r(a, b).
r(Y, f(Y)) :- r(X, Y), not stop(X).
stop(Y) :- r(X, Y).
```

**Is `r(a,b)` forbidden?**

**Grounding:**

# Detecting Forbidden Atoms: Toy Example

ToyExample.asp

```
r(a, b).
r(Y, f(Y)) :- r(X, Y), not stop(X).
stop(Y) :- r(X, Y).
```

**Grounding:**

**Is `r(a,b)` forbidden?**

No, `r(a,b)` can only originate from the first line and this is fine.

*Since this is a sufficient check, “no” actually means “we do not know” or “we do not think so”.*

# Detecting Forbidden Atoms: Toy Example

ToyExample.asp

```
r(a, b).
r(Y, f(Y)) :- r(X, Y), not stop(X).
stop(Y) :- r(X, Y).
```

**Grounding:**

r(a,b)

# Detecting Forbidden Atoms: Toy Example

ToyExample.asp

```
r(a, b).
r(Y, f(Y)) :- r(X, Y), not stop(X).
stop(Y) :- r(X, Y).
```

**Is `stop(b)` forbidden?**

**Grounding:**

`r(a,b)`

# Detecting Forbidden Atoms: Toy Example

ToyExample.asp

```
r(a, b).
r(Y, f(Y)) :- r(X, Y), not stop(X).
stop(Y) :- r(X, Y).
```

**Grounding:**

`r(a,b)`

**Is `stop(b)` forbidden?**

No, `stop(b)` originates from the last rule and only requires `r(a,b)`.

# Detecting Forbidden Atoms: Toy Example

ToyExample.asp

```
r(a, b).
r(Y, f(Y)) :- r(X, Y), not stop(X).
stop(Y) :- r(X, Y).
```

**Grounding:**

`r(a,b)`

`stop(b)`

# Detecting Forbidden Atoms: Toy Example

ToyExample.asp

```
r(a, b).
r(Y, f(Y)) :- r(X, Y), not stop(X).
stop(Y) :- r(X, Y).
```

**Grounding:**

`r(a,b)`

`stop(b)`

**Is `r(b,f(b))` forbidden?**

# Detecting Forbidden Atoms: Toy Example

ToyExample.asp

```
r(a, b).
r(Y, f(Y)) :- r(X, Y), not stop(X).
stop(Y) :- r(X, Y).
```

**Grounding:**

`r(a,b)`

`stop(b)`

**Is `r(b,f(b))` forbidden?**

`r(b,f(b))` originates from the second rule and requires `r(a,b)` but also `not stop(a)`.

# Detecting Forbidden Atoms: Toy Example

ToyExample.asp

```
r(a, b).
r(Y, f(Y)) :- r(X, Y), not stop(X).
stop(Y) :- r(X, Y).
```

**Grounding:**

`r(a,b)`

`stop(b)`

**Is `r(b,f(b))` forbidden?**

`r(b,f(b))` originates from the second rule and requires `r(a,b)` but also `not stop(a)`.

`stop(a)` cannot be derived, so everything is fine.

# Detecting Forbidden Atoms: Toy Example

ToyExample.asp

```
r(a, b).
r(Y, f(Y)) :- r(X, Y), not stop(X).
stop(Y) :- r(X, Y).
```

**Grounding:**

`r(a,b)`

`stop(b)`

`r(b,f(b))`

# Detecting Forbidden Atoms: Toy Example

ToyExample.asp

```
r(a, b).
r(Y, f(Y)) :- r(X, Y), not stop(X).
stop(Y) :- r(X, Y).
```

**Grounding:**

`r(a,b)`

`stop(b)`

`r(b,f(b))`

`stop(f(b))`

# Detecting Forbidden Atoms: Toy Example

ToyExample.asp

```
r(a, b).
r(Y, f(Y)) :- r(X, Y), not stop(X).
stop(Y) :- r(X, Y).
```

**Is `r(f(b),f(f(b)))` forbidden?**

**Grounding:**

`r(a,b)`

`stop(b)`

`r(b,f(b))`

`stop(f(b))`

# Detecting Forbidden Atoms: Toy Example

ToyExample.asp

```
r(a, b).
r(Y, f(Y)) :- r(X, Y), not stop(X).
stop(Y) :- r(X, Y).
```

**Grounding:**

`r(a,b)`

`stop(b)`

`r(b,f(b))`

`stop(f(b))`

**Is `r(f(b),f(f(b)))` forbidden?**

`r(f(b),f(f(b)))` originates from the second rule and requires `r(b, f(b))` but also `not stop(b)`.

# Detecting Forbidden Atoms: Toy Example

ToyExample.asp

```
r(a, b).
r(Y, f(Y)) :- r(X, Y), not stop(X).
stop(Y) :- r(X, Y).
```

**Grounding:**

`r(a,b)`

`stop(b)`

`r(b,f(b))`

`stop(f(b))`

**Is `r(f(b),f(f(b)))` forbidden?**

`r(f(b),f(f(b)))` originates from the second rule and requires `r(b, f(b))` but also `not stop(b)`.

We already know that `r(b,f(b))` requires `r(a,b)`, which leads to the derivation of `stop(b)`.

# Detecting Forbidden Atoms: Toy Example

ToyExample.asp

```
r(a, b).
r(Y, f(Y)) :- r(X, Y), not stop(X).
stop(Y) :- r(X, Y).
```

**Grounding:**

`r(a,b)`

`stop(b)`

`r(b,f(b))`

`stop(f(b))`

**Is `r(f(b),f(f(b)))` forbidden?**

`r(f(b),f(f(b)))` originates from the second rule and requires `r(b, f(b))` but also `not stop(b)`.

We already know that `r(b,f(b))` requires `r(a,b)`, which leads to the derivation of `stop(b)`.

So `r(f(b),f(f(b)))` is forbidden.

# Thanks for bearing with me!

**What we did:**

- Introduce classes of *frugal* and *non-proliferous* programs.
- Study computability result for this classification.
- Propose grounding procedure that terminates in more cases for frugal and non-proliferous programs.

**What could be next:**

Implementation(!); tradeoff between generality and performance required.
Extension of results to rules with disjunctions should not be hard.

# References

[MNR94] V. W. Marek, A. Nerode, and J. B. Remmel, "The Stable Models of a Predicate Logic Program," *The Journal of Logic Programming*, vol. 21, no. 3, pp. 129–154, Nov. 1994, doi: 10.1016/S0743-1066(14)80008-3.

[Dan+01] E. Dantsin, T. Eiter, G. Gottlob, and A. Voronkov, "Complexity and expressive power of logic programming," *ACM Comput. Surv.*, vol. 33, no. 3, pp. 374–425, Sep. 2001, doi: 10.1145/502807.502810.

[Har86] D. Harel, "Effective transformations on infinite trees, with applications to high undecidability, dominoes, and fairness," *J. ACM*, vol. 33, no. 1, pp. 224–248, Jan. 1986, doi: 10.1145/4904.4993.