Finite Groundings for ASP with Functions

A Journey through Consistency

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

08.08.2024

How are Functions used in ASP?

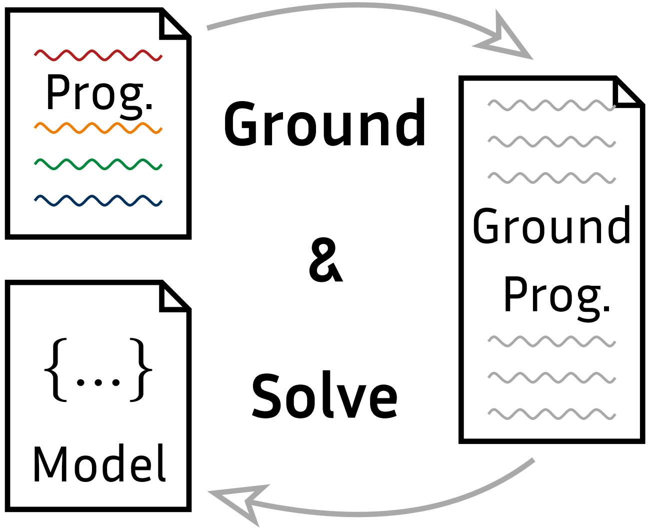
General Procedure:

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

How are Functions used in ASP?

General Procedure:

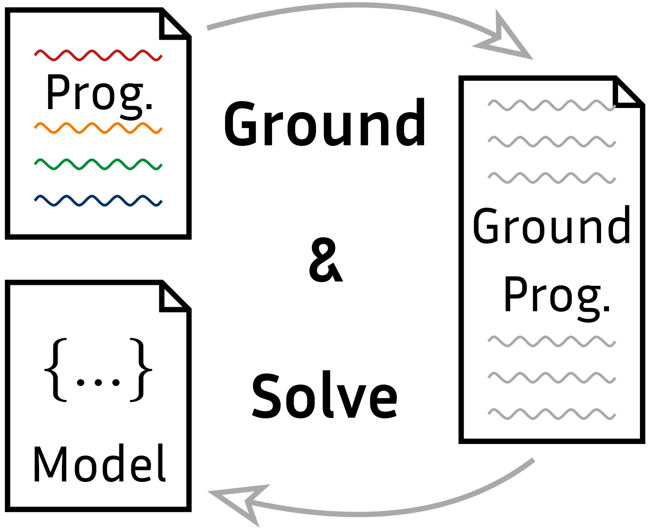
ASP systems like Clingo and (i)DLV work as follows.



How are Functions used in ASP?

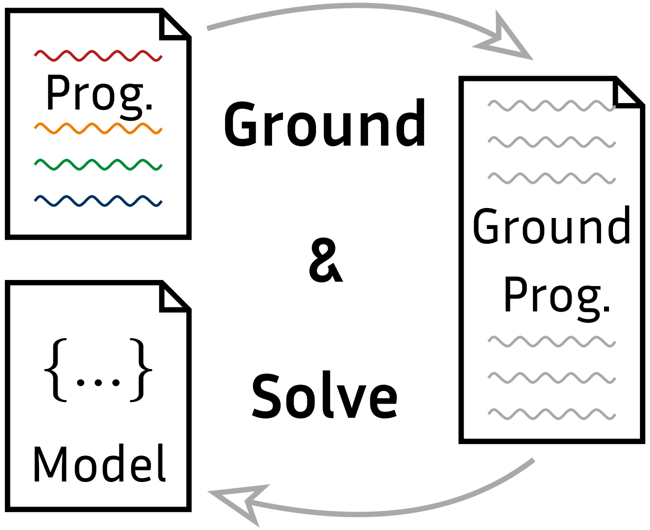
General Procedure:
ASP systems like Clingo and (i)DLV work as follows.

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.



Prog.

Ground

&

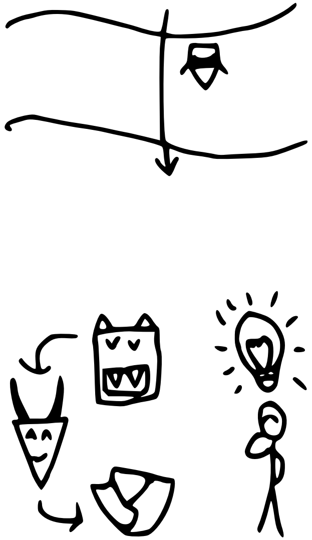
Solve

Ground Prog.

{…}

Model

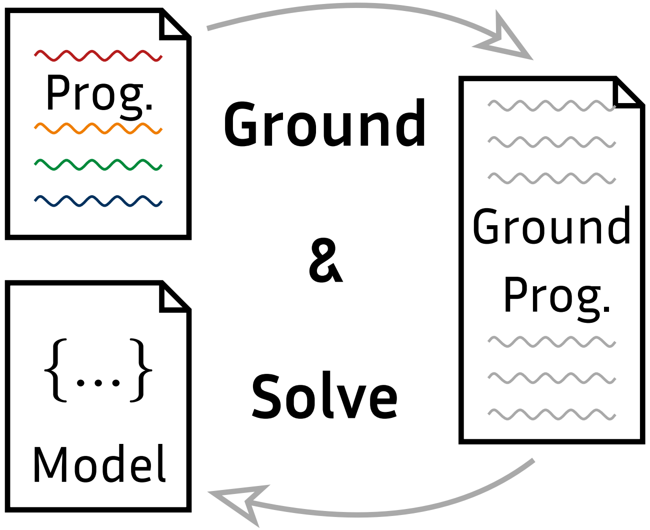
Example: Bring wolf, goat, and cabbage over river.



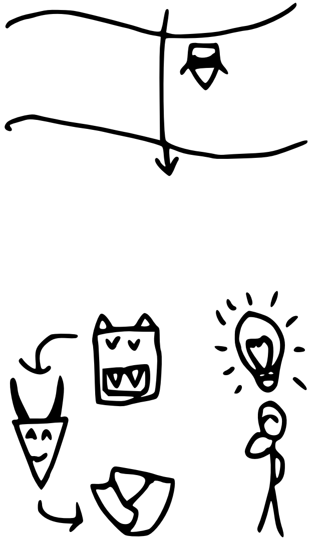
How are Functions used in ASP?

General Procedure:

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



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



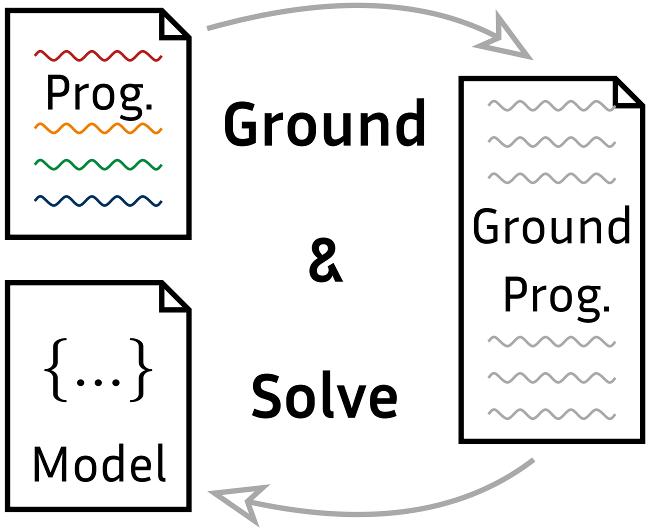
WolfGoatCabbage-GameRules.asp

```
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.


Prog.

Ground

&

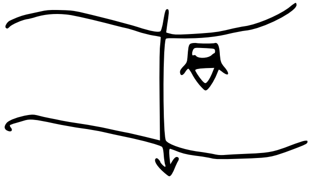
Solve

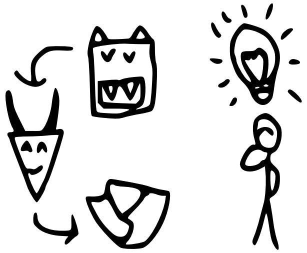
Ground Prog.

{...}

Model

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





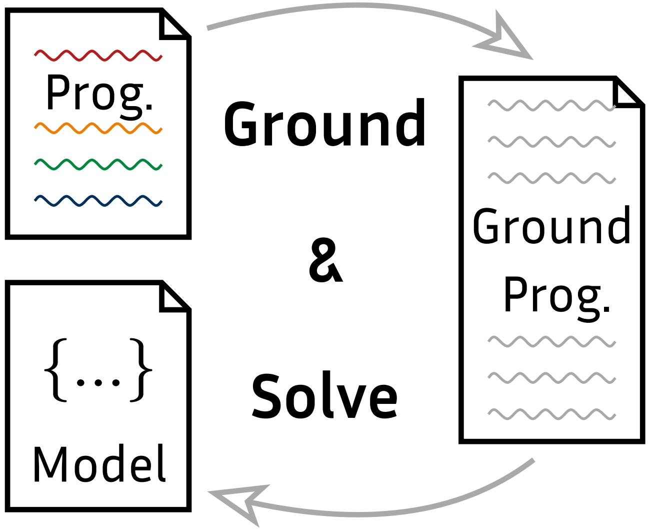
WolfGoatCabbage-ChooseMove.asp

```
% 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.

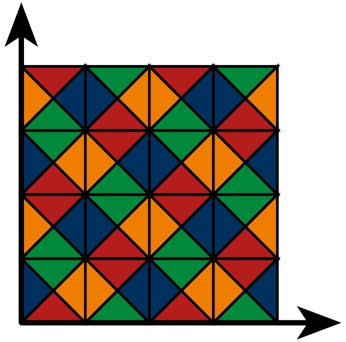
WolfGoatCabbage-UpdatePositions-LimitStepsAndRedundancies.asp

```
% 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).

% 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?

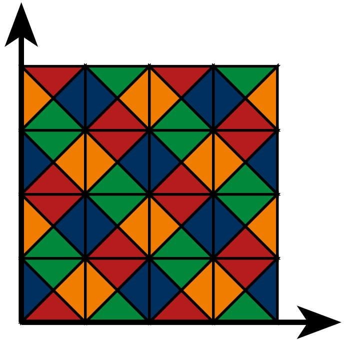
Understand:

- Consistency is Σ_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 Σ_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.

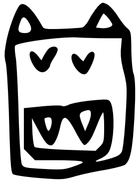
Overcome:

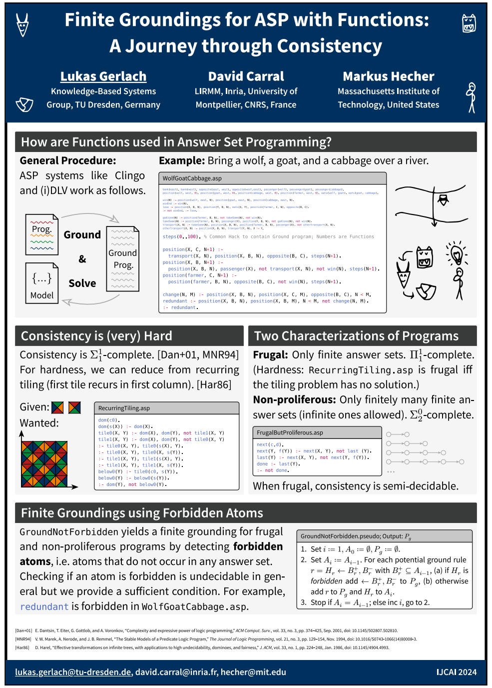
We propose `GroundNotForbidden` as a grounding procedure ignoring **forbidden atoms** that yields finite grounding for frugal and non-proliferous programs.

GroundNotForbidden.pseudo; Output: P_g

```
1. Set $i := 1, A_0 := \emptyset, P_g := \emptyset$.
2. Set $A_i := A_{i-1}$. For each potential 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.
```

Oops, time is over :(





Finite Groundings for ASP with Functions: A Journey through Consistency

Lukas Gerlach
Knowledge-Based Systems Group, TU Dresden, Germany

David Carral
LIRMM, Inria, University of Montpellier, CNRS, France

Markus Hecher
Massachusetts Institute of Technology, United States

How are Functions used in Answer Set Programming?

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

Prog.
Ground & Solve
Ground Prog.
{...}
Model

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

WolfGoatCabbage.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. :- lose.

goAlone(N) :- position(farmer, B, N), not takeSome(N), not win(N).
takeSome(N) :- position(farmer, B, N), passenger(Y), position(Y, B, N), not goAlone(N), not win(N).
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.

steps(0..100). % Common Hack to contain Ground program; Numbers are Functions

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).

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.
```

Consistency is (very) Hard

Consistency is Σ_1^1-complete. [Dan+01, MNR94] For hardness, we can reduce from recurring tiling (first tile recurs in first column). [Har86]

Given: 🟩 🟧

Wanted:

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)).
below0(Y) :- below0(s(Y)).
:- dom(Y), not below0(Y).
```

Two Characterizations of Programs

Frugal: Only finite answer sets. Π_1^1-complete. (Hardness: `RecurringTiling.asp` is frugal iff the tiling problem has no solution.)

Non-proliferous: Only finitely many finite answer sets (infinite ones allowed). Σ_2^0-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.
```

...

When frugal, consistency is semi-decidable.

Finite Groundings using Forbidden Atoms

`GroundNotForbidden` yields a finite grounding for frugal and non-proliferous programs by detecting **forbidden atoms**, i.e. atoms that do not occur in any answer set. Checking if an atom is forbidden is undecidable in general but we provide a sufficient condition. For example, `redundant` is forbidden in `WolfGoatCabbage.asp`.

GroundNotForbidden.pseudo; Output: P_g

```
1. Set $i := 1, A_0 := \emptyset, P_g := \emptyset$.
2. Set $A_i := A_{i-1}$. For each potential 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.
```

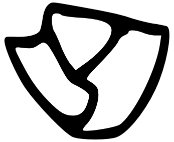
[Dan+01] E. Dantsin, T. Eiter, G. Gottlob, and A. Voronkov, "Complexity and expressive power of logic programming," *ACM Comput. Surv.*, vol. 33, no. 3, pp. 374–425, Sep. 2001, doi: 10.1145/502807.502810.

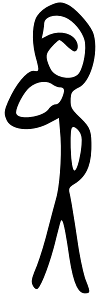
[MNR94] V. W. Marek, A. Nerode, and J. B. Remmel, "The Stable Models of a Predicate Logic Program," *The Journal of Logic Programming*, vol. 21, no. 3, pp. 129–154, Nov. 1994, doi: 10.1016/S0743-1066(14)80008-3.

[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.

lukas.gerlach@tu-dresden.de, david.carral@inria.fr, hecher@mit.edu

IJCAI 2024





We hope to discuss details at our poster with you :)

References

[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.

[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.

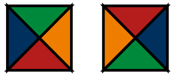
[Har86] D. Harel, “Effective transformations on infinite trees, with applications to high undecidability, dominoes, and fairness,” *J. ACM*, vol. 33, no. 1, pp. 224–248, Jan. 1986, doi: 10.1145/4904.4993.

Consistency is (very) hard, i.e. Σ_1^1-complete

Hardness: Reduction
from “Recurring Tiling”

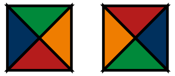
Consistency is (very) hard, i.e. Σ_1^1-complete

Hardness: Reduction
from "Recurring Tiling"

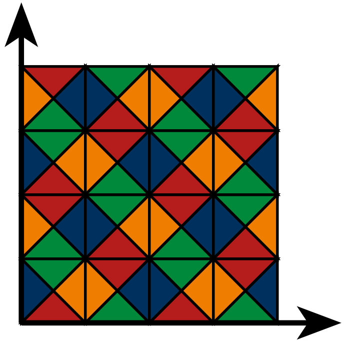
Given: 

Consistency is (very) hard, i.e. Σ_1^1-complete

Hardness: Reduction
from “Recurring Tiling”

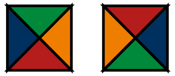
Given: 

Wanted:

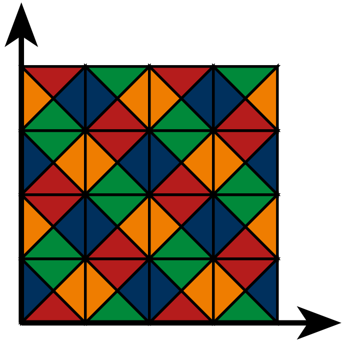


Consistency is (very) hard, i.e. Σ_1^1-complete

Hardness: Reduction from "Recurring Tiling"

Given: 

Wanted:



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. Σ_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. Σ_1^1-complete

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

$$H_r \leftarrow B_1^+, \ldots, B_n^+, \neg B_1^-, \ldots, \neg B_m^-$$

```
NTM-for-Consistency.pseudo; Input: Program $P$
1. Initialize an empty set $L_0$ of literals, and some 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. Π_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). Σ_2^0-complete.

Two Characterizations of Programs

Frugal: Only finite answer sets. Π^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). Σ^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.
```

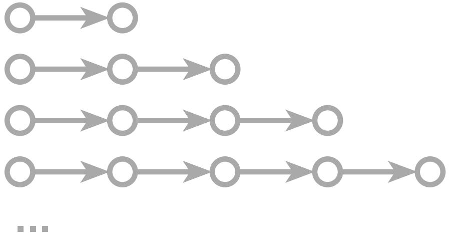
Two Characterizations of Programs

Frugal: Only finite answer sets. Π^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). Σ^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.
```



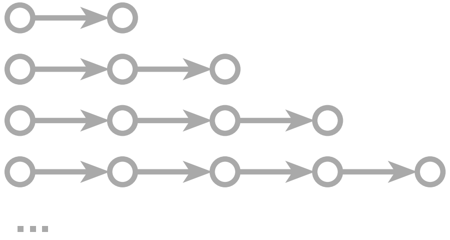
Two Characterizations of Programs

Frugal: Only finite answer sets. Π^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). Σ^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.
```



When frugal and non-proliferous, consistency is (only) semi-decidable.

Sketch for Σ_2^0-hardness of Non-Proliferous Check

Reduction from universal halting (Π_2^0-complete) for machine M.

1. Checking if a TM halts on infinitely many inputs is Π_2^0 hard. (Treat each input as natural number n and simulate M on all inputs of length n. The new machine halts on infinitely many inputs iff M universally halts.)
2. The complement (i.e. checking if a TM halts on only finitely many inputs) is Σ_2^0 hard.
3. We can generate finite inputs to a TM with an ASP program such that the program has a finite answer set for the input iff the TM halts on the input. That is, the program has finitely many finite answer sets iff the TM halts on finitely many inputs. (Generation uses idea from `FrugalButProliferous.asp` and is actually frugal.)

Finite Groundings using Forbidden Atoms

`GroundNotForbidden` yields finite grounding for frugal and non-proliferous programs by detecting **forbidden atoms**, i.e. atoms that do not occur in any answer set. Checking if an atom is forbidden is undecidable in general but we provide a sufficient condition. For example, `redundant` is forbidden in `WolfGoatCabbage.asp`.

GroundNotForbidden.pseudo; Output: P_g

```
1. Set $i := 1, A_0 := \emptyset, P_g := \emptyset$.
2. Set $A_i := A_{i-1}$. For each potential 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.
```