TECHNISCHE UNIVERSITÄT DRESDEN

Computational Logic Group

Hannes Strass

(based on slides by Michael Thielscher)

Faculty of Computer Science, Institute of Artificial Intelligence, Computational Logic Group

Negation: Proof Theory (SLDNF Resolution)

Lecture 7, 28th Nov 2022 // Foundations of Logic Programming, WS 2022/23

Previously ...

- Prolog employs SLD resolution with the **leftmost** selection rule, traverses the search space using **depth-first search** (including backtracking), and regards a program as a **sequence** of clauses.
- Prolog also offers list processing and arithmetics.
- The **cut** prunes certain branches of Prolog trees, and can lead to more efficient programs, but also to programming errors.

```
not(X) :- X, !, fail.
not(_).
% atom fail always fails
% not is also predefined in Prolog: :- op(900, fy, \+).
% not(X) is written as \+ X
```

Overview

Motivation: Why Negation?

Normal Logic Programs and Queries

SLDNF Resolution

Allowed Programs and Queries

Motivation: Why Negation?

Motivation: Example (1)

```
attends(andreas,fkr).
attends(maja,fkr).
attends(dirk,fkr).
attends(natalia,fkr).
attends(andreas,flp).
attends(maja,flp).
attends(stefan,flp).
attends(arturo,flp).
```

Who attends FLP but not FKR?

```
?- attends(X,flp), \+ attends(X,fkr).
```

Motivation: Example (2)

A list is a set $:\Longleftrightarrow$ there are no duplicates in it.

```
is_set([]).
is_set([H|T]) :- \+ member(H, T), is_set(T).
```

Motivation: Example (2)

A list is a set $:\Longleftrightarrow$ there are no duplicates in it.

```
is_set([]).
is_set([H|T]) :- \+ member(H, T), is_set(T).
```

The sets (lists) $A = [a_1, \ldots, a_m]$ and $B = [b_1, \ldots, b_n]$ are disjoint $:\Longleftrightarrow$

- $m = 0$, or
- $m > 0$, $a_1 \notin B$, and $[a_2, \ldots, a_m]$ and B are disjoint

```
disjoint([], _).
disjoint([X|Y], Z) :- \+ member(X, Z), disjoint(Y,Z).
```

Normal Logic Programs and Queries

Normal Logic Programs and Queries

Definition

- "$\sim$" (weak) **negation sign**
- $A, \sim A$ (weak) **literals** $:\Longleftrightarrow$ A atom
- $A, \sim A$ **ground literals** $:\Longleftrightarrow$ A ground atom
- **normal query** $:=$ finite sequence of (weak) literals
- $H \leftarrow \vec{B}$ **normal clause** $:\Longleftrightarrow$ H atom, $\vec{B}$ normal query
- **normal program** $:=$ finite set of normal clauses

- Everything as before, but now we are allowed to use (weak) negation in clause bodies (and queries).
- Negation "$\sim$" in $\sim A$ is "weak" because it does not state that A is false; it only states that A cannot be shown to be true from certain premises.
- In contrast, $\neg A$ states that A is false. More on this later in the course.

SLDNF Resolution

How Do We Compute?

Definition

The **Negation as Failure** (**nf**) rule is defined as follows:

1. Suppose $\sim A$ is selected in the query $Q = \vec{L}, \sim A, \vec{N}$.
2. If $P \cup \{A\}$ succeeds, then the derivation of $P \cup \{Q\}$ fails at this point.
3. If all derivations of $P \cup \{A\}$ fail, then Q resolves to $Q' = \vec{L}, \vec{N}$.

Thus:

$\sim A$ succeeds iff A finitely fails.

$\sim A$ finitely fails iff A succeeds.

Note

SLDNF = Selection rule driven Linear resolution for Definite clauses augmented by the Negation as Failure rule

SLDNF Resolvents

Definition

Let $Q = \vec{L}, K, \vec{N}$ be a query and K its selected literal.

1. $K = A$ is an atom:
   - $H \leftarrow \vec{M}$ is a variant of a clause c that is variable-disjoint with Q
   - θ is an mgu of A and H
   - $Q' = (\vec{L}, \vec{M}, \vec{N})\theta$ is the **SLDNF resolvent** of Q (and c w.r.t. A with θ)
   - We write this **SLDNF derivation step** as $Q \xrightarrow[c]{\theta} Q'$.
2. $K = {\sim}A$ is a negative ground literal:
   - $Q' = \vec{L}, \vec{N}$ **SLDNF resolvent** of Q (w.r.t. ${\sim}A$ with ε)
   - We write this **SLDNF derivation step** as $Q \xrightarrow{\varepsilon} Q'$.

Pseudo Derivations

Definition

A maximal sequence of SLDNF derivation steps

$$Q_0 \xrightarrow[c_1]{\theta_1} Q_1 \cdots Q_n \xrightarrow[c_{n+1}]{\theta_{n+1}} Q_{n+1} \cdots$$

is a **pseudo derivation of** $P \cup \{Q_0\}$ $:\Longleftrightarrow$

- $Q_0, \ldots, Q_{n+1}, \ldots$ are queries, each empty or with one literal selected in it;
- $\theta_1, \ldots, \theta_{n+1}, \ldots$ are substitutions;
- $c_1, \ldots, c_{n+1}, \ldots$ are clauses of program P
  (in case a positive literal is selected in the preceding query);
- for every SLDNF derivation step with input clause the condition **standardization apart** holds.

Forests

Definition

A triple $\mathcal{F} = (\mathcal{T}, T, subs)$ is a **forest** $:\Longleftrightarrow$

- $\mathcal{T}$ set of trees where
  - nodes are queries;
  - a literal is selected in each non-empty query;
  - leaves may be marked as "success", "failure", or "floundered".
- $T \in \mathcal{T}$ is the **main** tree
- *subs* assigns to some nodes of trees in $\mathcal{T}$ with selected negative ground literal $\sim A$ a **subsidiary** tree of $\mathcal{T}$ with root A.

Definition

Let $T \in \mathcal{T}$ be a tree.

- T is **successful** $:\Longleftrightarrow$ it contains a leaf marked as "success".
- T is **finitely failed** $:\Longleftrightarrow$ it is finite and all leaves are marked as "failure".

Pre-SLDNF Trees and their Extensions

Definition

The class of **pre-SLDNF trees** for a program P is the smallest class $\mathcal{C}$ of forests such that

- for every query Q: the **initial pre-SLDNF tree** $(\{T_Q\}, T_Q, subs)$ is in $\mathcal{C}$, where T_Q contains the single node Q and $subs(Q)$ is undefined;
- for every $\mathcal{F} \in \mathcal{C}$: the **extension** of $\mathcal{F}$ is in $\mathcal{C}$.

Definition

The **extension** of $\mathcal{F} = (\mathcal{T}, T, subs)$ is the forest that is obtained as follows:

1. Every occurrence of the empty query is marked as "success."
2. For every non-empty query Q that is an unmarked leaf in some tree in $\mathcal{T}$, perform the action *extend*$(\mathcal{F}, Q, L)$, where L is the selected literal of Q.

Action $extend(\mathcal{F}, Q, L)$

Recall that L is the selected literal of Q.

Definition

- L is positive. Then $extend(\mathcal{F}, Q, L)$ is obtained as follows:
  - Q has no SLDNF resolvents $\Rightarrow$ Q is marked as "failure"
  - else $\Rightarrow$ for every program clause c which is applicable to L, exactly one direct descendant of Q is added. This descendant is an SLDNF resolvent of Q and c w.r.t. L.
- $L = {\sim}A$ is negative. Then $extend(\mathcal{F}, Q, L)$ is obtained as follows:
  - A non-ground $\Rightarrow$ Q is marked as "floundered"
  - A ground: case distinction on Q:
    - $subs(Q)$ undefined
      $\Rightarrow$ new tree T' with single node A is added to $\mathcal{T}$ and $subs(Q)$ is set to T'
    - $subs(Q)$ defined and successful $\Rightarrow$ Q is marked as "failure"
    - $subs(Q)$ defined and finitely failed
      $\Rightarrow$ SLDNF resolvent of Q is added as the only direct descendant of Q
    - $subs(Q)$ defined and neither successful nor finitely failed $\Rightarrow$ no action

SLDNF Trees (Successful, Failed, Finite)

Definition

An **SLDNF tree** is the limit of a sequence $\mathcal{F}_0, \mathcal{F}_1, \mathcal{F}_2 \ldots$, where

- $\mathcal{F}_0$ is an initial pre-SLDNF tree;
- $\mathcal{F}_{i+1}$ is the extension of $\mathcal{F}_i$, for every $i \in \mathbf{N}$.

The SLDNF tree **for** $P \cup \{Q\}$ is the SLDNF tree in which Q is the root of the main tree.

Definition

- A (pre-)SLDNF tree is **successful** $:\Longleftrightarrow$ its main tree is successful.
- A (pre-)SLDNF tree **finitely failed** $:\Longleftrightarrow$ its main tree is finitely failed.
- An SLDNF tree is **finite** $:\Longleftrightarrow$ no infinite paths exist in it, where a **path** is a sequence of nodes $N_0, N_1, N_2, \ldots$ such that for every $i = 0, 1, 2, \ldots$:
  - either N_{i+1} is a direct descendant of N_i,
  - or N_{i+1} is the root of $subs(N_i)$.

Example (1)

Consider the following logic program P:

$$p \leftarrow p$$

The SLDNF tree for $P \cup \{\sim p\}$ is infinite:

$$\underline{\sim p}$$

Example (1)

$$p \leftarrow p$$

Consider the following logic program P:

The SLDNF tree for $P \cup \{\sim p\}$ is infinite:

$$\underline{\sim p}$$

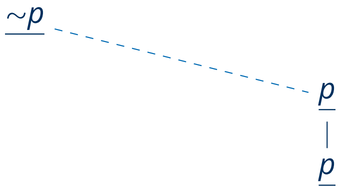
$$\underline{p}$$

Example (1)

Consider the following logic program P:

$$p \leftarrow p$$

The SLDNF tree for $P \cup \{\sim p\}$ is infinite:



Example (1)

$p \leftarrow p$

Consider the following logic program P:

The SLDNF tree for $P \cup \{\sim p\}$ is infinite:

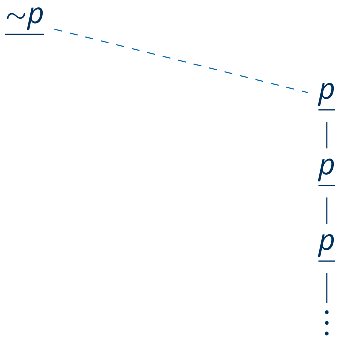
$$\underline{\sim p} \quad \cdots \quad \begin{array}{c} \underline{p} \\ | \\ \underline{p} \\ | \\ \underline{p} \end{array}$$

Example (1)

Consider the following logic program P:

$$p \leftarrow p$$

The SLDNF tree for $P \cup \{\sim p\}$ is infinite:



Example (2)

$$
\begin{array}{lll}
p & \leftarrow & \sim q \\
q & \leftarrow & \\
q & \leftarrow & q
\end{array}
$$

Consider the following logic program P:

The SLDNF tree for $P \cup \{\sim p\}$ is successful:

$$\underline{\sim p}$$

Example (2)

$$
\begin{aligned}
p &\leftarrow \sim q \\
q &\leftarrow \\
q &\leftarrow q
\end{aligned}
$$

Consider the following logic program P:

The SLDNF tree for $P \cup \{\sim p\}$ is successful:

$\underline{\sim p}$

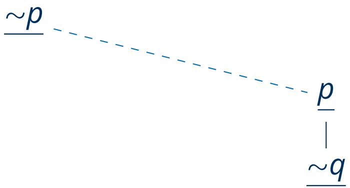
$\underline{p}$

Example (2)

$$\begin{array}{lcl} p & \leftarrow & \sim q \\ q & \leftarrow & \\ q & \leftarrow & q \end{array}$$

Consider the following logic program P:

The SLDNF tree for $P \cup \{\sim p\}$ is successful:

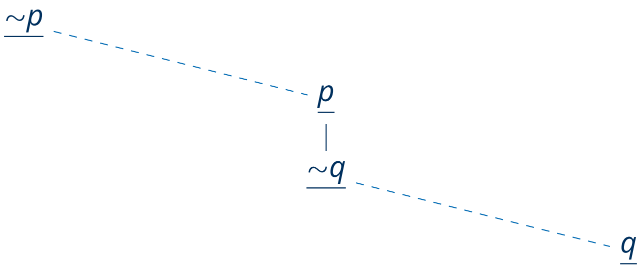


Example (2)

$$\begin{array}{lll} p & \leftarrow & \sim q \\ q & \leftarrow & \\ q & \leftarrow & q \end{array}$$

Consider the following logic program P:

The SLDNF tree for $P \cup \{\sim p\}$ is successful:

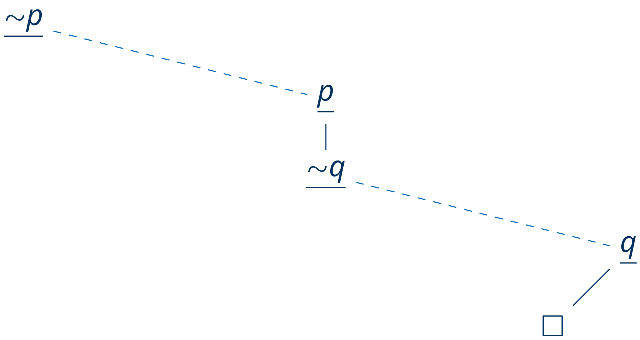


Example (2)

Consider the following logic program P:

$$
\begin{array}{lll}
p & \leftarrow & \sim q \\
q & \leftarrow & \\
q & \leftarrow & q
\end{array}
$$

The SLDNF tree for $P \cup \{\sim p\}$ is successful:

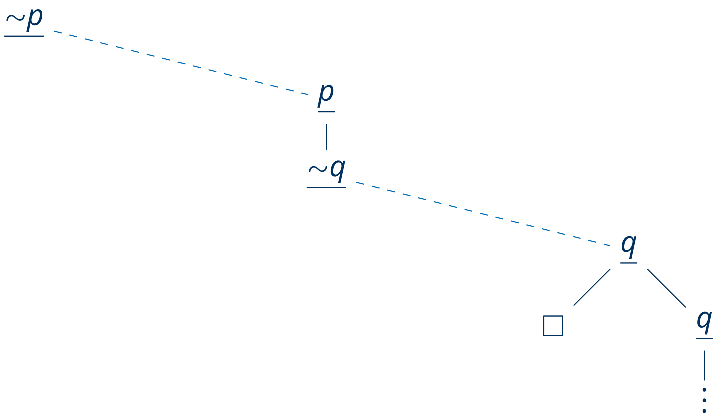


Example (2)

$$\begin{array}{lcl} p & \leftarrow & \sim q \\ q & \leftarrow & \\ q & \leftarrow & q \end{array}$$

Consider the following logic program P:

The SLDNF tree for $P \cup \{\sim p\}$ is successful:

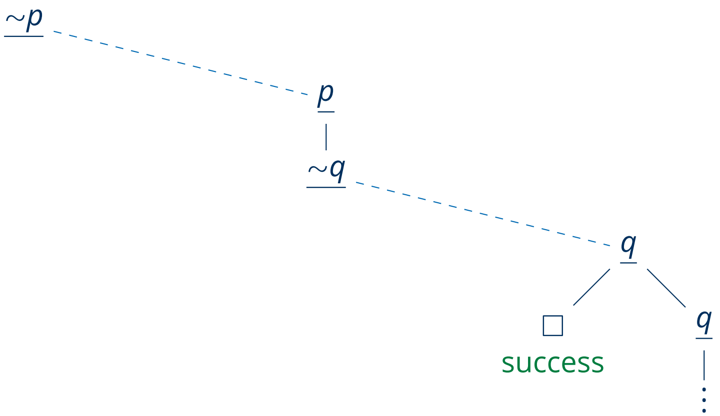


Example (2)

$$
\begin{aligned}
p &\leftarrow \sim q \\
q &\leftarrow \\
q &\leftarrow q
\end{aligned}
$$

Consider the following logic program P:

The SLDNF tree for $P \cup \{\sim p\}$ is successful:

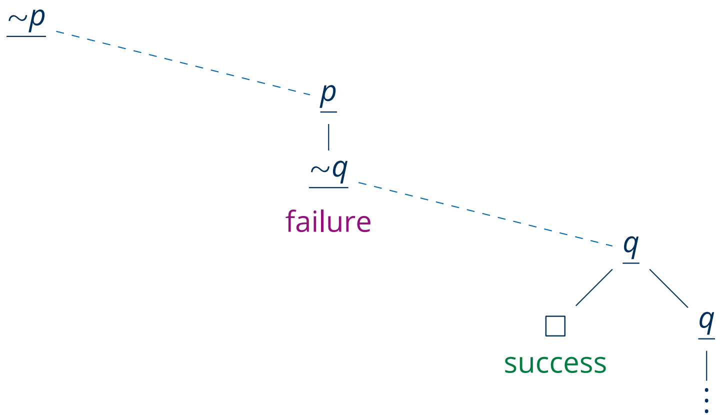


Example (2)

$$
\begin{aligned}
p &\leftarrow \sim q \\
q &\leftarrow \\
q &\leftarrow q
\end{aligned}
$$

Consider the following logic program P:

The SLDNF tree for $P \cup \{\sim p\}$ is successful:

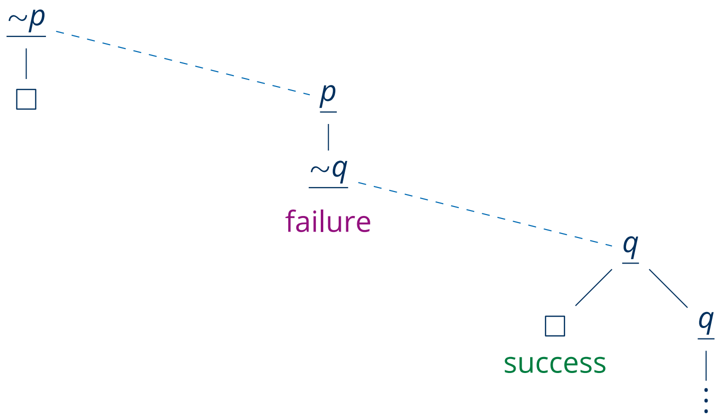


Example (2)

Consider the following logic program P:

$$\begin{array}{lll} p & \leftarrow & \sim q \\ q & \leftarrow & \\ q & \leftarrow & q \end{array}$$

The SLDNF tree for $P \cup \{\sim p\}$ is successful:

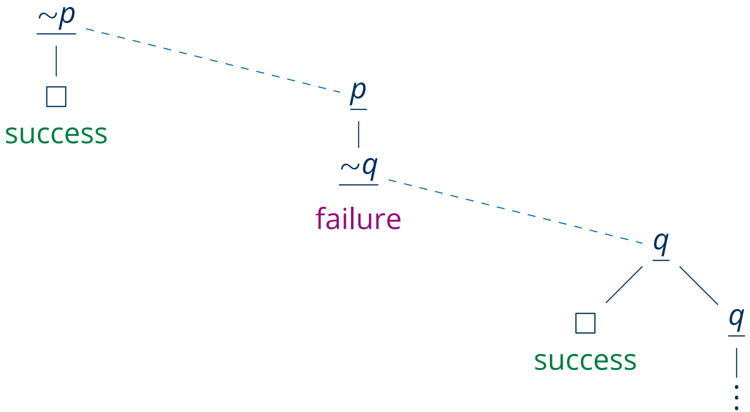


Example (2)

$$\begin{aligned} p &\leftarrow \sim q \\ q &\leftarrow \\ q &\leftarrow q \end{aligned}$$

Consider the following logic program P:

The SLDNF tree for $P \cup \{\sim p\}$ is successful:



Quiz: SLDNF Trees

Quiz

Consider the following logic program P: …

SLDNF derivation

Definition

An **SLDNF derivation** of $P \cup \{Q\}$ is

- a branch in the main tree of an SLDNF tree $\mathcal{F}$ for $P \cup \{Q\}$
- together with the set of all trees in $\mathcal{F}$ whose roots can be reached from the nodes in this branch.

An SLDNF derivation is **successful** $:\Longleftrightarrow$ the branch ends with $\Box$.

Definition

Let the main tree of an SLDNF tree for $P \cup \{Q_0\}$ contain a branch

$$\xi = Q_0 \xrightarrow{\theta_1} Q_1 \cdots Q_{n-1} \xrightarrow{\theta_n} Q_n = \Box$$

The **computed answer substitution** (**cas**) of Q_0 (w.r.t. ξ) is $(\theta_1 \cdots \theta_n)|_{Var(Q_0)}$.

A Theorem on Limits

Theorem 3.10 [Apt and Bol, 1994]

(i) Every SLDNF tree is the limit of a unique sequence of pre-SLDNF trees.

(ii) If the SLDNF tree $\mathcal{F}$ is the limit of the sequence $\mathcal{F}_0, \mathcal{F}_1, \mathcal{F}_2, \ldots$, then:

(a) $\mathcal{F}$ is successful and yields cas θ iff some $\mathcal{F}_i$ is successful and yields cas θ,

(b) $\mathcal{F}$ finitely failed iff some $\mathcal{F}_i$ is finitely failed.

Allowed Programs and Queries

Why Only Select *Ground* Negative Literals? (1)

$$zero(0) \leftarrow$$
$$positive(x) \leftarrow \ \sim zero(x)$$

$\underline{positive(y)}$

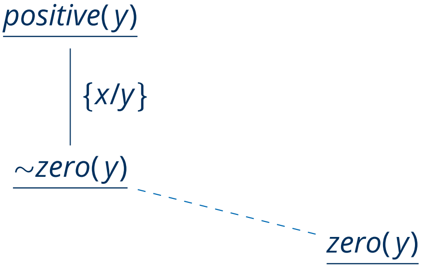
Why Only Select *Ground* Negative Literals? (1)

$$zero(0) \leftarrow$$
$$positive(x) \leftarrow \; {\sim}zero(x)$$

$$\underline{positive(y)}$$
$$\Big| \; \{x/y\}$$
$$\underline{{\sim}zero(y)}$$

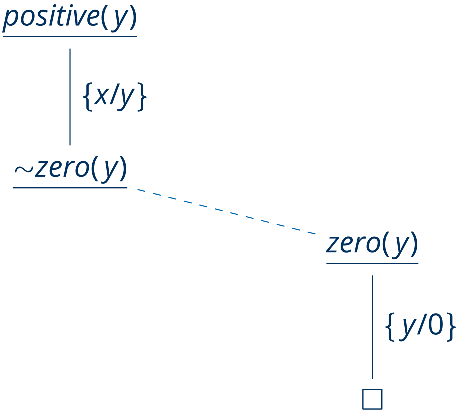
Why Only Select *Ground* Negative Literals? (1)

$$zero(0) \leftarrow$$
$$positive(x) \leftarrow \ \sim zero(x)$$



Why Only Select *Ground* Negative Literals? (1)

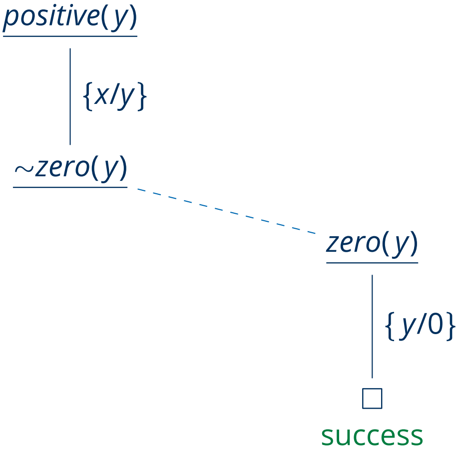
$$zero(0) \leftarrow$$
$$positive(x) \leftarrow \ \sim zero(x)$$



Why Only Select *Ground* Negative Literals? (1)

$zero(0) \leftarrow$

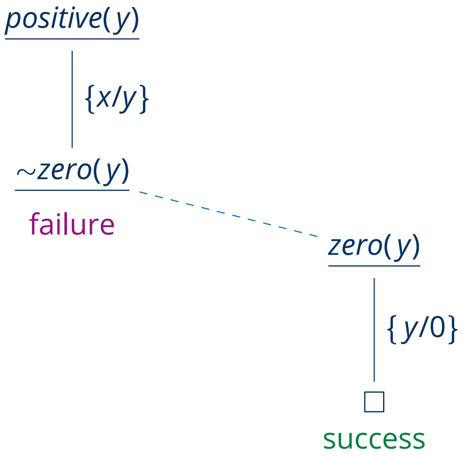
$positive(x) \leftarrow\ \sim zero(x)$



Why Only Select *Ground* Negative Literals? (1)

$$zero(0) \leftarrow$$

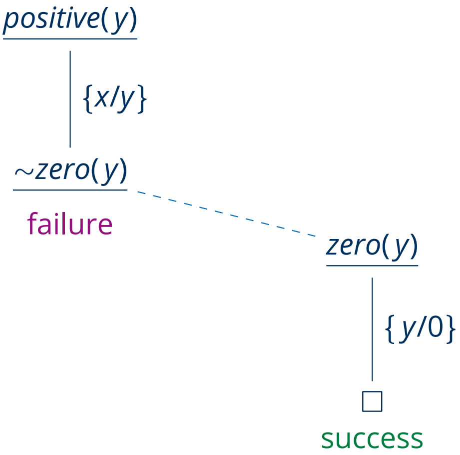
$$positive(x) \leftarrow \; \sim zero(x)$$



Why Only Select *Ground* Negative Literals? (1)

$zero(0) \leftarrow$

$positive(x) \leftarrow \ \sim zero(x)$



Hence, $\neg\exists y(positive(y))$? That is, $\forall y(\neg positive(y))$?

Why Only Select *Ground* Negative Literals? (2)

$$zero(0) \leftarrow$$
$$positive(x) \leftarrow \sim zero(x)$$

$\underline{positive(s(0))}$

Why Only Select *Ground* Negative Literals? (2)

$$zero(0) \leftarrow$$

$$positive(x) \leftarrow \ \sim zero(x)$$

$$\underline{positive(s(0))}$$

$$\Big| \ \{x/s(0)\}$$

$$\underline{\sim zero(s(0))}$$

Why Only Select *Ground* Negative Literals? (2)

$zero(0) \leftarrow$

$positive(x) \leftarrow \sim zero(x)$

$\underline{positive(s(0))}$

$\{x/s(0)\}$

$\underline{\sim zero(s(0))}$

$\underline{zero(s(0))}$

Why Only Select *Ground* Negative Literals? (2)

$zero(0) \leftarrow$

$positive(x) \leftarrow \;\sim zero(x)$

$\underline{positive(s(0))}$

$\{x/s(0)\}$

$\underline{\sim zero(s(0))}$

$\underline{zero(s(0))}$

failure

Why Only Select *Ground* Negative Literals? (2)

$$zero(0) \leftarrow$$
$$positive(x) \leftarrow \ \sim zero(x)$$

$\underline{positive(s(0))}$

$\{x/s(0)\}$

$\underline{\sim zero(s(0))}$ - - - $\underline{zero(s(0))}$

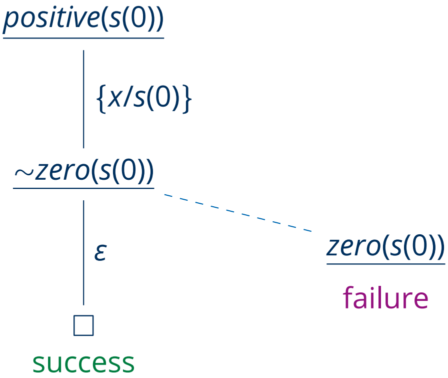
ε failure

$\square$

Why Only Select *Ground* Negative Literals? (2)

$zero(0) \leftarrow$

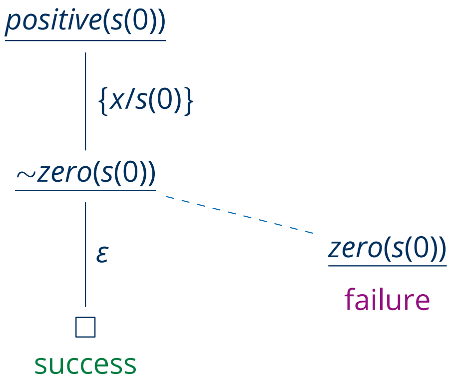
$positive(x) \leftarrow\ \sim zero(x)$



Why Only Select *Ground* Negative Literals? (2)

$zero(0) \leftarrow$

$positive(x) \leftarrow \ \sim zero(x)$

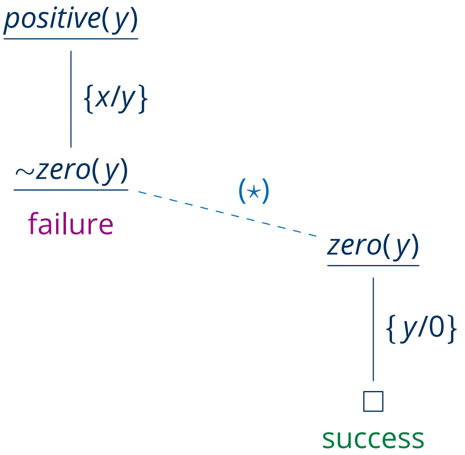


Hence, $positive(s(0))$**. That is,** $\exists y(positive(y))$**.**

Why Only Select *Ground* Negative Literals? (3)

$zero(0) \leftarrow$

$positive(x) \leftarrow \ \sim zero(x)$



Mistake in $(\star)$**:** $\exists y(zero(y)) \not\equiv \neg\exists y(\neg zero(y))$

Why Only Select *Ground* Negative Literals? (3)

$zero(0) \leftarrow$

$positive(x) \leftarrow\ \sim zero(x)$

$\underline{\exists y(positive(y))}$

$\{x/y\}$

$\underline{\exists y(\sim zero(y))}$

failure

$(\star)$

$\underline{\exists y(zero(y))}$

$\{y/0\}$

$\square$

success

Mistake in $(\star)$**:** $\exists y(zero(y)) \not\equiv \neg\exists y(\neg zero(y))$

Non-Ground Negative Literals in Prolog

```
zero(0).
positive(X) :- \+ zero(X).

| ?- positive(0).
no

| ?- positive(s(0)).
yes

| ?- positive(Y).
no
```

SLDNF Selection Rules & Blocked Queries

Definition

- An **SLDNF selection rule** is a function that, given a pre-SLDNF tree $\mathcal{F} = (\mathcal{T}, T, subs)$, selects a literal in every non-empty unmarked leaf in every tree in $\mathcal{T}$.
- An SLDNF tree $\mathcal{F}$ is **via** a selection rule $\mathcal{R}$ $:\Longleftrightarrow$ $\mathcal{F}$ is the limit of a sequence of pre-SLDNF trees in which literals are selected according to $\mathcal{R}$.
- A selection rule $\mathcal{R}$ is **safe** $:\Longleftrightarrow$ $\mathcal{R}$ never selects a non-ground negative literal.

Definition

- A query Q is **blocked** $:\Longleftrightarrow$ Q is non-empty and contains exclusively non-ground negative literals.
- $P \cup \{Q\}$ **flounders** $:\Longleftrightarrow$ some SLDNF tree for $P \cup \{Q\}$ contains a blocked node.

Allowed Programs and Queries

Definition

- A query Q is **allowed** $:\Longleftrightarrow$ every variable in Q occurs in a positive literal of Q.
- A clause $H \leftarrow \vec{B}$ is **allowed** $:\Longleftrightarrow$ the query $\sim H, \vec{B}$ is allowed.
  (Thus: A unit clause $H \leftarrow$ is **allowed** $:\Longleftrightarrow$ H is a ground atom.)
- A program P is **allowed** $:\Longleftrightarrow$ all its clauses are allowed.

Allowed clauses are also called *safe* (whenever no confusion with selection rules can arise).

Theorem 3.13 [Apt and Bol, 1994]

Suppose that P and Q are allowed. Then

(i) $P \cup \{Q\}$ does not flounder;

(ii) if θ is a cas of Q, then $Q\theta$ is ground.

Allowed Programs: Example

$$zero(0) \leftarrow$$
$$positive(x) \leftarrow \sim zero(x)$$

This program is not allowed.

$$zero(0) \leftarrow$$
$$positive(x) \leftarrow num(x), \sim zero(x)$$
$$num(0) \leftarrow$$
$$num(s(x)) \leftarrow num(x)$$

This program is allowed.

Conclusion

Summary

- **Normal logic programs** allow for negation in queries (clause bodies).
- A proof theory for normal logic programs is given by **SLDNF resolution**.
- Negated atoms $\sim A$ are treated by asking the query A in a **subsidiary tree**.
- Care must be taken not to let **non-ground negative literals** get selected.

Suggested action points:

- Construct the (leftmost selection rule) SLDNF tree for $positive(y)$ with the allowed version of the program.
- Find examples for programs and queries with blocked nodes in some SLDNF tree.