Procedural Semantics has 2 indeterminisms:

Indeterminism 1: Which program clause \( K \) is used for the next res. step?

Indeterminism 2: Which \( A_i \) in the current goal is used for the next resolution step?

For one configuration \((B_n, \sigma_n)\) there can be several successor configurations with \((B_n, \sigma_n) \Gamma \ (B_2, \sigma_2)\).

Ex. 43.1 Query: \( ? - \) ancestor(2, aline).

\[
(\{\neg \text{anc}(2, \text{aline})\}, \emptyset) \Gamma (\{\neg \text{mo}(2, \text{aline})\}, \{V/2, X/\text{aline}\})
\]

\[
(\{\neg \text{anc}(2, \text{aline})\}, \emptyset) \Gamma (\{\neg \text{mo}(2, Y), \neg \text{anc}(Y, \text{aline})\}, \{V/2, X/\text{aline}\})
\]

Indet 1 influences the solution:
Indet 2 influences the termination

To implement LP (on a deterministic computer), one has to resolve these 2 indeterminisms.

We first look at indeterminism 2.

It will turn out that this indet. is "harmless": it does not influence the solution, i.e., if one
resolves this indeterminism (e.g., by only taking the leftmost literal), then one still finds all solutions to the query.

Main reason: Exchange Lemma:

For a query \{\neg A_1, \ldots, \neg A_n\}, it does not matter whether one first resolves with \neg A_i and then with \neg A_j or vice versa.

**Lemma 432 (Exchange Lemma)**

Let \{\neg A_1, \ldots, \neg A_n\}, \{B, \neg C_1, \ldots, \neg C_m\}, \{D, \neg E_1, \ldots, \neg E_p\} be variable-disjoint Horn clauses. Let \sigma_n be the mgu of \neg A_i and B, let \sigma_2 be the mgu of \sigma_n (A_j) and D. Then the 2 resolution steps on the slide are possible (first resolve with \neg A_i, then with \neg A_j).

Then there exists an mgu \sigma_n' of A_j and D, and an mgu \sigma_2' of \sigma_n (A_i) and B. So it is also possible to resolve with A_j first and then with A_i.

Then \sigma_2' \circ \sigma_n' and \sigma_2' \circ \sigma_n are identical up to variable renaming, i.e., there is a variable renaming \psi so that \sigma_2' \circ \sigma_n' = \psi \circ \sigma_2' \circ \sigma_n.

**Ex 433** Illustration of the exchange lemma:

\[ p(z, z) : \neg r(z). \]

\[ \neg q(w). \]
\(?- p(X,Y), q(X).
\)

\(\{\neg p(X,Y), \neg q(X), \emptyset\} \vdash_{\mathcal{P}} \{\neg q(Z), \neg r(Z)\}, \{X/2, Y/Z\}\)

\(\vdash_{\mathcal{P}} \{\neg r(Z)\}, \{W/Z\} \circ \{X/2, Y/Z\}\)

\(\vdash_{\mathcal{P}} \{\neg r(Z)\}, \{W/Z\} \circ \{X/2, Y/Z, W/Z\}\)

Exchanging lemma states that one could exchange these 2 resolution steps (i.e., first resolve on \(q\), then on \(p\)). Then we get the same substitution up to variable renaming.

\(\{\neg p(X,Y), \neg q(X), \emptyset\} \vdash_{\mathcal{P}} \{\neg r(W,Y)\}, \{X/W,Y\}\)

\(\vdash_{\mathcal{P}} \{\neg r(Y)\}, \{W/Y, Z/Y\} \circ \{X/W\}\)

\(\vdash_{\mathcal{P}} \{\neg r(Y)\}, \{W/Y, Z/Y, W/Y, Z/Y\}\)

The resulting substitutions can be made equal by applying the variable renaming \(\mathcal{D} = \{Y/Z, Z/Y\}\).

**Proof of the exchangement lemma 4.3.2.**

Since the clauses are variable-disjoint, the mgu \(\sigma_1\) of \(A_i\) and \(B\) does not modify the variables in \(\mathcal{D}\), i.e.

\(\sigma_1(\mathcal{D}) = \mathcal{D}\).

\(\sigma_2\) is the mgu of \(\sigma_1(A_j)\) and \(\mathcal{D}\)

\(\Rightarrow \sigma_2 \circ \sigma_1 (A_j) = \sigma_2 \circ \sigma_1 (D)\)

\(\Rightarrow \sigma_2 \circ \sigma_1\) is a unifier of \(A_j\) and \(\mathcal{D}\).

\(\Rightarrow A_j\) and \(\mathcal{D}\) have an mgu \(\sigma_1\) and there exists a
Substitution $\sigma$ such that

$$\sigma_2 \circ \sigma_1 = \sigma \circ \sigma_1' \quad \text{(A)}$$

So we can perform the first resolution step using the mgu $\sigma_1'$. Now we have to show that one can also perform the second res. step, i.e., that $\sigma_1'(A_i)$ and $B$ are unifiable.

This indeed holds, since $\sigma$ is a unifier of $\sigma_1'(A_i)$ and $B$:

$$\sigma(\sigma_1'(A_i)) = \sigma_2(\sigma_1(A_i)) \quad \text{by (A)}$$

$$= \sigma_2(\sigma_1(B)) \quad \text{since } \sigma \text{ unifies } A_i \text{ and } B$$

$$= \sigma(\sigma_1'(B)) \quad \text{by (A)}$$

$$= \sigma(B) \quad \text{since } \sigma_1' \text{ does not modify the variables of } B \text{ (} \sigma_1' \text{ is mgu of } A_j \text{ and } B)$$

Since $\sigma$ is a unifier of $\sigma_1'(A_i)$ and $B$, they also have an mgu $\sigma_2'$. Thus, there exists a substitution $\sigma$ with

$$\sigma = \sigma \circ \sigma_2' \quad \text{(A A)}$$

Hence, one can exchange the resolution steps and first perform resolution on $\neg A_j$, then on $\neg A_i$.

We still have to show that $\sigma_2 \circ \sigma_1$ and $\sigma_2' \circ \sigma_1'$ are the same up to variable renaming.

To show this: Prove that $\sigma_2' \circ \sigma_1'$ is an instance of $\sigma_2 \circ \sigma_1$ and $\sigma_2 \circ \sigma_1'$ is an instance of $\sigma_2' \circ \sigma_1$.

Thus, $\sigma_2 \circ \sigma_1 = \sigma \circ \sigma_2' \circ \sigma_1'$ holds.
Reason: \( \sigma_2 \circ \sigma_n = \sigma \circ \sigma_n' \) by (4)
\[= \sigma \circ \sigma_2' \circ \sigma_n' \] by (4) (4).

In a similar way, one can show that there also exists a subst. \( \sigma' \) with \( \sigma_2' \circ \sigma_n' = \sigma' \circ \sigma_2 \circ \sigma_n \).

The exchange lemma implies that one can impose an arbitrary ordering on literals in a clause and restrict ourselves to resolution steps with the "first" literal in the clause (w.r.t. the ordering).

\( \Rightarrow \) Regard clauses as sequences of literals and use an arbitrary selection function to select some literal from the clause for the next resolution step.

(5 LD = selection fun.)

Prolog uses the selection fact that always takes the leftmost literal. Computation steps that use the first literal in the goal are called Canonical.

**Def 4.34** (Canonical Computations)

A computation \( (G, \emptyset) \vdash (G_2, 0_2) \vdash \ldots \) is called canonical if each resolution step is done using the first literal of the respective goal \( G_i \).

**Thm 4.35** (Resolving Indeterminism 2)

Let \( \Phi \) be a LP, let \( G \) be a query.

For every successful computation \( (G, \Theta) \vdash (\emptyset, 0) \) there also exists a Canonical Computation.
there also exists a **canonical computation**

\[(G, \emptyset) \vdash^+ (\emptyset, \emptyset')\]

of the same length and

\[\emptyset \text{ and } \emptyset' \text{ are identical up to variable renaming.}\]

**Proof**: apply the exchange lemma repeatedly to the original computation \[(G, \emptyset) \vdash^+ (\emptyset, \emptyset)\] until it is canonical.

\[\Rightarrow \text{ Completeness of SLD-resolution still holds if one is restricted to canonical computations.}\]

\[\Rightarrow \text{ improves efficiency - derivation tree does not have to explore the different possibilities resulting from indet. 2.}\]

**Ex 436** In the derivation tree of Ex 431, we can restrict ourselves to canonical computations without losing any solutions. In this example the resulting tree becomes finite.

**Ex 437** Indet 2 can influence the termination behavior:

\[
P : - p.
\]

\[
q(a).
\]

\[
? - q(b), p.
\]

terminates in Prolog, because there is no canonical computation starting in \[(\neg q(b), p), \emptyset)\].
But there exists an infinite non-canonical computation

\[ \{ \neg p(b), \neg p, \emptyset \} \vdash \{ \neg q(b), \neg p, \emptyset \} \vdash \ldots \]

2 Indeterminismus

1. Which rule of the LP is used for the next res. step?

2. Which literal of the current goal is used for the next step?

Def 438 (SLD Tree)

Let \( \mathcal{P} \) be a LP, \( G \) be a query. The SLD tree of \( \mathcal{P} \) w.r.t. the query \( G \) is a finite or infinite tree whose nodes are marked with sequences of atomic formulas. Its edges are marked with substitutions. The SLD tree is the smallest tree such that

- If \( G = \{ \neg A_1, \ldots, \neg A_k \} \), then the root of the tree is marked with \( A_1, \ldots, A_k \).
- If a node is marked with \( B_1, \ldots, B_n \) and \( B_1 \) is unifiable with the positive literals of the prog. clauses \( U_1, \ldots, U_k \) (where \( U_1 \) occurs before \( U_2 \), \( U_2 \) before \( U_3 \) etc. in \( \mathcal{P} \)), then the node has \( k \) successors. Then the \( i \)-th is marked by those atoms that result from a canonical resolution step with \( U_i \).

So if \( \{ \neg B_1, \ldots, \neg B_n \}, \emptyset \) \( \vdash \{ \neg C_1, \ldots, \neg C_m \}, \emptyset \) is this canonical res. step, then the \( i \)-th child is marked with \( C_1, \ldots, C_m \) and the edge to this
child is marked with $\sigma$ (restricted to the variables in $B_1, \ldots, B_n$).

The answer substitutions can be obtained from paths ending in $\Delta$. If the edges from the root to the leaf $\Delta$ are marked with $\sigma_1, \ldots, \sigma_k$, then we obtain the answer subst: $\sigma_k \circ \cdots \circ \sigma_2 \circ \sigma_1$ (restricted to the variables in $\Delta$).

Thus 435 guarantees that by regarding the SCD-tree, we still find all answer substitutions.

Finite paths that end in a clause different from $\Delta$:

Finite failures: $B_1, \ldots, B_n$ where $B_n$ cannot be unified with the positive literal of any prog. clause.

Moreover, there can be infinite paths.

Indet 2: resolved by the SCD-tree

Indet 1: To resolve this indet., we have to fix the order in which the SCD-tree is constructed/traversed. (Evaluation Strategy.) We also have to decide whether to stop as soon as one has found the first $\Delta$ or whether to search for all solutions? (In holog: stop at the first $\Delta$, but entering "\" makes holog continue to the next $\Delta$, etc.)

Options: \ldots
**breadth-first search:** first construct all nodes of height 0, then of height 1, etc.

**advantage:** completeness of canonical SCL-resolution is "preserved", i.e., every □ in the SCL-tree will be found. So if the query follows from the □, this will be found out.

**disadvantage:**

building up the whole SCL-tree up to the level of the first □ can take very long ⇒ too inefficient

**depth-first search:** used by Robig (left-to-right)

**advantage:** □ is found very quickly if it is in the left-most paths

**disadvantage:** this strategy is incomplete

backtrack if a path ends in finite failure or if "□" is encountered after reaching □

there, depth-first search

□ does not terminate and
The programmer should take Prolog's evaluation strategy into account and ensure that solutions are found quickly before entering infinite paths.

(Choose suitable order of the literals in prog. clauses and of the clauses in the prog.)

(Violation of the principle of declarative programming:
The programmer should think (a bit) about how Prolog operates to solve queries:
1. Prog. clauses are used from top to bottom.
2. Literals in a goal are solved from left to right.)

Ex. 4.3.9 Illustrate the effect of exchanging literals in a prog. clause.
Here: the rightmost path of the tree becomes infinite.

Ex. 4.3.10 Illustrate the effect of exchanging clauses in a program.
Here: the leftmost path becomes infinite
⇒ prog. does not terminate and does not find the solutions.
⇒ non-recursive clauses for a pred. p should usually
come before recursive clauses.