Logic Programming and Deductive Databases

Chapter 5: SLD Resolution

Prof. Dr. Stefan Brass

Martin-Luther-Universität Halle-Wittenberg

Summer 2023

http://www.informatik.uni-halle.de/~brass/lp23/

Objectives

After completing this chapter, you should be able to:

- define most general unifier of two termns or literals.
- compute a most general unifier of two terms or literals.
- define the resut of an SLD resolution step for a given proof goal and applicable rule.
- develop an SQL-proof tree for a given query and logic program.
- understand the Prolog debugger output.

Contents

Unification (1)

- Unification is used in Prolog for parameter passing: It matches the actual parameters with the formal parameters of a predicate. It can fail.
- It can also be seen as an assignment that is that is
  - symmetric: $X = a$ and $a = X$ are both legal and have the same effect (X is bound to a),
  - one-time: Once a variable is bound to a value, it is always automatically replaced by that value. It is impossible to assign a new value.
- Unification does pattern matching of tree-structures (terms).

Unification (2)

Definition (Unifier):

- A unifier of two literals A and B is a substitution θ with $A\theta = B\theta$.
- A and B are called unifiable if there is a unifier of A and B.
- θ is a most general unifier of A and B if for every other unifier θ' of A and B there is a substitution σ with $\theta' = \theta \circ \sigma$.

  $\theta \circ \sigma$ denotes the composition of θ and σ, i.e. $(\theta \circ \sigma)(A) = \sigma(\theta(A))$.

Unification (3)

Examples:

- $p(X, b)$ and $p(a, Y)$ are unifiable with most general unifier $\{X/a, Y/b\}$.
- $q(a)$ and $q(b)$ are not unifiable.
- Consider $q(X)$ and $q(Y)$:
  - $\{X/Y\}$ is a most general unifier of these literals.
  - $\{Y/X\}$ is another most general unifier of these literals. (It maps both literals to $q(X)$).
  - $\{X/a, Y/a\}$ is an example for a unifier that is not a most general unifier.

Unification (4)

Lemma:

- If there is a unifier of A and B, there is also a most general unifier (MGU).
- The most general unifier is unique up to variable renamings, i.e. if θ and θ' are both most general unifiers of A and B there is a substitution σ which is a bijective mapping from variables to variables such that $\theta' = \theta \circ \sigma$.

Notation:

- Let $mgu(A, B)$ be a most general unifier of A and B.

Unification (5)

unify(Literal/Term t, u): Substitution θ

if $t = u$ **then**

  $\theta := \{\}$;

else if t is a variable that does not occur in u **then**

  $\theta := \{t/u\}$;

else if u is a variable that does not occur in t **then**

  $\theta := \{u/t\}$;

else if t is $f(t_1, \ldots, t_n)$ and u is $f(u_1, \ldots, u_n)$ **then**

  $\theta := \{\}$;

  for $i := 1$ **to** n **do** $\theta := \theta \circ \text{unify}(t_i\,\theta, u_i\,\theta)$;

else /* Different Functors/Constants */

  $\theta :=$ "not unifiable";

Unification (6)

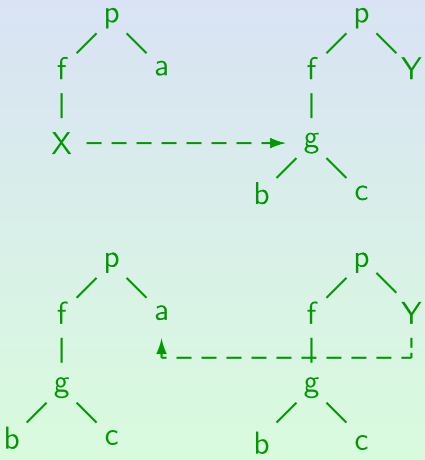
Example:

- $p(X, X)$ and $p(a, b)$ are not unifiable:
  - The first argument is unified with X/a.
  - However, then one has to unify $p(a, a)$ and $p(a, b)$. That is not possible.
- $p(X, X)$ and $p(Y, f(Y))$ are not unifiable:
  - First, one unifies X and Y, e.g. with $\{X/Y\}$.
  - Then one has to unify $p(Y, Y)$ and $p(Y, f(Y))$. It is not possible to bind Y to $f(Y)$, because Y occurs in $f(Y)$.

    $\{Y/f(Y)\}$ would not make the terms equal.

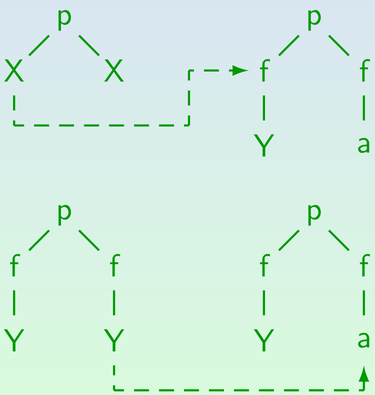
Unification (7)

Example:



Unification (8)

Example:



Unification (9)

Exercises:

- Compute the most general unifier if possible:
  - $length([1, 2, 3], X)$ and $length([], 0)$.
  - $length([1, 2, 3], X)$ and $length([E|R], N1)$.
  - $append(X, [2, 3], [1, 2, 3])$ and $append([F|R], L, [F|A])$.
  - $p(f(X), Z)$ and $p(Y, a)$.
  - $p(f(a), g(X))$ and $p(Y, Y)$.
  - $q(X, Y, h(g(X)))$ and $q(Z, h(Z), h(Z))$.
- Use Prolog to check the solution.

Occur Check (1)

- Suppose that the following to literals are unified:
  - $p(X_1, \ldots, X_n)$,
  - $p(f(X_0, X_0), \ldots, f(X_{n-1}, X_{n-1}))$.
- The unifier is

$$\begin{aligned}\theta = \{&X_1/f(X_0, X_0),\\ &X_2/f(f(X_0, X_0),\ f(X_0, X_0)),\\ &\ldots\}.\end{aligned}$$

- The test, whether X_k appears in t_k ("occur check") costs exponential time.

  An explicit representation of θ would cost exponential time, too. But one normally uses pointers from variables to their values to represent a substitution internally: Then common subterms are stored only once.

Occur Check (2)

- Unification is the basic step in Prolog evaluation. It is bad if it can take exponential time.
- Solutions:
  - Unification without occur check: dangerous.

    This can give wrong solutions: E.g. consider the program consisting of $p \leftarrow q(X, X)$ and $q(Y, f(Y))$. Prolog systems without occur check answer "p" with "yes". It is also possible that unification or the printing of terms get into infinite loops.
  - With better data structures, the occur check has linear runtime.
  - Static analysis of a Program can show where no occur check is needed.

Contents

SLD-Resolution (1)

- SLD-resolution is the theoretical basis of Prolog execution.
- It is a theorem proving procedure that is complete for Horn clauses.
- SLD stands for "Linear resolution for Definite clauses with Selection function".

  In resolution, the basic derivation step is to conclude $A \vee C$ from $A \vee B$ and $\neg B \vee C$: I.e. one matches complementary literals (with a unifier) and composes the rests of the two clauses. It is a refutation proof procedure that starts with the negation of the proof goal and ends with the empty clause (the obvious contradiction). In linear resolution, one of the two clauses is always the result of the previous step.

SLD-Resolution (2)

- The idea of SLD-resolution is to simplify the query (i.e. the proof goal) step by step to "true".

  If seen as refutation proof procedure, the current clause is the negation of the query, and one ends with "false".

- Each step makes a literal from the query and a rule head from the program equal with a unifier.

- Then literal in the query is replaced by the body of the rule. This gives a new query (hopefully simpler).

- Facts are treated as rules with empty body. Using facts makes the query shorter.

SLD-Resolution (3)

Example:

- Consider the following program:

$$\begin{aligned}
&(1)\ \mathtt{ancestor}(\mathtt{X},\mathtt{Y}) \leftarrow \mathtt{parent}(\mathtt{X},\mathtt{Y}).\\
&(2)\ \mathtt{ancestor}(\mathtt{X},\mathtt{Z}) \leftarrow \mathtt{parent}(\mathtt{X},\mathtt{Y}) \wedge \mathtt{ancestor}(\mathtt{Y},\mathtt{Z}).\\
&(3)\ \mathtt{parent}(\mathtt{X},\mathtt{Y}) \leftarrow \mathtt{mother}(\mathtt{X},\mathtt{Y}).\\
&(4)\ \mathtt{parent}(\mathtt{X},\mathtt{Y}) \leftarrow \mathtt{father}(\mathtt{X},\mathtt{Y}).\\
&(5)\ \mathtt{father}(\mathtt{julia},\mathtt{eric}).\\
&(6)\ \mathtt{mother}(\mathtt{eric},\mathtt{bianca}).
\end{aligned}$$

- Let the query be

$$\mathtt{ancestor}(\mathtt{julia},\mathtt{bianca}).$$

SLD-Resolution (4)

- The given query is the first proof goal:

$$\mathtt{ancestor(julia, bianca)}.$$

- The only literal in the proof goal can be resolved with

$$(2)\ \mathtt{ancestor(X, Z)} \leftarrow \mathtt{parent(X, Y)} \land \mathtt{ancestor(Y, Z)}.$$

- The most general unifier of query literal and rule head is $\{\mathtt{X/julia},\ \mathtt{Z/bianca}\}$.

- Now the new proof goal is

$$\mathtt{parent(julia, Y)} \land \mathtt{ancestor(Y, bianca)}.$$

SLD-Resolution (5)

- Prolog always works on the first literal of the proof goal (this is a special selection function):

$$\underline{\texttt{parent}(\texttt{julia}, \texttt{Y})} \wedge \texttt{ancestor}(\texttt{Y}, \texttt{bianca}).$$

- It can be resolved with rule (4):

$$(4)\ \texttt{parent}(\texttt{X}, \texttt{Y}) \leftarrow \texttt{father}(\texttt{X}, \texttt{Y}).$$

  This gives

$$\underline{\texttt{father}(\texttt{julia}, \texttt{Y})} \wedge \texttt{ancestor}(\texttt{Y}, \texttt{bianca}).$$

- Then the fact (5) is applied (with unifier $\{\texttt{Y}/\texttt{eric}\}$).

$$(5)\ \texttt{father}(\texttt{julia}, \texttt{eric}).$$

  This gives the proof goal:

$$\texttt{ancestor}(\texttt{eric}, \texttt{bianca}).$$

SLD-Resolution (6)

- For the proof goal

$$\texttt{ancestor}(\texttt{eric}, \texttt{bianca}),$$

one can e.g. apply rule (1) $\texttt{ancestor}(\texttt{X}, \texttt{Y}) \leftarrow \texttt{parent}(\texttt{X}, \texttt{Y})$.

- This replaces the proof goal by:

$$\texttt{parent}(\texttt{eric}, \texttt{bianca}).$$

- Now one can apply rule (3) $\texttt{parent}(\texttt{X}, \texttt{Y}) \leftarrow \texttt{mother}(\texttt{X}, \texttt{Y})$. and get the proof goal

$$\texttt{mother}(\texttt{eric}, \texttt{bianca}).$$

- This is given as a fact (line (6) in the program), and one gets the empty proof goal "$\square$".

- Thus, the query indeed follows from the given program, and the answer "yes" is printed.

SLD-Resolution (7)

- A sequence of proof goals that
  - starts with a query Q and
  - ends in the empty goal

  is called a derivation of Q from the given program.
- In the above derivation, the right program rule was "guessed" in each step. Prolog will try all possibilities with backtracking.
- If a query contains variables, the answer computed by a derivation is the composition of all substitutions applied.

SLD-Resolution (8)

Definition (Selection Function):

- A selection function is a mapping that, given a proof goal $A_1 \wedge \cdots \wedge A_n$, returns an index i in the range from 1 to n. (I.e. it selects a literal A_i.)

Note:

- Prolog uses the first literal selection rule, i.e. it selects always A_1 in $A_1 \wedge \cdots \wedge A_n$.
- As we will see, in deductive databases, a good selection function is an important part of the optimizer.

    The Prolog selection function also does not guarantee completeness for the answer "no". However, it is easy to implement with a stack.

SLD-Resolution (9)

Definition (SLD-Resolution Derivation Step):

- Let $A_1 \wedge \cdots \wedge A_n$ be a proof goal (query).
- Suppose the selection function chooses A_i.
- Let $B \leftarrow B_1 \wedge \cdots \wedge B_m$ be a rule from the program.
- Replace the variables in the rule by new variables, let the result be $B' \leftarrow B'_1 \wedge \cdots \wedge B'_m$.
- Let A_i and B' be unifiable, $\theta := mgu(A_i, B')$.
- Then the result of the SLD-resolution step is

$$(A_1 \wedge \cdots \wedge A_{i-1} \wedge B'_1 \wedge \cdots \wedge B'_m \wedge A_{i+1} \wedge \cdots \wedge A_n)\theta.$$

SLD-Resolution (10)

Definition (Applicable Rule):

- In the above situation, the rule $B \leftarrow B_1 \wedge \cdots \wedge B_m$ is called applicable to the proof goal $A_1 \wedge \cdots \wedge A_n$.
- I.e. after renaming the variables in the rule, giving $B' \leftarrow B'_1 \wedge \cdots \wedge B'_m$, the head literal B' unifies with the selected literal A_i in the proof goal.

Note:

- Several rules in the program can be applicable to the same proof goal.

  This leads to branches in the SLD-tree explained below.

SLD-Resolution (11)

- It is important that the variables of the rule are renamed such that there is no name clash with a variable in the proof goal.

  Or a previous substitution, see computed answer substitution below.

- E.g. suppose the proof goal is $p(X, a)$ and the rule to be applied is $p(b, X) \leftarrow$.
- There is no unifier of $p(X, a)$ and $p(b, X)$.
- However, variable names in rules are not important. If the variable in the rule is renamed, e.g. to X_1, the MGU is $\{X/b,\ X_1/a\}$.

SLD-Derivations (1)

Definition (SLD-Derivation, Successful SLD-Derivation):

- Let a logic program P, a query Q, and a selection function be given.
- An SLD-derivation for Q is a (finite or infinite) sequence of proof goals $Q_0, \; Q_1, \ldots, Q_n, \; \ldots$ such that
  - $Q_0 = Q$ and
  - Q_i for $i \geq 1$ is the result of an SLD-derivation step from Q_{i-1} and a rule from P.
- An SQL-derivation is successful iff it is finite and ends in the empty clause $\square$.

SLD-Derivations (2)

Definition (Failed SLD-Derivation):

- An SLD-derivation $Q_0, \ldots, Q_n$ is failed iff it is finite, the last goal Q_n is not the empty clause $\square$, and the given program does not contain a rule that is applicable to Q_n.

Summary: Classification of SLD-Derivations:

- Successful: Finite, ends in $\square$.
- Failed: Finite, ends not in $\square$, no applicable rule.
- Incomplete: Finite, there is an applicable rule.
- Infinite.

SLD-Derivations (3)

Example (shown also on next page with applied rules):

- $\texttt{ancestor}(\texttt{julia}, \texttt{bianca})$.
- $\texttt{parent}(\texttt{julia}, \texttt{Y}) \wedge \texttt{ancestor}(\texttt{Y}, \texttt{bianca})$.
- $\texttt{father}(\texttt{julia}, \texttt{Y}) \wedge \texttt{ancestor}(\texttt{Y}, \texttt{bianca})$.
- $\texttt{ancestor}(\texttt{eric}, \texttt{bianca})$.
- $\texttt{parent}(\texttt{eric}, \texttt{bianca})$.
- $\texttt{mother}(\texttt{eric}, \texttt{bianca})$.
- $\square$.

SLD-Derivations (4)

ancestor(julia, bianca).

$\boxed{\texttt{ancestor(X, Z)} \leftarrow \texttt{parent(X, Y)} \wedge \texttt{ancestor(Y, Z).}}$

parent(julia, Y) ∧ ancestor(Y, bianca).

$\boxed{\texttt{parent(X, Y)} \leftarrow \texttt{father(X, Y).}}$

father(julia, Y) ∧ ancestor(Y, bianca).

$\boxed{\texttt{father(julia, eric).}}$

ancestor(eric, bianca).

$\boxed{\texttt{ancestor(X, Y)} \leftarrow \texttt{parent(X, Y).}}$

parent(eric, bianca).

$\boxed{\texttt{parent(X, Y)} \leftarrow \texttt{mother(X, Y).}}$

mother(eric, bianca).

$\boxed{\texttt{mother(eric, bianca).}}$

□

SLD-Derivations (5)

Exercise:

- Let the following logic program be given:

```
append([], L, L).
append([F|R], L, [F|A])  ←  append(R, L, A).
```

- Give a successful SLD-derivation for

```
append([1], [2], [1,2]).
```

- What are the applied rules and most general unifiers in each step?

Contents

Computed Answers (1)

Definition (Computed Answer Substitution):

- Given a logic program P and a query Q, let
$$Q_0 = Q,\; Q_1, \ldots, Q_n$$
be a successful SLD-derivation for Q, and $\theta_1, \ldots, \theta_n$ be the most general unifiers applied in the SLD resolution steps.
- Let θ be the composition $\theta_1 \circ \cdots \circ \theta_n$ of these unifers, restricted to the variables that occur in the query Q.
- Then θ is a computed answer substitution for Q.

  Or: The answer substitution computed by this SLD-derivation.

Computed Answers (2)

Example (For Program on Slide 18):

- A successful derivation for $\mathtt{parent(X, Y)}$ is as follows:
  - Goal: $\mathtt{parent(X, Y)}$.
    Rule: $\mathtt{parent(X_1, Y_1)} \leftarrow \mathtt{mother(X_1, Y_1)}$.
    MGU: $\theta_1 := \{\mathtt{X/X_1},\ \mathtt{Y/Y_1}\}$.
  - Goal: $\mathtt{mother(X_1, Y_1)}$.
    Rule: $\mathtt{mother(eric, bianca)}$.
    MGU: $\theta_2 := \{\mathtt{X_1/eric, Y_1/bianca}\}$.
  - Goal: $\square$.
- $\theta_1 \circ \theta_2 = \{\mathtt{X/eric},\ \mathtt{Y/bianca},\ \mathtt{X_1/eric},\ \mathtt{Y_1/bianca}\}$.
- Computed answer substitution: $\{\mathtt{X/eric},\ \mathtt{Y/bianca}\}$.

Computed Answers (3)

Theorem (Correctness of SLD-Resolution):

- For every program P, query Q, and computed answer substitution θ: $P \models Q\,\theta$.

  I.e. the program (set of Horn clauses) logically implies the query (conjunction of positive literals) after the answer substitution is applied to the query. As always, variables are treated as universally quantified.

Theorem (Completeness of SLD-Resolution):

- For every program P, query Q, and substitution θ with $P \models Q\,\theta$, there is a computed answer substitution θ_0 and a substitution θ_1 such that $\theta = \theta_0 \circ \theta_1$.

  I.e. for every correct answer substitution, SLD-resolution either computes it, or it computes a more general substitution.

Computed Answers (4)

Note (On the Completeness):

- E.g. consider the program consisting of the rule

$$p(f(X)) \leftarrow .$$

- Let the query be $p(Y)$.
- The substitution $\theta := \{Y/f(a)\}$ is correct, i.e. it satisfies $P \models Q\,\theta$, but SLD-resolution computes the more general substitution $\theta_0 := \{Y/f(X)\}$.
- θ_0 is more general than θ, because it can be composed with $\theta_1 := \{X/a\}$ to give θ.

Computed Answers (5)

Note (On Prolog):

- The correctness result holds only if the Prolog system does the occur check, e.g. try the program P:

```
p ← q(X,X).
q(X, f(X)).
```

  Prolog systems without occur check answer "p" with "yes", but p is not a logical consequence of P.

- The completeness result holds only if the Prolog system terminates. Prolog might run into an infinite loop before it finds all answers.

Contents

SLD-Trees (1)

- There are usually more than one SLD-derivation for a given query, because for every proof goal, more than one rule might be applicable.
- Every successful SLD-derivation computes only one answer substitution, but a query might have several distinct correct answer substitutions.

  Thus, it is important for the completeness of SLD-resolution, that there can be several SLD-derivations for the same query.
- The different SLD-derivations for a given query are usually displayed in form of a tree, the SLD-tree.

SLD-Trees (2)

Definition (SLD-Tree):

- The SLD-tree for a program P and a query Q (and a given selection function) is constructed as follows:
  - Every node of the tree is labelled with a proof goal (query). The root node is labelled with Q.
  - Let a node $\mathcal{N}$ be labelled with the proof goal
  $$A_1 \wedge \cdots \wedge A_n, \qquad n \geq 1.$$
  Then $\mathcal{N}$ has a child node for every rule
  $$B \leftarrow B_1 \wedge \cdots \wedge B_m$$
  in P that is applicable to $A_1 \wedge \cdots \wedge A_n$.

    The child node is labelled with the result of the corresponding SLD-resolution step.

SLD-Trees (3)

Example:

- Consider the following program:

$$
\begin{array}{l}
(1)\ \mathtt{parent}(\mathtt{X}, \mathtt{Y}) \leftarrow \mathtt{mother}(\mathtt{X}, \mathtt{Y}).\\
(2)\ \mathtt{parent}(\mathtt{X}, \mathtt{Y}) \leftarrow \mathtt{father}(\mathtt{X}, \mathtt{Y}).\\
(3)\ \mathtt{father}(\mathtt{julia}, \mathtt{eric}).\\
(4)\ \mathtt{mother}(\mathtt{julia}, \mathtt{fiona}).\\
(5)\ \mathtt{father}(\mathtt{ian}, \mathtt{eric}).\\
(6)\ \mathtt{mother}(\mathtt{ian}, \mathtt{fiona}).
\end{array}
$$

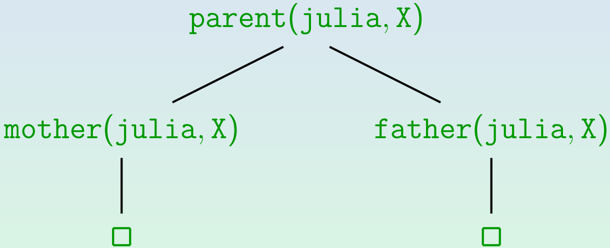
- Let the query be

$$\mathtt{parent}(\mathtt{julia}, \mathtt{X}).$$

- The SLD-Tree is shown on the next page.

SLD-Trees (4)

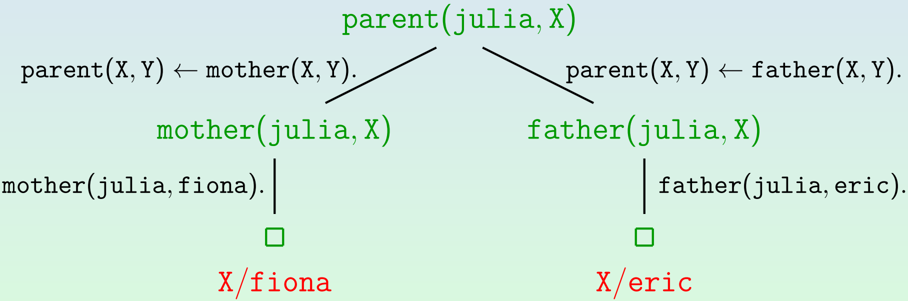
SLD-Tree:



- Often, it is also useful to know the applied rules and/or the computed answers. This information is shown in the variant on the next page.

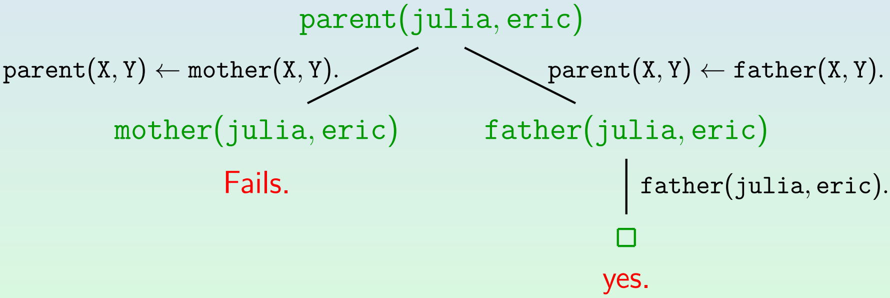
SLD-Trees (5)

SLD-Tree (with applied rules and computed answers):



SLD-Trees (6)

Another Example (Is eric parent of julia?):



SLD-Trees (7)

- Please note that branching in an SLD-tree happens only when there are several applicable rules.

  There is exactly one child node for each applicable rule, i.e. a rule of which the head literal is unifiable with the selected literal in the current node. I.e. the branching is done only for disjunctions ($\vee$).

- If a rule has several body literals, these are added together to the current goal.

  I.e. for conjunctions ($\wedge$) no branching is done (otherwise the binding of common variables would become difficult). If there is always only one applicable rule, the SLD-tree is a single path from root to leaf, even if the rules have many body literals. In the examples on the slides, the rules have only a single body literal, because there is little space. On Slide 30 an SLD-derivation (a single branch in the SLD-tree) is shown in which a rule has two body literals.

SLD-Trees (8)

Exercise:

- Consider again the program for list concatenation:

```
(1)  append([], L, L).
(2)  append([F|R], L, [F|A]) ← append(R, L, A).
```

- What is the SLD-tree for

```
append(X, Y, [1,2]).
```

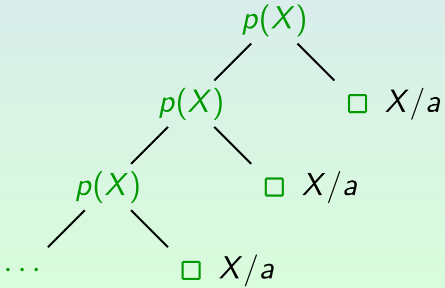
- Which answers do the different paths in the SLD-tree (i.e. the SLD-derivations) compute?

Infinite Paths (1)

- Consider the following program:

$$\begin{array}{lll} (1) & p(X) \leftarrow & p(X). \\ (2) & p(a). & \end{array}$$

- The query $p(X)$ has the following SLD-tree:



Infinite Paths (2)

- Prolog searches the SLD-tree depth first.

  It also uses alternative rules always in the order that they are written down in the program.

- In this example, Prolog will get into an infinite loop and will not compute the correct answer substitution $\{X/a\}$. Thus, Prolog is not complete.

- However, if one would search the SLD-tree breadth-first, one would find all correct answer substitutions (because of the completeness of SLD-resolution).

Infinite Paths (3)

- But depth-first search is much more efficient to implement (with a stack).

- One solution is iterative deepening.

  First, one searches the SLD-tree depth-first, but e.g. only to depth 5. Then, one searches the SLD-tree again up to depth 10 (printing only answers below depth 5). And so on.

- In the XSB-system, it one can switch on "tabling" for selected predicates. Then the system detects when the same selected literal appears again.

  Then infinite loops can happen only when more and more complicated terms are constructed. For programs without function symbols (and built-in predicates), termination is guaranteed.

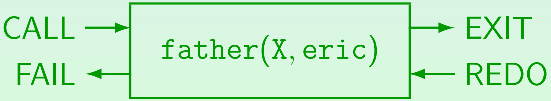
Contents

Box Model (1)

- Prolog uses SLD-resolution with
  - the first-literal selection function, and
  - depth-first search of the SLD-tree.
- However, the Prolog debugger does not show the entire proof goal (node label in the SLD-tree).
- Instead, it views predicates as nondeterministic procedures (procedures that can have more than one solution).
- The four-port debugger model is standard among Prolog systems.

Box Model (2)

- Each predicate invocation (selected literal in the SLD-tree) is represented as a box with four ports:
  - CALL A: Call of A, find first solution.
  - REDO A: Is there another solution for A?
  - EXIT A: A solution was found, A is proven.
  - FAIL A: There is no (more) solution for A.



Box Model (3)

- E.g. consider the following small program:

```
father(ian, eric).
father(julia, eric).
father(eric, alan).
```

- Debugger output for the query `father(X, eric)`:
  - `CALL father(X, eric)`
  - `EXIT father(ian, eric)`

    Note that the proven instance is shown.
  - Then the solution `X/ian` is displayed.
    Suppose one presses "`;`" to get more solutions.

Box Model (4)

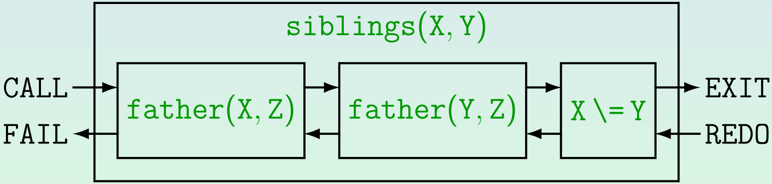
- Example debugger output, continued:
  - `REDO father(X, eric)`
  - `EXIT father(julia, eric)`
  - The solution `X/julia` is displayed. Some systems already know that there is no further solution. Otherwise, one can press again "`;`".
  - `REDO father(X, eric)`
  - `FAIL father(X, eric)`
  - The system prints "`no`".

Box Model (5)

- Suppose the program is extended with the rule

$$\texttt{siblings}(\texttt{X}, \texttt{Y}) \leftarrow \texttt{father}(\texttt{X}, \texttt{Z}) \wedge \texttt{father}(\texttt{Y}, \texttt{Z}) \wedge \texttt{X} \mathbin{\backslash=} \texttt{Y}.$$

- The box model is:



E.g. when the first or second body literal exists, the next body literal is called. When the last body literal is proven, `siblings` exits.

Box Model (6)

Debugger Output for the query siblings(ian, Y):

```
(1)  0   CALL  siblings(ian, Y).
(2)  1   CALL  father(ian, Z).
(2)  1   EXIT  father(ian, eric).
(3)  1   CALL  father(Y, eric).
(3)  1  *EXIT  father(ian, eric).
(4)  1   CALL  ian \= ian.
(4)  1   FAIL  ian \= ian.
(3)  1   REDO  father(Y, eric).
(3)  1   EXIT  father(julia, eric).
(5)  1   CALL  julia \= ian.
(5)  1   EXIT  julia \= ian.
(1)  0   EXIT  siblings(ian, julia).
```

Box Model (7)

Remark:

- The exact form of the output depends on the Prolog system.
- The above output contains a box number in the first column and a nesting depth (call stack depth) in the second column.
- The asterisc "∗" before `EXIT` marks that there are possibly further solutions (nondeterministic exit).

  Otherwise, the box is already removed, and not visited during backtracking (i.e. no `REDO`-`FAIL` will be shown). Because of such optimizations, the debugger output might violate the pure four-port model.

Box Model (8)

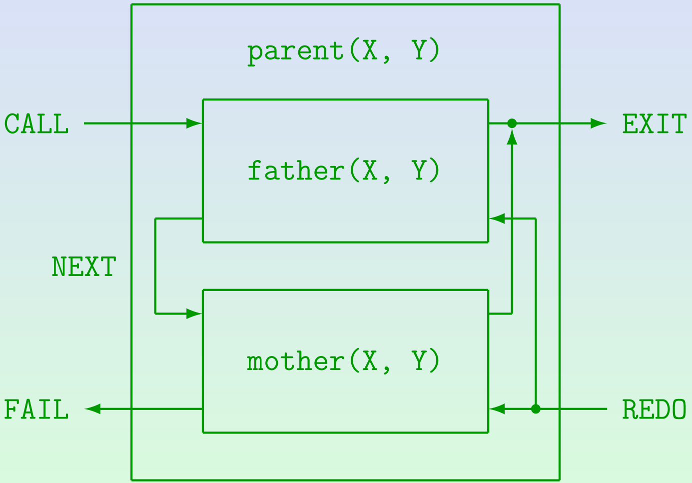
- Consider now a predicate defined with two rules:

$$\begin{array}{lcl} \texttt{parent}(X, Y) & \leftarrow & \texttt{father}(X, Y). \\ \texttt{parent}(X, Y) & \leftarrow & \texttt{mother}(X, Y). \end{array}$$

- The box model for parent is shown on the next page.

  There, also a port NEXT appears. This is a speciality of ECLiPSe Prolog. It shows when execution moves to another rule for the same predicate. In general, different Prolog systems have extended the basic Four-Port Model in various ways. E.g. SWI-Prolog can display a port "UNIFY" that shows the called literal after unification with the rule head.

Box Model (9)



REDO enters the inner box that was last left with EXIT.

Using the Debugger (1)

- The debugger output is switched on by executing the built-in predicate "`trace`" (as a query).

  It is switched off with "`notrace`". In SWI-Prolog, `trace` means only that the next query is traced.

- The debugger then displays a line for every port and waits for commands after each line.
- With "`Return`" one steps to the next port.
- Other commands are listed in the manual.

  Often, they are displayed when one enters "`?`". The command "`a`" should stop execution of the query ("abort").

Using the Debugger (2)

- It is possible to produce debugger output only selectively.
- One can set breakpoints ("spypoints") on a predicate with e.g.

```
spy father/2.
```

- If instead of "`trace`", one uses "`debug`", Prolog executes the program without interruption until it reaches a predicate with a spypoint set.

  Then one can continue debugging as above or "leap" to the next spypoint (usually with the command "`l`"). Of course, there are "`nodebug`" and "`nospy`".