Chapter 3. Operational Semantics

- Last updated: May 22, 2017 -

Unification
SLD-Resolution
Negation as Failure
Incompleteness of SLD-Resolution
Unification

Equality
Variable bindings, Substitutions, Unification
Most general unifiers
Resolution and Unification

- **Resolution** is the process by which Prolog derives answers (successful substitutions) for queries.

- During **resolution**, clauses that can be used to prove a goal are determined via **unification**.

Unification is the basic operation of any Prolog interpreter.
Equality (1)

Testing equality of terms

- \texttt{europe = europe.} \quad \text{yes}
- \texttt{5 = 2.} \quad \text{no}
- \texttt{5 = 2 + 3.} \quad \text{no}
- \texttt{2 + 3 = +(2, 3).} \quad \text{yes}

- Terms are not evaluated!

- Terms are equal if they are \textbf{structurally equal}!!

- \textbf{Structural equality for ground terms:}
  - functors are equal and …
  - … all argument values in the same position are structurally equal.

\textbf{Constants are just functors with zero arity!}
Testing equality of terms with variables

These terms are obviously not equal.

Idea
- A variable can take on any value.
- If we substitute Name by mueller and MM by 11 the two terms above will be equal.

Principle
- Equality = terms are equal
- Unifiability = terms can be made equal via a substitution.
- Prolog doesn’t test equality but unifiability!
Unifiability (3)

Two terms are **unifiable** if there is a **substitution** that makes them equal!

```
?- person(peter, Name, date(27, 11, 2007)) =
   person(peter, mueller, date(27, MM, 2007)).
```

**Definitions: Bindings, substitutions and unifiers**

- A **binding** is an association of a variable to a term different from the variable
  - `Name ← mueller`
  - `Y ← Z`  \(\rightarrow\) The right-hand-side term can be a variable!
  - `MM ← month(M)`  \(\rightarrow\) The right-hand-side term can contain variables!

- A **substitution** is a set of bindings with pairwise different variables
  - \{`Name ← mueller, MM ← 11`\}

- A **unifier** is a substitution that makes two terms equal
  - The above substitution is a unifier for the two `person/3` terms above
Unifiability (2)

- Can you find out the unifiers for these terms?

\[
\begin{align*}
\text{date}(1, 4, 1985) &= \text{date}(1, 4, \text{Year}) \\
\text{date}(\text{Day}, \text{Month}, 1985) &= \text{date}(1, 4, \text{Year}) \\
a(b, C, d(e, F, g(h, i, J))) &= a(B, c, d(E, f, g(H, i, j))) \\
[[\text{the}, Y]|Z] &= [[X, \text{dog}], [\text{is}, \text{here}]] \\
X &= Y + 1
\end{align*}
\]

- What about

\[
\begin{align*}
p(X) &= p(q(X)) \\
x &= q(x)
\end{align*}
\]

produces a cyclic substitution
Application of a Substitution to a Term (1)

Notation

- Substitutions are denoted by greek letters: \( \gamma, \sigma, \tau, \ldots \)
  - \( \tau = \{ \text{Year} \leftarrow 1985, \text{Month} \leftarrow 4 \} \)

Definition: Application of a Substitution

Let \( T \) be any term.
Let \( \tau = \{ V_1 \leftarrow t_1, \ldots, V_n \leftarrow t_n \} \) be any substitution.

\( T\tau \), the application of \( \tau \) to \( T \), replaces (for \( i = 1 \ldots n \)) each occurrence of \( V_i \) in \( T \) by \( t_i \tau \).

Examples

\[
\text{date}(\text{Day}, \text{Month}, 1985) \tau \equiv \text{date}(\text{Day}, 4, 1985)
\]
\[
X = Y + 1 \{ X \leftarrow Y + 1 \} \equiv Y + 1 = Y + 1
\]
\[
f(X, 1) \{ Y \leftarrow 2, X \leftarrow g(Y) \} \equiv f(g(2), 1)
\]
Application of a Substitution to a Term (2)

Important

For \( \tau = \{ V_1 \leftarrow t_1, \ldots, V_n \leftarrow t_n \} \) and \( i = 1..n \)

\( T_\tau \) replaces all the occurrences of \( V_i \) in \( T \) by \( t_i \).

Substitutions are applied to their own right-hand-sides too!

Therefore:

\[
\begin{align*}
  f(X,1) \{ Y \leftarrow 2, X \leftarrow g(Y) \} & \equiv f(g(2),1) \\
  p(X) \{ X \leftarrow q(X) \} & \equiv p(q(q(q(q(q(q(q(q(...)))))),...) 
\end{align*}
\]

This would be wrong:

\[
\begin{align*}
  f(X,1) \{ Y \leftarrow 2, X \leftarrow g(Y) \} & \equiv f(g(Y),1) \\
  p(X) \{ X \leftarrow q(X) \} & \equiv p(q(q(q(q(q(q(q(q(q(q(...)))))),...) 
\end{align*}
\]

Resulting Problem

- Application of cyclic substitutions creates infinite terms

Therefore:

\[
\begin{align*}
  f(X,1) \{ Y \leftarrow 2, X \leftarrow g(Y) \} & \equiv f(g(2),1) \\
  p(X) \{ X \leftarrow q(X) \} & \equiv p(q(q(q(q(q(q(q(q(q(q(...)))))),...) 
\end{align*}
\]

This would be wrong:

\[
\begin{align*}
  f(X,1) \{ Y \leftarrow 2, X \leftarrow g(Y) \} & \equiv f(g(Y),1) \\
  p(X) \{ X \leftarrow q(X) \} & \equiv p(q(q(q(q(q(q(q(q(q(q(q(...)))))),...) 
\end{align*}
\]

Prevention: Don’t create cyclic substitutions in the first place!

- “Occurs Check” verifies whether unification would create cyclic substitutions
“Occurs Check”

Theory
- Unification must fail if it would create substitutions with cyclic bindings

\[
p(X) = p(q(X)) \quad \% \text{ must fail}
\]

Problem
- Unification with “occurs-check” has exponential worst-case run-time
- Unification without “occurs-check” has linear worst-case run-time

Practical Prolog implementations
- Prolog implementations do not perform the occurs check

\[
p(X) = p(q(X)) \quad \% \text{succeeds}
\]
- … unless you explicitly ask for it

\[
\text{unify_with_occurs_check}(p(X), p(q(X))) \quad \% \text{ fails}
\]
Cyclic Substitutions

- Cyclic substitutions result from lack of „occurs check“:

  ```prolog
  ?- X=f(X).
  X = f(X).
  ```

- Many Prolog implementations produce an infinite loop when applying cyclic substitutions, e.g. when trying to write a term that contains a cyclically substituted variable:

  ```prolog
  - X=f(X), write('This is what we get: '), write(X).
  'This is what we get: f(f(f(f(f(f(f(...
  ```

- SWI-Prolog supports graceful printing of cyclic terms:

  ```prolog
  ?- X=f(X), write('This is what we get: '), write(X).
  'This is what we get: @(S_1,[S_1=f(S_1)])
  X = f(X).
  ```

  The `@`(Var,Subst) term tells us that Subst is a cyclic substitution for Var

  See http://www.swi-prolog.org/pldoc/man?section=cyclic
Unification

- Unification of terms $T_1$ and $T_2$ finds a substitution $\sigma$ for the variables of $T_1$ and $T_2$ such that $T_1\sigma = T_2\sigma$.

- Unification satisfies equations
- … but only if possible

Question

- How to unify two variables?
  - Problem: Infinitely many unifying substitutions possible!!

Solution

- Unification finds the most general unifying substitution
  - “most general unifier” (mgu)

$\text{?- } p(X, f(Y), a) = p(a, f(a), Y).$
$X = a, Y = a.$
$\text{?- } p(X, f(Y), a) = p(a, f(b), Y).$
fail.

$\text{?- } p(X) = p(Y).$
$X = a, Y = a;$
$X = b, Y = b;$
...

$\text{?- } p(X) = p(Z).$
$X = _G800, Z = _G800;$
true.
Unification yields Most General Unifier (MGU)

- Unification of terms $T_1$ and $T_2$ finds a most general substitution $\sigma$ for the variables of $T_1$ and $T_2$ such that $T_1\sigma = T_2\sigma$.

**Theorem (Uniqueness of MGU):** The most general unifier of two terms $T_1$ and $T_2$ is uniquely determined, up to renaming of variables.

- More formally: If there are two different most general unifiers of $T_1$ and $T_2$, say $\sigma$ and $\tau$, then there is also a renaming substitution $\gamma$ such that $T_1\sigma\gamma \equiv T_2\tau$.

- A renaming substitution only binds variables to variables, e.g:

  $\text{f}(A) \{A \leftarrow B, \ B \leftarrow C, \ D \leftarrow E\} \equiv \text{f}(C)$
Computing the Most General Unifier \( \text{mgu}(T_1, T_2) \)

- **Input:** two terms, \( T_1 \) and \( T_2 \)
- **Output:** \( \sigma \), the most general unifier of \( T_1 \) and \( T_2 \) (only if \( T_1 \) and \( T_2 \) are unifiable)

- **Algorithm**
  1. If \( T_1 \) and \( T_2 \) are the same constant or variable then \( \sigma = \{ \} \)
  2. If \( T_1 \) is a variable not occurring in \( T_2 \) then \( \sigma = \{ T_1 \leftarrow T_2 \} \)
  3. If \( T_2 \) is a variable not occurring in \( T_1 \) then \( \sigma = \{ T_2 \leftarrow T_1 \} \)
  4. If \( T_1 = f(T_{11}, \ldots, T_{1n}) \) and \( T_2 = f(T_{21}, \ldots, T_{2n}) \) are function terms with the same functor and arity
     1. Determine \( \sigma_1 = \text{mgu}(T_{11}, T_{21}) \)
     2. Determine \( \sigma_2 = \text{mgu}(T_{12}\sigma_1, T_{22}\sigma_1) \)
     3. \ldots
     4. Determine \( \sigma_n = \text{mgu}(T_{1n}\sigma_1\ldots\sigma_{n-1}, T_{2n}\sigma_1\ldots\sigma_{n-1}) \)
     5. If all unifiers exist then \( \sigma = \sigma_1\ldots\sigma_{n-1}\sigma_n \)
        (otherwise \( T_1 \) and \( T_2 \) are not unifiable)
  6. Occurs check: If \( \sigma \) is cyclic fail, else return \( \sigma \)

5. If none of the above applies, fail.
Composition of Substitutions

The MGU algorithm needs to apply a sequence of computed substitutions \( \sigma_1 \ldots \sigma_n \) to a term \( T \) (previous page, line 4.4):

\[
T \sigma_1 \ldots \sigma_n \quad \uparrow \quad T \sigma
\]

The operation that computes a single substitution

\[ \sigma = \sigma_1 \ldots \sigma_n \]

whose application to any term will yield the same result as the application of the entire sequence \( \sigma_1 \ldots \sigma_n \) is called composition of substitutions.
Composition Defined

Let \( \alpha = \{v_1 \leftarrow t_1, \ldots, v_n \leftarrow t_n\} \) and \( \beta = \{w_1 \leftarrow u_1, \ldots, w_m \leftarrow u_m\} \) be two substitutions.

Then \( \alpha \beta \), the composition of \( \alpha \) and \( \beta \), is the substitution obtained by

1. deleting conflicting bindings from \( \beta \)

\[ \beta = \beta \setminus \{w_i \leftarrow u_i \mid w_i \in \{v_1, \ldots, v_n\}\} \]

2. applying the non-conflicting subset of \( \beta \) to each right-hand-side of \( \alpha \)

\[ \delta = \{v_1 \leftarrow t_1 \beta, \ldots, v_n \leftarrow t_n \beta\} \]

3. deleting trivial bindings resulting from the previous step and

\[ \delta = \delta \setminus \{v_i \leftarrow t_i \beta \mid t_i \beta = v_i\} \]

4. appending the non-conflicting bindings from \( \beta \)

\[ \alpha \beta = \delta \cup \beta \]
Composition Example (1)

- Let $\alpha = \{X \leftarrow f(Y), Y \leftarrow Z\}$ and $\beta = \{X \leftarrow a, Z \leftarrow Y, Y \leftarrow b\}$.

  1. Delete the conflicting bindings $X \leftarrow a$ and $Y \leftarrow b$
     
     $\beta = \beta \setminus \{X \leftarrow a, Y \leftarrow b\} = \{Z \leftarrow Y\}$

  2. Apply $\beta$ to the right-hand sides of $\alpha$
     
     $\delta = \{X \leftarrow f(Y) \beta, Y \leftarrow Z \beta\} = \{X \leftarrow f(Y), Y \leftarrow Y\}$

  3. Delete the trivial binding $Y \leftarrow Y$:
     
     $\delta = \{X \leftarrow f(Y)\}$

  4. Union $\delta$ and $\beta$
     
     $\alpha \beta = \delta \cup \beta = \{X \leftarrow f(Y), Z \leftarrow Y\}$
Composition Example (2)

Let \( \alpha = \{ Y \leftarrow 2 , \ X \leftarrow g(Y,Z) \} \)
and \( \beta = \{ Y \leftarrow 3 , \ Z \leftarrow 4 \} \).

1. Delete the conflicting binding \( Y \leftarrow 3 \)
   \[ \beta = \beta \setminus \{ Y \leftarrow 3 \} = \{ Z \leftarrow 4 \} \]

2. Apply \( \beta \) to the right-hand sides of \( \alpha \)
   \[ \delta = \{ Y \leftarrow 2 , \ X \leftarrow g(Y,Z) \beta \} = \{ Y \leftarrow 2 , \ X \leftarrow g(Y,4) \} \]

3. Skip (No trivial bindings to be deleted):
   \[ \tilde{\delta} = \tilde{\delta} \]

4. Union \( \tilde{\delta} \) and \( \beta \)
   \[ \alpha \beta = \tilde{\delta} \cup \beta = \{ Y \leftarrow 2 , \ X \leftarrow g(Y,4) , \ Z \leftarrow 4 \} \]
Composition Comments

- Note the difference
  - \( \sigma_1 \sigma_2 \) is the composition of two substitutions
  - \( t \sigma_1 \) is the application of a substitution to a term
  - \( t \sigma_1 \sigma_2 \) is a shorthand for \((t \sigma_1) \sigma_2\)
  - \((t \sigma_1) \sigma_2 \) is the successive application of two substitution to a term
  - \( t (\sigma_1 \sigma_2) \) is the application of a composed substitution to a term

- From the definition of composition given above it follows that
  \[
  (t \sigma_1) \sigma_2 = t (\sigma_1 \sigma_2) 
  \]
Chapter 3: Operational Semantics

SLD-Resolution
General Resolution Idea

Resolution Idea

Given a goal G to be proved  
\(?- P, L, Q.\)

and a clause  
\(L_0 : - L_1, \ldots, L_n (n \geq 0)\)

such that  
L and \(L_0\) are identical

we could prove  
\(?- P, L_1, \ldots, L_n, Q.\)

instead of G, knowing that  
L is implied by \(L_1, \ldots, L_n\)

* Each time we can use a clause \(L_0 : - true.\)  
(that is, a fact) we would replace L by true.

* If we can do that with all the literals we get  
"true and ... and true"  
thus having proved that the entire conjunction is true.

* But what if the clause head \(L_0\) is not exactly identical to L?

  ➢ Unify \(L_0\) and L!!! 😊
General Resolution

- Resolution Principle

  The proof of the goal \( G \vdash P, L, Q \).
  
  if there exists a clause \( L_0 : \neg L_1, \ldots, L_n \) (\( n \geq 0 \))
  
  such that \( \sigma = \text{mgu}(L, L_0) \)
  
  can be reduced to proving \( \vdash P\sigma, L_1\sigma, \ldots, L_n\sigma, Q\sigma \).

- Informal “resolution algorithm”

  To prove the goal \( G \vdash P, L, Q \).
  
  select one literal in \( G \), say \( L \),
  
  select a copy of a clause \( L_0 : \neg L_1, \ldots, L_n \) (\( n \geq 0 \))
  
  such that there exists \( \sigma = \text{mgu}(L, L_0) \)
  
  apply \( \sigma \) to the clause \( L_0\sigma : \neg L_1\sigma, \ldots, L_n\sigma \)
  
  apply \( \sigma \) to the goal \( \vdash P\sigma, L_1\sigma, \ldots, L_n\sigma, Q\sigma \)
  
  replace \( L\sigma \) by the clause body \( \vdash P\sigma, L_1\sigma, \ldots, L_n\sigma, Q\sigma \).
General Resolution

- Resolution Principle

The proof of the goal $G \iff P, L, Q$.

if there exists a clause $L_0 \iff L_1, \ldots, L_n$ ($n \geq 0$)

such that $\sigma = \text{mgu}(L, L_0)$

can be reduced to proving $\iff P\sigma, L_1\sigma, \ldots, L_n\sigma, Q\sigma$.

- Graphical illustration of resolution by “derivation trees”

Initial goal $\rightarrow \iff P, L, Q$.

Copy of clause with renamed variables $\rightarrow L_0 \iff L_1, L_2, \ldots, L_n$.

Unifier of selected literal and clause head $\rightarrow \sigma = \text{mgu}(L, L_0)$.

Derived goal $\rightarrow \iff P\sigma, L_1\sigma, \ldots, L_n\sigma, Q\sigma$. 
Resolution reduces goals to subgoals

For we also say or and write “Goal₂ results from Goal₁ by resolution” “Goal₂ is derived from Goal₁” “Goal₁ is reducible to Goal₂” “Goal₁ |- Goal₂”

Goal / Goal₁  →  ?- P, L, Q.

Derivation / reduction step  →  

L₀ := L₁, L₂, ..., Lₙ

σ = mgu(L, L₀)

Subgoal / Goal₂  →  ?- Pσ, L₁σ, ..., Lₙσ, Qσ.
**SLD Resolution**

**General Resolution Principle**

Unrestricted selection functions

1. any literal of the goal
2. any clause of the predicate

**Problem**

- For a practical implementation we have to specify *which* literal / clause to select.
- Too complex selection criteria are inefficient.

**SLD Resolution**

Simple selection functions

1. First literal of the goal
2. First clause of the predicate

**Remember**

SLD-resolution works

1. Left-to-right for literals and
2. Top-to-bottom for clauses

Prolog uses SLD Resolution
SLD-Resolution Example: Program and Goal

- **Goal**

  ```
  ?- isGrandmaOf(maria,Granddaughter).
  ```

- **Program**

  ```
  isMotherOf(maria, klara).
  isMotherOf(maria, paul).
  isMotherOf(eva, anna).

  isMarriedTo(paul, eva).

  
  isGrandmaOf(G, E) :- isMotherOf(G, V), isFatherOf(V, E).
  isGrandmaOf(G, E) :- isMotherOf(G, M), isMotherOf(M, E).

  
  isFatherOf(V, K) :- isMarriedTo(V, M), isMotherOf(M,K).
  ... 
  ```
SLD-Resolution Example: Derivation

?- isGrandmaOf(maria, Granddaughter).

isGrandmaOf(G1, E1) :- isMotherOf(G1, V1), isFatherOf(V1, E1).
σ₁ = {G1 ← maria, E1 ← Granddaughter}

?- isMotherOf(maria, V1), isFatherOf(V1, Granddaughter).

isMotherOf(maria, paul).
σ₂ = {V1 ← paul}

?- isFatherOf(paul, Granddaughter).

isFatherOf(V2, K2) :- isMarriedTo(V2, M2), isMotherOf(M2, K2).
σ₃ = {V2 ← paul, K2 ← Granddaughter}

?- isMarriedTo(paul, M2), isMotherOf(M2, Granddaughter).

isMarriedTo(paul, eva).
σ₄ = {M2 ← eva}

?- isMotherOf(eva, Granddaughter).

isMotherOf(eva, anna).
σ₅ = {Granddaughter ← anna}
SLD-Resolution Example: Result?

?- isGrandmaOf(maria, Granddaughter).

\[ \sigma_1 = \{ G1 \leftarrow maria, E1 \leftarrow Granddaughter \} \]

\[ \sigma_2 = \{ V1 \leftarrow paul \} \]

\[ \sigma_3 = \{ V2 \leftarrow paul, K2 \leftarrow Granddaughter \} \]

\[ \sigma_4 = \{ M2 \leftarrow eva \} \]

\[ \sigma_5 = \{ Granddaughter \leftarrow anna \} \]

So what is the result?

\[ \rightarrow \text{ the last substitution?} \]

\[ \rightarrow \text{ the substitution(s) for the variable(s) of the goal?} \]
SLD-Resolution Example: Derivation with different variable bindings

?- isGrandmaOf(maria, Granddaughter).

isGrandmaOf(G1, E1) :- isMotherOf(G1, V1), isFatherOf(V1, E1).
\( \sigma_1 = \{ G1 \leftarrow \text{maria}, \text{Granddaughter} \leftarrow E1 \} \)

?- isMotherOf(maria, V1), isFatherOf(V1, E1).

isMotherOf(maria, paul).
\( \sigma_2 = \{ V1 \leftarrow \text{paul} \} \)

?- isFatherOf(paul, E1).

isFatherOf(V2, K2) :- isMarriedTo(V2, M2), isMotherOf(M2, K2).
\( \sigma_3 = \{ V2 \leftarrow \text{paul}, E1 \leftarrow K2 \} \)

?- isMarriedTo(paul, M2), isMotherOf(M2, K2).

isMarriedTo(paul, eva).
\( \sigma_4 = \{ M2 \leftarrow \text{eva} \} \)

?- isMotherOf(eva, K2).

isMotherOf(eva, anna).
\( \sigma_5 = \{ K2 \leftarrow \text{anna} \} \)
SLD-Resolution Example: Result revisited

?- isGrandmaOf(maria,Granddaughter).

\[ \sigma_1 = \{ G1 \leftarrow \text{maria}, \text{Granddaughter} \leftarrow E1 \} \]

\[ \sigma_2 = \{ V1 \leftarrow \text{paul} \} \]

\[ \sigma_3 = \{ V2 \leftarrow \text{paul}, E1 \leftarrow K2 \} \]

\[ \sigma_4 = \{ M2 \leftarrow \text{eva} \} \]

\[ \sigma_5 = \{ K2 \leftarrow \text{anna} \} \]

Observation

The result is not

→ the last substitution

→ the substitution(s) for the variable(s) of the goal

→ We need to „compose“ the substitutions!
The result is the composition of all substitutions computed along a derivation path …

\[ \sigma_1 = \{ G1 \leftarrow \text{maria}, \ \text{Granddaughter} \leftarrow \text{E1} \} \]
\[ \sigma_2 = \{ V1 \leftarrow \text{paul} \} \]
\[ \sigma_3 = \{ V2 \leftarrow \text{paul}, \ E1 \leftarrow \text{K2} \} \]
\[ \sigma_4 = \{ M2 \leftarrow \text{eva} \} \]
\[ \sigma_5 = \{ \text{K2} \leftarrow \text{anna} \} \]

\[ \sigma_1 \sigma_2 \sigma_3 \sigma_4 \sigma_5 = \{ G1 \leftarrow \text{maria}, \ \text{Granddaughter} \leftarrow \text{anna}, \ V1 \leftarrow \text{paul}, \ V2 \leftarrow \text{paul}, \ E1 \leftarrow \text{anna}, \ M2 \leftarrow \text{eva}, \ \text{K2} \leftarrow \text{anna} \} \]

… restricted to the bindings for variables from the initial goal.

\[ \sigma = \sigma_1 \sigma_2 \sigma_3 \sigma_4 \sigma_5 | \text{Vars} \left( \text{isGrandmaOf(maria,Granddaughter)} \right) \]
\[ = \sigma_1 \sigma_2 \sigma_3 \sigma_4 \sigma_5 | \{ \text{Granddaughter} \} \]
\[ = \{ \text{Granddaughter} \leftarrow \text{anna} \} \]
The result is the composition of all substitutions computed along a derivation path …

\[
\sigma_1 = \{G1 \leftarrow \text{maria}, \ E1 \leftarrow \text{Granddaughter}\} \\
\sigma_2 = \{V1 \leftarrow \text{paul}\} \\
\sigma_3 = \{V2 \leftarrow \text{paul}, \ K2 \leftarrow \text{Granddaughter}\} \\
\sigma_4 = \{M2 \leftarrow \text{eva}\} \\
\sigma_5 = \{\text{Granddaughter} \leftarrow \text{anna}\}
\]

\[
\sigma_1 \sigma_2 \sigma_3 \sigma_4 \sigma_5 = \{G1 \leftarrow \text{maria}, \ E1 \leftarrow \text{anna}, \ V1 \leftarrow \text{paul}, \ V2 \leftarrow \text{paul}, \ K2 \leftarrow \text{anna}, \ M2 \leftarrow \text{eva}, \ \text{Granddaughter} \leftarrow \text{anna}\}
\]

… restricted to the bindings for variables from the initial goal

\[
\sigma = \sigma_1 \sigma_2 \sigma_3 \sigma_4 \sigma_5 | \text{Vars( isGrandmaOf(maria,Granddaughter) )} \\
= \sigma_1 \sigma_2 \sigma_3 \sigma_4 \sigma_5 | \{ \text{Granddaughter} \} \\
= \{ \text{Granddaughter} \leftarrow \text{anna}\}
\]

Wouldn't that yield another result? Wouldn't that yield another result?

Different bindings than on the previous page!

Does that mean we get a different result???

No!

Same result substitution as on the previous page!
Resolution Result Defined

Let \( \sigma_1, \ldots, \sigma_n \) be the mgus computed along a successful derivation path for the goal \( G \) and let \( \text{Vars}(G) \) be the set of variables in \( G \).

- Then the result substitution is \( \sigma_1 \ldots \sigma_n \mid \text{Vars}(G) \)

- Informally: The result substitution for a successful derivation path (= a proof) of goal \( G \) is obtained by
  a) Composing all substitutions computed during the proof of the goal
  b) …and restricting the composition result to the variables of the goal.
Restriction Defined

Let $\sigma = \{V_1 \leftarrow t_1, \ldots, V_n \leftarrow t_n\}$ be a substitution and $V$ be a set of variables.

- Then $\sigma|V = \{V_i \leftarrow t_i \mid (V_i \in V) \land (V_i \leftarrow t_i \in \sigma)\}$

- Terminology: $\sigma|V$ is called the restriction of $\sigma$ to $V$

- Informally: The restriction $\sigma|V$ is obtained by eliminating from $\sigma$ all bindings for variables that are not in $V$
SLD-Resolution with Backtracking

OK, we've seen how resolution finds one answer. But how to find more answers?

→ Backtracking!
Derivation with Backtracking

We use \( f#1 \) as a shorthand notation for referring to the first clause of \( f \).

\[ \begin{align*}
\text{\( \checkmark \) } & f(a). \quad \checkmark g(a). \\
\Rightarrow & f(b). \quad \Rightarrow g(b). \quad \Rightarrow h(b). \\
\text{?-} & f(X), g(X), h(X).
\end{align*} \]

\[ \begin{align*}
\Rightarrow & f#1, X=a \quad \text{choicepoint: } f#2 \\
\text{?-} & g(a), h(a). \\
\Rightarrow & g#1 \quad \text{choicepoint: } g#2 \\
\text{?-} & h(a).
\end{align*} \]

- The subgoal \( h(a) \) fails because there is no clause whose head unifies with it.
- The interpreter backtracks to the last choicepoint and tries to redo the goal \( g(a) \) with clause \( g#2 \).
Derivation with Backtracking

✓ f(a). ✓ g(a).
✓ f(b). ✓ g(b). ✓ h(b).

?- f(X), g(X), h(X).

f#1, X=a
choicepoint: f#2

?- g(a), h(a).

- Redoing g(a) fails because there is no clause (at the choicepoint or after it) whose head unifies with g(a).
- The interpreter backtracks to the last “choicepoint” for f(X)
Derivation with Backtracking

\[ f(a). \quad g(a). \]
\[ \checkmark f(b). \quad \checkmark g(b). \quad \checkmark h(b). \]

\[ ?- f(X), g(X), h(X). \]

\[ f#2, X=b \quad \text{choicepoint: ---} \]

\[ ?- g(b), h(b). \]

\[ g#2 \quad \text{choicepoint: ---} \]

\[ ?- h(b). \]

\[ h#1 \quad \text{choicepoint: ---} \]

\[ ?- true. \]

- \( f(X) \) succeeds with substitution \( X=b \) and the derivation continues successfully (deriving the subgoal “true”).
- The interpreter reports the successful substitution.
SLD-Resolution with Backtracking: Summary

1. SLD-Resolution always selects
   - the leftmost (unused) literal in a goal as a candidate for being resolved
   - the topmost (unused) clause of a predicate definition as a candidate for resolving the current goal

2. If a clause’s head is not unifiable with the current goal the search proceeds immediately to the next clause

3. If a clause’s head is unifiable with the current goal
   - the goal is resolved with that clause
   - the interpreter remembers the next clause as a choicepoint

4. If no clause is found for a goal (= the goal fails), the interpreter undoes the current derivation up to the last choicepoint.

5. Then the search for a candidate clause (step 2) continues from that choicepoint
A goal is a box with four ports: call, succeed, redo, fail

A conjunction is a chain of connected boxes

- the “succeed” port is connected to the “call” port of the next goal
- the “fail” port is connected to the “redo” port of the previous goal
Box-Model of Backtracking

- **Subgoals of a clause** are boxes nested within the clause box, with outer and inner ports of the same kind connected:
  - clause’s call to first subgoal’s call
  - last subgoal’s succeed to clause’s succeed
  - clause’s redo to last subgoal’s redo
  - first subgoal’s fail to the fail of the clause
Viewing Backtracking in the Debugger (1)

?- gtrace, simplify_aexpr(a-a+b-b, Simple).

... for this goal.

... call the graphical tracer ...

goals without choice points
variable bindings in selected stack frame
reference to next choice point
goals with choice points
call of “built-in” predicate (has no choicepoint)
source code view of goal associated to selected stack frame
the only exception is “repeat”
Viewing Backtracking in the Debugger (2)

The debugger visualizes the port of the current goal according to the box model.
Recursion

- Prolog predicates may be defined recursively

- A predicate is recursive if one or more rules in its definition refer to itself.
  
  \[ \text{descendant}(C,X) :\neg \text{child}(C,X). \]
  
  \[ \text{descendant}(C,X) :\neg \text{child}(C,D), \text{descendant}(D,X). \]

- What does the descendant/2 definition mean?
  
  1. C is a child of X \implies C is a descendant of X
  2. C is a child of D \text{ and } D \text{ is a descendant of } X \implies C \text{ is a descendant of } X
child(martha, charlotte).
child(charlotte, caroline).
child(caroline, laura).
child(laura, rose).

descend(X, Y):- child(X, Y).
descend(X, Y):-
    child(X, Z), descend(Z, Y).

?- descend(martha, laura)
yes
Example: Derivation and Recursion

- A program (List membership: Arg1 is a member of the list Arg2)

```prolog
member(X, [X|_]). % clause #1
member(X, [_|R]) :- member(X, R). % clause #2
```

- A query, its successful substitutions …

```prolog
?- member(E, [a,b,c]).
E = a ; E = b ; E = c ; fail.
```

- … and its derivation tree

```
?- member(E, [a,b,c])
member(X1, [X1|_]). {X1←E, X1←a, E←a}  

?- member(E, [b,c])
member(X2, [_|R2]) :- member(X2, R2).  
{X2←E, R2←[b,c],}  

?- member(E, [c])
member(X3, [X3|_]). {X3←E, X3←b, E←b}  

member(X4, [_|R4]) :- member(X4, R4).  
{X4←E, R4←[c]}  

?- member(E, [abc])
member(X5, [X5|_]). {X5←E, X5←b, E←c}  
```

Alternative derivations for
?- member(E, [abc])
Recursion: Successor

- Suppose we want to express that
  - 0 is a numeral
  - If \( X \) is a numeral, then \( \text{succ}(X) \) is a numeral

```
numeral(0).
numeral(succ(X)) :- numeral(X).
```

- Let's see how this behaves:

```
?- numeral(X).
X = 0 ;
X = succ(0) ;
X = succ(succ(0)) ;
X = succ(succ(succ(0))) ;
...```
Negation

OK, we’ve seen how to prove or conclude what is true. But what about negation?

→ Closed world assumption
→ Negation as failure
→ “Unsafe negation” versus existential variables
Closed World Assumption

- We cannot prove that something is false.
- We can only show that we cannot prove its contrary.

```prolog
isFatherOf(kurt,peter).

?- isFatherOf(adam,cain).
no.  \(\Leftarrow\) means: we cannot prove that “isFatherOf(adam,cain)” is true.
```

- If we **assume** that everything that is true is entailed by the program, we may then **conclude** that what is not entailed / provable is not true.
- This **assumption** is known as the “**Closed World assumption**” (CWA)
- The **conclusion** is known as “**Negation by Failure**” (NF)

```prolog
?- not( isFatherOf(adam,cain) ).
yes.
\(\Leftarrow\) means: we conclude that “not(isFatherOf(adam,cain))” is true because we cannot prove that “isFatherOf(adam,cain)” is true.
```
Negation with Unbound Variables (1)

- Deductive databases consider all variables to be universally quantified.
- However, the set of values for $X$ for which isFatherOf/2 fails is infinite and unknown because it consists of everything that is not represented in the program.
- So it is impossible to list all these values!
- Therefore, the above negated query with universal quantification is unsafe.

isFatherOf(kurt, peter).

?- $\forall X$.isFatherOf(adam, X).
no.

?- $\forall X$.not(isFatherOf(adam, X)).
← unsafe, infinite result set!
Negation with Unbound Variables (2)

- **Prolog** treats free variables in negated goals as existentially quantified. So it does not need to list all possible values of $X$.

- It shows that **there is some value for which the goal $G$ fails**, by showing that $G$ does not succeed for any value

$$\exists x. \neg G \iff \neg \forall x. G$$

- This is precisely negation by failure!
Negation with Unbound Variables (3)

Existential variables can also occur in clause bodies:

- The clause
  
  | single(X) :- human(X), not(married(X,Y)). |
  | \forall X. human(X) \land \neg(\exists Y. married(X,Y)) \rightarrow single(X) |

- means

Take care: Changing the order of negated subgoals can change the (operational) semantics of the clause. If X is unbound in the call

- the clause
  
  | single(X) :- not(married(X,Y)), human(X). |
  | single(X) :- not(married(X1,Y)), human(X). |

- is the same as

- Both mean

  | \forall X. human(X) \land \neg(\exists X1. \exists Y. married(X1,Y)) \rightarrow single(X). |

Remember: Free variables in negated goals are existentially quantified.

- They do not “return bindings” outside of the scope of the negation.
- They are different from variables outside of the negation that accidentally have the same name.
Negation with Unbound Variables (4)

Explanations for the previous slide

- The clause
  
  \[ \text{single}(X) :\neg \text{human}(X), \neg \text{married}(X,Y). \]
  
  means
  
  \[ \forall X. \text{human}(X) \land \neg \exists Y. \text{married}(X,Y) \rightarrow \text{single}(X) \]
  
  because \( X \) is already bound by \( \text{human}(X) \) when the negation is entered.

- The clause
  
  \[ \text{single}(X) : \neg \text{married}(X,Y), \text{human}(X). \]
  
  is the same as
  
  \[ \text{single}(X) : \neg \text{married}(X,Y), \text{human}(X). \]
  
  Both mean
  
  \[ \forall X. \text{human}(X) \land \neg \exists X1. \exists Y. \text{married}(X1,Y) \rightarrow \text{single}(X). \]
  
  because the yellow \( X \) in the first clause is not bound when the negation is reached. So it is existentially quantified, whereas the blue \( X \) is universally quantified. Thus both are actually different variables since the same variable cannot be quantified differently in the same scope (remember the “disjoint variables” normalization step).
Eliminate accidentally equal names!

Remember: Free variables in negated goals are existentially quantified.
- They do not “return bindings” outside of the scope of the negation.
- They are different from variables outside of the negation that accidentally have the same name.

```
nestedneg1(Y) :- q(Y), not(p(X,Y), not(f(X,Z), g(Z))), q(X, Z), q(X). % INST

nestedneg1(Y) :- q(Y), not(p(X,Y), not(f(X,Z), g(Z))), q(X, Z1), q(X1). % INST
```

Eliminate accidentally equal names
A Test

• Predict what this program does!

  f(1,a).
  f(2,b).
  f(2,c).
  f(4,c).
  q(1).
  q(2).
  q(3).

  negation(X) :-
      not(
          ( f(X,c),
              output(X),
              g(X)
          ),
          q(X).
      )

  output(X) :-
      format('Found f(~a,c) ', [X]).
      output(X) :-
          format('but no g(~a)~n', [X]).

• This is what it does (try it out):

  ?- negation(X).
  Found f(2,c) but no g(2)
  Found f(4,c) but no g(4)
  X=1 ;
  X=2 ;
  X=3 ;
  fail.

• Homework:

  If you don’t understand the result reread the slides about
  negation (and also those about backtracking if you do
  not understand why output/1 has two clauses).

format/2 is a built-in predicate. It outputs its first argument to the
console, replacing the control elements ~a or ~w (in the order of
their appearance) by the values of the respective list elements
from the second argument. ~a stands for an element that must
be an atom. ~w stands for an arbitrary term. ~n is a newline.
Incompleteness of SLD-Resolution

Can we prove truth or falsity of every goal?

→ No, unfortunately!
Incompleteness of SLD-Resolution

- **Provability**
  - If a goal can be reduced to the empty subgoal then the goal is **provable**.

- **Undecidability**
  - There is no automated proof system that always answers **yes** if a goal is provable from the available clauses and answers **no** otherwise.
  - Prolog answers **yes**, **no** or **does not terminate**.
The evaluation strategy of Prolog is incomplete.

- Because of non-terminating derivations, Prolog sometimes only derives a subset of the logical consequences of a program.

**Example**

- \( r, p, \) and \( q \) are logical consequences of this program

However, Prolog’s evaluation strategy cannot derive them. It loops indefinitely:
Practical Implications

- Need to understand both semantics
  - **The model-based (declarative) semantics** is the “reference”
    - We can apply bottom-up fixpoint iteration to understand the set of logical consequences of our programs
  - **The proof-based (operational) semantics** is the one Prolog uses to prove that a goal is among the logical consequences
    - SLD-derivations can get stuck in infinite loops, missing some correct results

- Need to understand when these semantics differ
  - When do Prolog programs fail to terminate?
    - Order of goals and clauses
    - Recursion and “growing” function terms
    - Recursion and loops in data
  - Which other problems could prevent the operational semantics match the declarative semantics?
    - The cut!
    - Non-logical features
    - …
General Principles

- Try to match both semantics!
  - Your programs will be more easy to understand and maintain

- Write programs with the model-based semantics in mind!
  - If they do not behave as intended change them so that they do!
Two different ways to give meaning to logic programs

<table>
<thead>
<tr>
<th>Operational Semantics</th>
<th>Declarative Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>● Proof-based approach</td>
<td>● Model-based semantics</td>
</tr>
<tr>
<td>◆ Algorithm to find a proof</td>
<td>◆ Mathematical structure</td>
</tr>
<tr>
<td>◆ Refutation proof using SLD resolution</td>
<td>◆ Herbrand interpretations and Herbrand models</td>
</tr>
<tr>
<td>◆ Basic step: Derivation</td>
<td>◆ Basic step: Entailment (Logical consequence)</td>
</tr>
<tr>
<td>● To prove a goal prove each of its subgoals</td>
<td>● A formula is true if it is a logical consequence of the program</td>
</tr>
<tr>
<td>● Algorithm = Logic + Control</td>
<td>● Algorithm = Logic + Control</td>
</tr>
<tr>
<td>◆ Logic = Clauses</td>
<td>◆ Logic = Clauses</td>
</tr>
<tr>
<td>◆ Control = Top-down resolution process</td>
<td>◆ Control = Bottom-up fixpoint iteration</td>
</tr>
</tbody>
</table>
Chapter Summary

- **Proof-theoretic Semantics**
  - Refutation Proofs
  - Resolution Principle
  - SLD-Resolution
  - Horn Clauses
  - Unification
  - Incompleteness / non-termination

- **Negation as Failure**
  - Closed World Assumption
  - Negation as Failure
  - Existential Variables