Lecture 9: Foundations of Logic Programming

Marin Mircea

April 2021

Contents

1 Notions specific to Logic Programming

- An **atom** is a formula $p(t_1, \ldots, t_n)$ where p is a predicate symbol with arity n and $t_1, \ldots, t_n$ are terms.
- A **literal** is either an atom or the negation of atom. A literal is positive if it is an atom, and negative otherwise.
- A **query** is a formula $\bigwedge_{i=1}^{p} A_i$

  where $A_1, \ldots, A_p$ are atoms.
- A **clause** is a formula $H \leftarrow B$

  where H is atom and $B = \bigwedge_{i=1}^{p} B_i$ is a (possibly empty) conjunction of atoms. H is called the **head**, and B is the **body** of the clause.
- A **program** is a finite set of clauses.

Programs and queries have two interpretations: **declarative** and **procedural**.

- A query $\bigwedge_{i=1}^{p} A_i$ with variables $X_1, \ldots, X_n$ has the declarative interpretation "There exist $X_1, \ldots, X_n$ such that A_1 and ... and A_p hold" and the procedural interpretation "Find the values of $X_1, \ldots, X_n$ for which the formulas A_1, ..., A_p hold simultaneusly."
- A clause $H \leftarrow B$ with variables $X_1, \ldots, X_n$ has the declarative interpretation "for all $X_1, \ldots, X_n$, H holds if B holds" and the procedural interpretation "To prove that H holds, it is sufficient to prove that B holds."

We say that queries and clauses of this kind are **positive** because all their components are atoms, which are positive literals. Later on, we will see that Prolog can work with more general kinds of queries and clauses.

Prolog uses a simplified syntax to write programs and queries:

- A clause $H \leftarrow \bigwedge_{i=1}^{p} B_i$ is written

```
H:-B1,...,Bp.
```

  The variables are recognized because they start with an uppercase letter, and all of them are universally quantified. Clauses $H \leftarrow \bigwedge_{i=1}^{0} B_i$ with empty body are called **facts** and are written

```
H.
```

- A query $\bigwedge_{i=1}^{p} A_i$ is written after the `?-` prompt of Prolog, as follows:

```
?- A1,...,Ap.
```

where the variables are recognized because they start with an uppercase letter, and all of them are existentially quantified.

Prolog programs are saved in text files with extension `.pl`

For example, the following program contains some facts about parent-child relations and clauses that define the relations

`parent(X,Y)` with intended reading "`X` is the parent of `Y`", and
`siblings(X,Y)` with intended reading "`X` and `Y` are siblings".

```
father(john,jack).
father(john,bob).
mother(mary,jack).
mother(ana,ray).
parent(X,Y) :- father(X,Y).
parent(X,Y) :- mother(X,Y).
siblings(X,Y) :- parent(Z,X),parent(Z,Y),X \= Y.
```

Note the declarative interpretation of the last clause: "for all `X`, `Y`, `Z`: `X` and `Y` are siblings if `Z` is the parent of `X`, `Z` is the parent of `Y`, and `X` and `Y` are different." Although all variables in a clause are universally quantified, it is often more natural to read variables in the conditions that are not in the conclusion as existentially quantified with the body of the rule as their scope. For example, the following interpretation is equivalent with the previous one: "`X` and `Y` are siblings id there is a `Z` who is parent of both `X` and Y, and `X` and `Y` are different."

A possible query could be

```
?- parent(X,jack).
X = john ;
X = mary.
```

with the declarative reading "Prove that there exists `X` which is the parent of `jack`." Prolog uses a proof method called SLDNF-resolution that proves queries in a constructive way. "Constructive" means here that the values of the variables are in effect computed. In the previous example, the answer is not a simple `"yes"` but a constructive one, like "`X = john`" or "`X = mary`".

The goal of this lecture is to explain the computational mechanism of Prolog for the kinds of programs and queries defined before.

The theoretical foundations of logic programming rely on the following concepts:

Unification: the basic mechanism which assigns values to variables in logic programming.

Logic programs and queries. A logic program is the knowledge base that represents what we know about the world, and a query is a formula that we want to prove from a program in a constructive way.

SLD-resolution: a proof method built upon unification that allows us to prove queries from a program in a constructive way.

Semantics, which gives a meaning logic programs and queries. The relationships between the proof method of SLD-resolution and the semantics (of programs and queries) are clarified by the so-called *soundness* and *completeness* properties of SLD-resolution.

2 Unification

In logic programming, variables represent unknown values, like in mathematics. The values assigned to variables are terms, and the assignment of terms to variables is by means of substitutions called **most general unifiers**. The process of computing most general unifiers is called **unification**.

Historical note. Unification was defined by J.A. Robinson [Rob65] in the context of automated teorem proving. Its use for computing is due to R. Kowalski [Kow74].

From now on we consider terms from $T(F, \mathcal{X})$ and write $Var(t)$ for the set of variables occurring in a term t. If $Var(t) = \emptyset$ we say that t is a *ground term.*

A **substitution** is a mapping θ from a finite set of variables $Dom(\theta) \subseteq \mathcal{X}$ to terms, which assigns to each variable $X \in Dom(\theta)$ a term t different from X. In this lecture, we write a substitution θ as $\{X_1 \to t_1, \ldots, X_n \to t_n\}$ where

- $X_1, \ldots, X_n$ are the different variables in its domain $Dom(\theta)$,
- for every $1 \leq i \leq n$, t_i is the term $\theta(X_i)$. By definition, $t_i \neq X_i$.

Every pair $X_i \to t_i$ is called a *binding* of θ, and we say that X_i is *bound to* t_i. We denote by $Range(\theta)$ the set of terms $\{t_1, \ldots, t_n\}$, by $Ran(\theta)$ the set of variables with occurrences in $t_1, \ldots, t_n$, and let $Var(\theta) = Dom(\theta) \cup Ran(\theta)$. When $n = 0$, θ becomes the empty mapping, which is called *empty substitution* and is denoted by ϵ.

Consider a substitution $\theta = \{X_1 \to t_1, \ldots, X_n \to t_n\}$. If all terms $t_1, \ldots, t_n$ are ground then θ is called *ground substitution*, and if all $t_1, \ldots, t_n$ are variables then θ is called a *pure variable substitution.* When $Dom(\theta) = Range(\theta)$ then θ is a bijective mapping from its domain to itself, and is called a *renaming.* For example, the substitutions ϵ and $\{X \to Y, Y \to Z, Z \to X\}$ are renamings, but $\{X \to Y, Y \to Z\}$ is not a renaming.

Substitutions can be applied to terms. The result of applying a substitution θ to a term t, written as $t\theta$, is the result of *simultaneous* replacement of each occurrence in t of a variable from $Dom(\theta)$ by the corresponding term in

$Range(\theta)$. Formally, this operation is defined by induction on the structure of the term t:

- if $X \in \mathcal{X}$ then $X\theta = \begin{cases} \theta(X) & \text{if } X \in Dom(\theta), \\ X & \text{otherwise.} \end{cases}$
- $f(t_1, \dots, t_n)\theta = f(t_1\theta, \dots, t_n\theta)$.

  In particular, if t is a constant c then $t\theta = c\theta = c = t$.

The term $t\theta$ is called an **instance** of t. An instance is called **ground** if it has no variables. If θ is a renaming, then $s\theta$ is a **variant** of s. Finally, a term s is called **more general** than another term t is t is an instance of s. For example,

1) $f(Y, X)$ is a variant of $f(X, Y)$ since $f(Y, X) = f(X, Y)\{X \to Y, Y \to X\}$.
2) $f(X, Z)$ is a variant of $f(X, Y)$ since $f(X, Z) = f(X, Y)\{Y \to Z, Z \to Y\}$. Note that the binding $Z \to Y$ had to be added to make the substitution a renaming.
3) $f(X, X)$ is not a variant of $f(X, Y)$.

Next, we define the *composition* of substitutions θ and η written as $\theta\eta$, as follows:

$$\theta\eta(X) = (X\theta)\eta \quad \text{for all } X \in \mathcal{X}.$$

In other words, $\theta\eta$ assigns to a variable X the term obtained by applying the substitution η to the term $X\theta$. Clearly, $Dom(\theta\eta) \subseteq Dom(\theta) \cup Dom(\eta)$.

A constructive definition of composition of substitutions

The definition of composition is equivalent with the following, which is easier to use in computations: If $\theta = \{X_1 \to t_1, \dots, X_n \to t_n\}$ and $\eta = \{Y_1 \to s_1, \dots, Y_m \to s_m\}$ then $\theta\eta$ is the result of the following computation:

1. Remove from the sequence

$$X_1 \to t_1\eta, \dots, X_n \to t_n\eta, Y_1 \to s_1, \dots, Y_m \to s_m$$

   the bindings $X_i \to t_i\eta$ for which $t_i\eta = X_i$ and the bindings $Y_j \to s_j$ for which $Y_j \in \{X_1, \dots, X_n\}$
2. Form from the resulting sequence of bindings a substitution.

For example, if $\theta = \{U \to Z, X \to 3, Y \to f(X, 1)\}$ and $\eta = \{X \to 4, Z \to U\}$ then $\theta\eta = \{X \to 3, Y \to f(3, 1), Z \to U\}$.

Substitution composition is **associative**: If θ, η, γ are substitutions and s is a term, then $(\theta\eta)\gamma = \theta(\eta\gamma)$ and $(s\theta)\eta = s(\theta\eta)$.

Of special interest are the renamings and the variants. Renamings have the following remarkable properties:

1. For every renaming θ there exists exactly one substitution θ^{-1} such that $\theta\theta^{-1} = \theta^{-1}\theta = \epsilon$. Moreover, θ^{-1} is also a renaming.

2. If $\theta\eta = \epsilon$ then θ and η are renamings.

Variants have the following remarkable properties:

1. s is a variant of t if and only if t is a variant of s.

2. If s is a variant of t then there is a renaming θ such that $s = t\theta$ and $Var(\theta) \subseteq Var(s) \cup Var(t)$.

For example, the renaming $\theta = \{X \to Y, Y \to Z, Z \to X\}$ has the inverse $\theta^{-1} = \{X \to Z, Y \to X, Z \to Y\}$. The pure variable substitution $\eta = \{X \to Y\}$ is not a renaming, and there is no substitution η^{-1} such that $\eta\eta^{-1} = \eta^{-1}\eta = \epsilon$.

2.1 Unifiers

Informally, unification is the process of making terms identical by means of certain substitutions. For example, the terms $f(a, Y, Z)$ and $f(X, b, Z)$ can be made identical by applying to them the substitution $\{X \to a, Y \to b\}$: both sides then become $f(a, b, Z)$. But the substitution $\{X \to a, Y \to b, Z \to a\}$ also makes these two terms identical. Such substitutions are called **unifiers**. The first unifier is preferable because it is more general — the second one is a special case of the first one. The following definition clarifies this difference.

Let θ and τ be substitutions. We say that θ is **more general than** τ if we have $\tau = \theta\eta$ for some η.

For example, $\{X \to a, Y \to b\}$ is more general than $\{X \to a, Y \to b, Z \to a\}$ because $\{X \to a, Y \to b, Z \to a\} = \{X \to a, Y \to b\}\{Z \to a\}$.

From now on, we write $\theta \preceq \eta$ if θ is more general than η. Note that '$\preceq$' is

1. reflexive: $\theta \preceq \theta$ because $\theta = \theta\epsilon$,

2. transitive: if $\theta_1 \preceq \theta_2$ and $\theta_2 \preceq \theta_3$ then there exist η_1, η_2 such that $\theta_2 = \theta_1\eta_1$ and $\theta_3 = \theta_2\eta_2$. Thus $\theta_1 \preceq \theta_3$ because $\theta_3 = (\theta_1\eta_1)\eta_2 = \theta_1(\eta_1\eta_2)$,

but is not antisymmetric because, for example, $\theta = \{X \to Y, Y \to X\}$ has the following properties: $\theta \neq \epsilon$, $\epsilon \preceq \theta$ because $\epsilon\theta = \theta$ and $\theta \preceq \epsilon$ because $\epsilon = \theta\theta$. It can be shown that the following lemma holds.

RENAMING LEMMA. $\theta \preceq \eta$ and $\eta \preceq \theta$ if and only if $\eta = \theta\gamma$ for some renaming γ with $Var(\gamma) \subseteq Var(\theta) \cup Var(\eta)$.

The following are the key notions of this section.

Let $s, t \in T(F, \mathcal{X})$. A substitution θ is called

- a **unifier** of s and t is $s\theta = t\theta$. If a unifier of s and t exists we say that s and t are **unifiable**.
- a **most general unifier** of s and t (**mgu**) if it is a unifier of s and t, and it is more general than all unifiers of s and t.
- **strong** mgu of s and t if for all unifiers η of s and t we have $\eta = \theta\eta$.

Intuitively, an mgu is a substitution which makes two terms equal but which does it in a "most general way", without unnecessary bindings. So θ is an mgu if every other unifier η is of the form $\eta = \theta\gamma$ for some γ. An mgu γ is strong if for every unifier η, the substitution γ for which $\eta = \theta\gamma$ holds can be always chosen to be η itself.

The problem of deciding whether two terms are unifiable is called the **unification problem**. We solve this problem by providing an algorithm that terminates with failure if the terms are not unifiable and that otherwise produces a strong mgu. In general, two terms may be not unifiable for two reasons:

1. Two terms starting with a different function symbol can not unify. For example, $f(g(X, a), X)$ and $f(g(X, b), b)$ are not unifiable because the corresponding red subterms start with different function symbols.
2. A variable X can not be unified with a term $t \neq X$ with $X \in Var(t)$. For example, $g(X, a)$ and $g(f(X), a)$ are not unifiable because the corresponding red subterms can not be unified for this reason.

   In the literature, this reason is called **occur-check failure**.

Each possibility can occur at some "inner level" of the considered two terms.

2.2 The Martelli-Montanari algorithm

This algorithm was proposed by Martelli and Montanari [MM82] to solve an apparently more general problem:

Given a set of term equations $S = \{s_1 = t_1, \ldots, s_n = t_n\}$

Compute An mgu of this set, that is, a substitution θ such that

1. $s_i\theta = t_i\theta$ for all $1 \leq i \leq n$. Such a substitution θ is called **unifier** of the system S.
2. θ is more general than all other unifiers of S.

The algorithm works by nondeterministically choosing an equation of a form below from the set of equations, and performing the associated action:

1) $f(s_1, \ldots, s_n) = f(t_1, \ldots, t_n)$.

   Replace it by the equations $s_1 = t_1, \ldots, s_n = t_n$.

2) $f(s_1, \ldots, s_n) = g(t_1, \ldots, t_m)$ with $f \neq g$.
   Halt with failure.

3) $X = X$.
   Delete the equation.

4) $t = X$ where $t \notin \mathcal{X}$.
   Replace it with $X = t$.

5) $X = t$ where $X \notin Var(t)$ and X occurs elsewhere.
   Perform the substitution $\{X \to t\}$ on all other equations,

6) $X = t$ where $X \in Var(t)$ and $X \neq t$.
   Halt with failure.

To use this algorithm for unifying two terms s, t we activate it with the singleton set $\{s = t\}$. It is well known that the algorithm always terminates. Moreover,

- If the algorithm terminates with failure, the terms s and t are not unifiable.
- Otherwise, the algorithm stops with a system of equations of the form $\{X_1 = t_1, \ldots, X_n = t_n\}$ where the X_is are distinct variables and none of them occurs in a term t_j.

  In this case, $\{X_1 \to t_1, \ldots, X_n \to t_n\}$ is a strong mgu of s and t.

Examples:

1. $f(g(X, a), X)$ and $f(g(X, b), b)$ are not unifiable because

$$\{\underline{f(g(X,a),X) = f(g(X,b),b)}\} \Rightarrow \{\underline{g(X,a) = g(X,b)}, X = b\} \Rightarrow$$
$$\{X = X, \underline{a = b}, X = b\} \Rightarrow \text{failure}.$$

2. $\{X \to g(a), Y \to b, Z \to h(g(a))\}$ is a strong mgu of $k(Z, f(X, b, Z))$ and $k(h(X), f(g(a), Y, Z))$ because

$$\{\underline{k(Z, f(X,b,Z)) = k(h(X), f(g(a),Y,Z))}\} \Rightarrow$$
$$\{Z = h(X), \underline{f(X,b,Z) = f(g(a),Y,Z)}\} \Rightarrow$$
$$\{Z = h(X), X = g(a), b = Y, \underline{Z = Z}\} \Rightarrow \{Z = h(X), X = g(a), \underline{b = Y}\} \Rightarrow$$
$$\{Z = h(X), \underline{X = g(a)}, Y = b\} \Rightarrow \{Z = h(g(a)), X = g(a), Y = b\}.$$

We conclude this section by mentioning two important properties of strong mgus:

1. Any strong mgu θ is **idempotent**, that is, $\theta\theta = \theta$.
2. In general, a substitution θ is idempotent iff $Dom(\theta) \cap Ran(\theta) = \emptyset$.

3 SLD-resolution

The computation process within the logic programming framework can be explained as follows. A program P can be viewed as a set of axioms and a query Q as a request to find an instance $Q\theta$ of it which follows from P. A successful computation yields such a θ and can be viewed as a proof of $Q\theta$ from P.

The computation is a sequence of basic steps. Each basic step selects an atom A_i in the current query $\bigwedge_{i=1}^{p} A_i$ and a clause $H \leftarrow B$ in the program P. If A_i **unifies** with H, then the next query is obtained by **replacing** A_i by the clause body B and by applying to the outcome an mgu of A_i and H. The computation terminates successfully when the empty query is produced. θ is then the composition of the mgus used.

Thus logic programs compute through a combination of two mechanisms: **replacement** and **unification**. To understand better various fine points of this computation, let's introduce a few auxiliary notions.

Let P be a program, $Q = \bigwedge_{A \in \mathcal{A}} A$ a non-empty query, and $H \leftarrow B$ a variant of $c \in P$ such that $Var(H \leftarrow B) \cap Var(Q) = \emptyset$, and θ is an mgu of H and $A_0 \in \mathcal{A}$.

The query

$$Q' = \left(B \wedge \bigwedge_{A \in \mathcal{A} - \{A_0\}} A \right) \theta$$

is called an **SLD-resolvent** of Q and c w.r.t. A_0, with mgu θ, and we write $Q \Rightarrow_{\theta,c} Q'$ to express this fact. A_0 is called the **selected atom**, c is called the **input clause**, and this relation is called an **SLD-derivation step**.

An SLD-derivation step can also be presented in the form of a rule of deduction:

$$\frac{\bigwedge_{A \in \mathcal{A}} A \quad H \leftarrow B}{\left(B \wedge \bigwedge_{A \in \mathcal{A} - \{A_0\}} A \right) \theta}$$

where $(H \leftarrow B)$ is a variant of a clause $c \in P$ such that $Var(H \leftarrow B) \cap Var(Q) = \emptyset$, and θ is an mgu of H and $A_0 \in \mathcal{A}$. This rule of deduction is called **SLD-resolution**. It was introduced by R.A. Kowalski in logic programming [Kow74] to derive conclusions by means of backward reasoning.[1]

Thus a resolvent of a non-empty query and a clause is obtained by the following successive steps:

Selection of an atom A in the query.

Renaming, if necessary, a clause chosen from the program.

[1] Backward reasoning, or *backward chaining*, is an inference method used in automated theorem proving that works backwards, from conclusion to hypotheses from which it can follow.

Instantiation of the query and the clause by an mgu of the selected atom and the head of the clause.

Replacement of the instance of the selected atom by the instance of the body of the clause.

By iterating SLD-derivation steps we obtain an SLD-derivation. Formally, if P is a program and Q_0 a query, the an **SLD-derivation** of $P \cup \{Q_0\}$ is a maximal sequence

$$Q_0 \Rightarrow_{\theta_1, c_1} Q_1 \cdots Q_n \Rightarrow_{\theta_{n+1}, c_{n+1}} Q_{n+1} \cdots$$

of SLD-derivation steps, and each step satisfies the following additional technical condition:

Standardization apart: each variant $H_i \leftarrow B_i$ of an input c_i has variables which did not occur in previous SLD-derivation steps.

Intuitively, this means that at each step of an SLD-derivation the variables of the input clauses should be fresh.

A few more definitions will be helpful.

1. A clause is called **applicable** to an atom if a variant of its head unifies with the atom.
2. The **length** of an SLD-derivation is the number of SLD-derivation steps used in it. So an SLD-derivation of length 0 consists of a single query Q such that either Q is empty or no clause of the program is applicable to its selected atom.
3. A derivation $\xi := Q_0 \Rightarrow_{\theta_1} Q_1 \cdots \Rightarrow_{\theta_n} Q_n$ is either

   successful if $Q_n =$ `true` is the empty conjunction.

   failed if Q_n is not the empty conjunction and no clause of P is applicable to the selected atom of Q_n.

If the derivation is successful then the restriction $(\theta_1 \cdots \theta_n)|_{Var(Q)}$ of $\theta_1 \cdots \theta_n$ to the variables of Q is called a **computed answer substitution** (c.a.s. in short) of Q and $Q\theta_1 \cdots \theta_n$ is called a **computed instance** of Q.

For example, let's consider terms built out of the constant `0` ("zero") by means of the unary function symbol `s` ("successor"). We call such terms numerals. The following program defines the predicate `sum(X,Y,Z)` which holds if `Z` is the sum of numerals `X` and `Y`:

```
sum(X,0,X) ← true.                  % c1
sum(X,s(Y),s(Z)) ← sum(X,Y,Z).      % c2
```

The query `sum(s(s(0)),s(s(0)),Z)` asks for the value of `Z` which is the sum of numerals `s(s(0))` and `s(s(0))`. The SLD-derivation of this query is

$$\begin{aligned}\mathtt{sum(s(s(0)),s(s(0)),Z)} &\Rightarrow_{\theta_1=\{\mathtt{X_1}\to\mathtt{s(s(0))},\mathtt{Y_1}\to\mathtt{s(0)},\mathtt{Z}\to\mathtt{s(Z_1)}\},\mathtt{c_2}} \mathtt{sum(s(s(0)),s(0),Z_1)}\\ &\Rightarrow_{\theta_2=\{\mathtt{X_2}\to\mathtt{s(s(0))},\mathtt{Y_2}\to\mathtt{0},\mathtt{Z_1}\to\mathtt{s(Z_2)}\},c_2} \mathtt{sum(s(s(0)),0,Z_2)}\\ &\Rightarrow_{\theta_3=\{\mathtt{X_3}\to\mathtt{s(s(0))},\mathtt{Z_2}\to\mathtt{s(s(0))}\},c_1} \mathtt{true.}\end{aligned}$$

where the variants used in these steps are $\mathtt{sum(X_1, s(Y_1), s(Z_1))} \leftarrow \mathtt{sum(X_1, Y_1, Z_1)}$, $\mathtt{sum(X_2, s(Y_2), s(Z_2))} \leftarrow \mathtt{sum(X_2, Y_2, Z_2)}$, and $\mathtt{sum(X_3, 0, X_3)} \leftarrow \mathtt{true}$.

The corresponding computed answer substitution is

$$\theta_1\theta_2\theta_3|_{\{\mathtt{Z}\}} = \{\mathtt{Z} \to \mathtt{s(s(s(s(0))))}\}.$$

3.1 Selection rules

According to the definition of an SLD-derivation, the following four choices are made in each SLD-derivation step:

(A) choice of the selected atom in the considered query,
(B) choice of the program clause applicable to the selected atom,
(C) choice of the renaming of the program clause used,
(D) choice of the mgu.

Here we discuss the impact of choice (A): the selection of an atom in a query.

Let $INIT$ be the set of initial fragments of SLD-derivations in which the last query is non-empty. A **selection rule** is a function $\mathcal{R}$ which, when applied to an element of $INIT$, yields an occurrence of an atom in its last query.

Given a selection rule $\mathcal{R}$, we say that an SLD-derivation ξ is via $\mathcal{R}$ if all choices of the selected atoms in ξ are performed according to $\mathcal{R}$. That is, for each initial fragment $\xi^<$ of ξ ending with a non-empty query Q, $\mathcal{R}(\xi^<)$ is the selected atom of Q.

Remark. Every SLD-derivation is via a selection rule. The name **SLD** comes from **S**election rule driven **L**inear resolution for **D**efinite clauses.

Examples of selection rules could be:

1. always choose the leftmost atom. This selection rule is used by the implementations of Prolog.

2. always choose the rightmost atom.

3. choose the leftmost atom at even SLD-derivation steps, and the last atom at odd SLD-derivation steps.

When searching for all computed answers of a query Q, we construct all SLD-derivations via a selection rule $\mathcal{R}$, with the aim of generating the empty query. All these derivations form a tree-like *search space*, called SLD-tree.

An **SLD-tree** for $P \cup \{Q\}$ via a selection rule $\mathcal{R}$ is a tree such that

- Its branches are SLD-derivations of $P \cup \{Q\}$ via $\mathcal{R}$.
- Every node Q with selected atom A has exactly one descendant for every clause c from P which is applicable to A. This descendant is a resolvent of Q and c with respect to A.

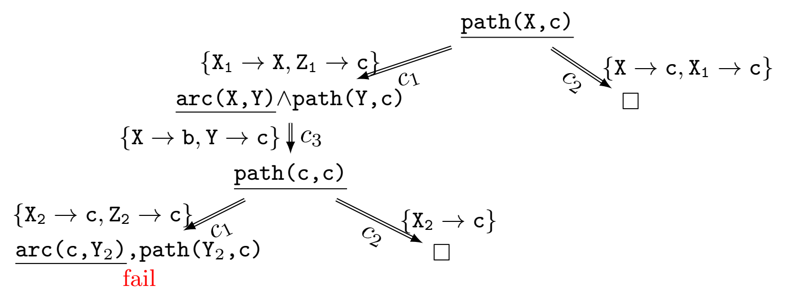
We call an SLD-tree **successful** if it contains the empty query. We call an SLD-tree **finitely failed** if it is finite and not successful.

To illustrate, consider the logic program P below

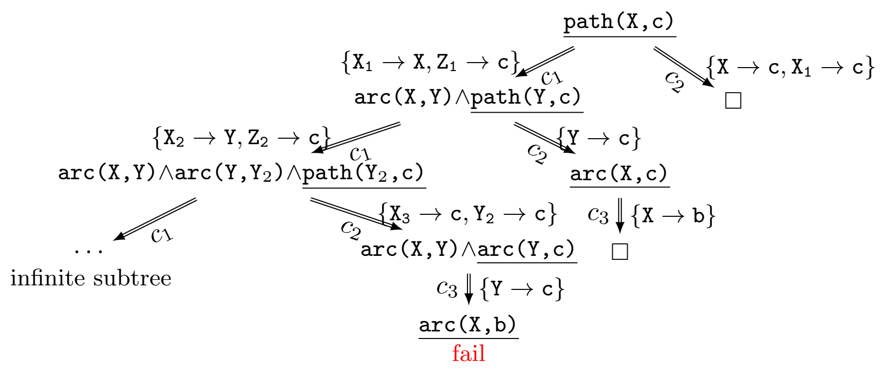
```
path(X,Z) ← arc(X,Y)∧path(Y,Z).   % c1
path(X,X) ← true.                  % c2
arc(b,c) ← true.                   % c3
```

and the query Q = `path(X,c)`. A possible interpretation of the relations `arc` and `path` defined by this program is as follows: `arc(X,Y)` holds if there is an arc from `X` to `Y`, and `path(X,Y)` holds if there is a path from `X` to `Y`. Two SLD-trees for $P \cup \{Q\}$ are illustrated below:

SLD-tree via leftmost atom selection rule



SLD-tree via rightmost atom selection rule



where $\Box$ represents the empty clause (`true`). The first tree is finite while the second one is infinite. Both trees are successful and contain the same computed answer substitutions: $\{\mathtt{X} \to \mathtt{b}\}$ and $\{\mathtt{X} \to \mathtt{c}\}$.

3.1.1 Successful SLD-trees

> If an SLD-tree for $P \cup \{Q\}$ is successful, then all SLD-trees of $P \cup \{Q\}$ are successful.

Moreover, if θ is an answer substitution obtained with an SLD-derivation ξ_1 of length n for a selection rule $\mathcal{R}_1$, then the same θ can be obtained with any other selection rule $\mathcal{R}_2$, by constructing another SLD-derivation ξ_2 of length n for $\mathcal{R}_2$.

For example, the SLD-derivation via leftmost atom selection

$$\begin{aligned}\xi_{\text{left}} := \underline{\mathtt{path(X,c)}} &\Rightarrow_{\{\mathtt{X_1}\to\mathtt{X},\mathtt{Z_1}\to\mathtt{c}\},c_1} \underline{\mathtt{arc(X,Y)}}\wedge\mathtt{path(Y,c)}\\ &\Rightarrow_{\{\mathtt{X}\to\mathtt{b},\mathtt{Y}\to\mathtt{c}\},c_3} \underline{\mathtt{path(c,c)}} \Rightarrow_{\{\mathtt{X_2}\to\mathtt{c}\},c_2} \Box\end{aligned}$$

has length 3 and computed answer $\{\mathtt{X} \to \mathtt{c}\}$. The SLD-derivation via rightmost atom selection corresponding to ξ_{left} is

$$\begin{aligned}\xi_{\text{right}} := \underline{\mathtt{path(X,c)}} &\Rightarrow_{\{\mathtt{X_1}\to\mathtt{X},\mathtt{Z_1}\to\mathtt{c}\},c_1} \mathtt{arc(X,Y)}\wedge\underline{\mathtt{path(Y,c)}}\\ &\Rightarrow_{\{\mathtt{Y}\to\mathtt{c}\},c_2} \underline{\mathtt{arc(X,c)}} \Rightarrow_{\{\mathtt{X}\to\mathtt{b}\},c_3} \Box\end{aligned}$$

3.1.2 Finitely failed SLD-trees

If an SLD-tree is not successful, it is either finitely failed or infinite. Unfortunately, there are queries Q such that $P \cup \{Q\}$ has both a finitely failed SLD-tree and an infinite tree which is unsuccessful. For example, if P is the previous program and Q is the query `path(a,b)`, then the SLD-tree of $P \cup \{Q\}$ via leftmost selection rule is finitely failed, and the SLD-tree of $P \cup \{Q\}$ via rightmost selection rule is infinite and unsuccessful.

3.2 Concluding remarks

Suppose Q is a query with variables $X_1, \ldots, X_n$. We write $P \vdash \exists X_1. \cdots .\exists.X_n.Q$ if the formula

$$\exists X_1. \cdots .\exists.X_n.Q$$

follows from a program P, and wish to decide if the relation $P \vdash \exists X_1. \cdots .\exists.X_n.Q$ holds or not. Moreover, if $P \vdash \exists X_1. \cdots .\exists.X_n.Q$ holds, we want to find the substitutions θ such that $P \vdash Q\theta$ holds.

To solve this problem, we compute an SLD-tree of $P \cup \{Q\}$. If the tree is finite, then the computation terminates and:

1. If the tree is finitely failed, then $P \not\vdash \exists X_1. \cdots .\exists.X_n.Q$, and therefore there are no substitutions θ such that $P \vdash Q\theta$.

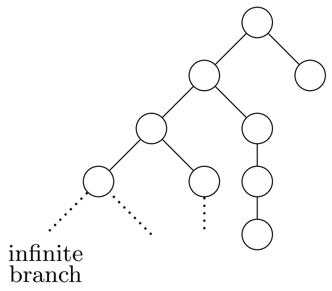
2. If the tree is successful, then $P \vdash \exists X_1. \cdots .\exists.X_n.Q$. Moreover, we can extract from the tree the computed answers θ such that $P \vdash Q\theta$.

If the tree is infinite, the computation of the whole tree does not terminate. If the tree is computed incrementally, by breadth-first traversal, then

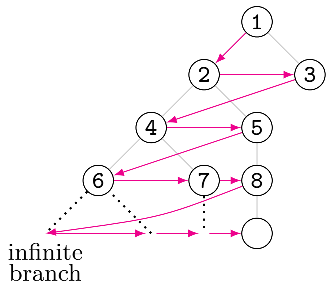
3. We decide $P \vdash \exists X_1. \cdots .\exists.X_n.Q$ as soon as we produce a node for $\Box$. In this case, we can also report $P \vdash Q\theta$ for the computed answer θ by the SLD-derivation to that node.

Note that, if the tree is infinite, we can decide $P \vdash \exists X_1. \cdots .\exists.X_n.Q$ only when a node for $\Box$ is produced. If such a node is never produced, the computation of the tree runs forever and we can not decide anything.

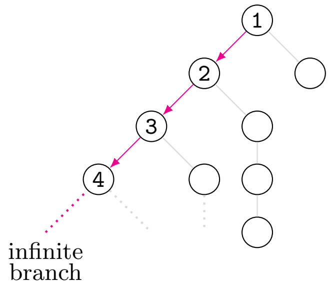
Breadth-first is a **fair** traversal strategy because it traverses all nodes of a tree, even if it the tree is infinite. By contrast, depth-first is an **unfair** traversal strategy because it does not traverse all nodes of some infinite trees. For example, the depth-first traversal of the SLD-tree via rightmost atom selection from page 13 has the shape



Breadth-first will generate all nodes of the tree in the order depicted below:



and depth-first will generate only the leftmost branch of the tree:



Implementation of SLD-derivations in Prolog

The incremental computation of a SLD-tree by breadth-first traversal is fair but memory-consuming. For this reason, most languages for logic programming, including Prolog, construct an SLD-tree incrementally, by depth-first traversal, which consumes much less memory. Since this traversal strategy is unfair, it may never produce successful derivations (and computed answers) which exist. In contrast, breadth-first traversal guarantees that all successful derivations (and computed answers) are eventually produced.

4 Programming in Prolog

Prolog is the most popular language used for logic programming. It is designed to answer queries using a predefined and predictable search strategy called SLDNF-resolution.

4.1 SLDNF-resolution

SLDNF [AB94] is an extension of SLD-resolution to deal with negation as failure. This means that Prolog can work with

- general clauses, which are formulas $H \leftarrow \bigwedge_{i=1}^{p} L_i$ where H is atom and L_i are literals.
- general queries, which are formulas $\bigwedge_{i=1}^{p} L_i$ where L_i are literals.

The simplified syntax of Prolog can be used to write such clauses and queries too; the only extension is that we write a literal $\neg p(t_1, \ldots, p_n)$ as `not(`$p(t_1, \ldots, t_n)$`)`.

For example, we can write the Prolog program

```
bird(olaf).                               % c1
penguin(olaf).                            % c2
fly(X) :- bird(X), not(abnormal(X)).      % c3
abnormal(X) :- penguin(X).                % c4
```

to encode the set of clauses {`bird(tweety)`, `fly(X)` $\leftarrow$ `bird(X)` $\wedge \neg$`abnormal(X)`, `abnormal(X)` $\leftarrow$ `penguin(X)`} which represent the following knowledge:

- Tweety is a bird,
- Every `X` can fly if it is a bird and it is not abnormal.
- Every `X` is abnormal if it is a penguin.

If P is a program, $Q = \bigwedge_{L \in \mathcal{L}} L$ is a non-empty query, and $H \leftarrow B$ a variant of a clause $c \in P$ such that $Var(H \leftarrow B) \cap Var(Q) = \emptyset$, then

1. If θ is an mgu of H and an atom $A_0 \in \mathcal{L}$, then the query

   $$Q' = \left(B \wedge \bigwedge_{L \in \mathcal{L} - \{A_0\}} L \right) \theta$$

   is an **SLDNF-resolvent** of Q and c w.r.t. A_0, with mgu θ, and we write $Q \Rightarrow_{\theta,c} Q'$ to express this fact.

2. If $\neg A_0 \in \mathcal{L}$ is a ground literal and there is a finitely failed SLDNF-tree of $P \cup \{A_0\}$, then $Q' = \bigwedge_{L \in \mathcal{L} - \{\neg A_0\}} L$ is a an **SLDNF-resolvent** of Q, and we write $Q \Rightarrow_{\epsilon} Q'$ to express this fact.

This means that, when such a variable-free literal $\neg A_0$ is selected, a subproof (or subcomputation) is attempted to determine whether there is an SLDNF-refutation (=finitely failed tree) of $P \cup \{A_0\}$ starting from A_0. The selected subgoal $\neg A_0$ succeeds if the subproof fails, and it fails if the subproof succeeds.

Example 1. $\neg$`fly(olaf)` succeeds because $P \cup \{$`fly(olaf)`$\}$ has the finitely failed SLDNF-tree

$$\begin{tikzcd}
\underline{\texttt{fly(olaf)}} \arrow[d, Rightarrow, "c_3"', "\{\texttt{X}_1 \to \texttt{olaf}\}"] \\
\underline{\texttt{bird(olaf)}} \wedge \neg\texttt{abnormal(olaf)} \arrow[d, Rightarrow, "c_1"', "\epsilon"] \\
\underline{\neg\texttt{abnormal(olaf)}} \\[-20pt]
\text{fail}
\end{tikzcd}$$

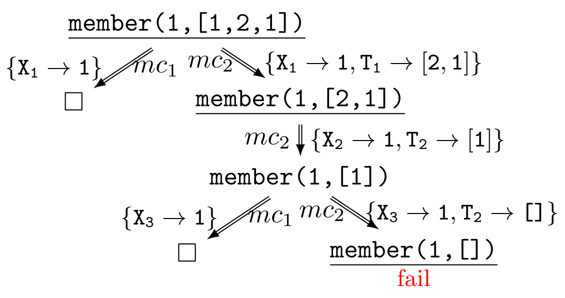
The subgoal $\neg$`abnormal(olaf)` fails because the subgoal `abnormal(olaf)` succeeds:

$$\begin{tikzcd}
\underline{\texttt{abnormal(olaf)}} \arrow[d, Rightarrow, "c_4"', "\epsilon"] \\
\underline{\texttt{penguin(olaf)}} \arrow[d, Rightarrow, "c_2"', "\epsilon"] \\
\square
\end{tikzcd}$$

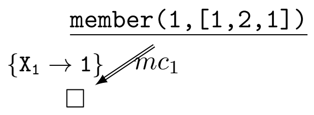
Example 2. List membership is a built-in Prolog relation, defined by the program P_{member}

```
% base case: X is member of any list that starts with X
member(X,[X|_]).                                % mc1
% recursive case: X is member of any list whose tail contains X
member(X,[_|T]) :- member(X,T).                 % mc2
```

The query $Q =$ `member`$(1, [1, 2, 1])$ is successful because $P_{member} \cup \{Q\}$ has the following successful tree starting from Q:

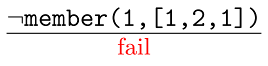


In fact, Prolog returns `true` immediately after the generation of the partial SLDNF-tree



```
?-member(1,[1,2,1]).
true.
```

and will generate the rest of the tree only if the user decides to do so by pressing `;`. Because Prolog can find a successful tree of Q, the query $\neg$`member`$(1, [1, 2, 1])$ fails, and Prolog returns `false`:



```
?- not(member(1,[1,2,1])).
false.
```

4.2 Concluding remarks

Closed-world assumption and Open-world assumption

Prolog programming are based on the **closed-world assumption** (CWA), which presumes that a statement is true if and only if it can be deduced from the knowledge (=program) we have. The opposite of the closed-world assumption is the **open-world assumption** (OWA), stating that lack of knowledge does not imply falsity. For example, if our knowledge is the logic program $P = \{$`bird(olaf)`$\}$ then `bird(theety)` is

- false, in reasoning systems based on the closed world assumption (including Prolog), because it can not be deduced from P.
- unknown, in reasoning systems based on the open world assumption.

Reasoning systems based on the closed-world assumption are **non-monotonic**: this means that, if we extend our knowledge P with new knowledge to P', then some stetements which were previously true may become false. For example, $\neg$`bird(tweety)` is true with respect to the program $P = \{$`bird(olaf)`$\}$ and false with respect to the extended program $P' = \{$`bird(olaf)`$,$ `bird(tweety)`$\}$.

According to Kowalski [Kow14], programming with positive clauses and queries using SLD-resolution is sufficient for database applications, but not adequate for Artificial Intelligence, most importantly because it fails to capture non-monotonic reasoning. SLDNF-resolution with general clauses and queries is adequate for AI.

Sources of nondeterminism

The search for solutions of a query Q with respect to a program P is based on the construction of a SLDNF-tree for $P \cup \{Q\}$. We noticed that there are four sources of nondeterminism in the computation of a successful SLDNF-derivation

$$Q \Rightarrow_{\theta_1,c_1} Q_2 \ \cdots \ \Rightarrow_{\theta_{n-1},c_{n-1}} Q_{n-1} \Rightarrow_{\theta_n,c_n} \square$$

that produces a computed answer $\theta = \theta_1 \ldots \theta_n|_{Var(Q)}$:

(A) choice of the selected atom in the considered query,
(B) choice of the program clause applicable to the selected atom,
(C) choice of the renaming of the program clause used,
(D) choice of the mgu.

(C) and (D) are **don't care** sources of nondeterminism: we are free to choose **any** mgu and **any** standardized apart variant of a program clause, because these choices don't affect the computation of a complete set of answers for a query.

(B) is a **don't know** source of nondeterminism. This means that, if we want to find all answers of a query, we must consider all program clauses applicable to the selected atom. That is, we must generate all nodes of the SLDNF-tree. We can do so with a fair tree traversal strategy, like breadth-first, but Prolog is based on depth-first traversal, where nodes are generated in the top-down order of the applicable rules in the program. Therefore, the computation of some answers may depend on the order how program clauses are written in a program. A simple heuristic is:

- Write the program clauses for base cases before the recursive program clauses.

(A) is a **don't care** source of determinism for positive programs and queries, but is relevant in Prolog, which deals with negation as failure, because some selection rules guarantee the generation of a finitely failed SLDNF-tree and others do not. To illustrate, consider the query $Q = \neg\texttt{path}(\texttt{d},\texttt{c})$ with respect to the logic program P below:

```
path(X,Z) ← arc(X,Y)∧path(Y,Z).  % c1
path(X,X) ← true.                % c2
arc(b,c) ← true.                 % c3
```

$\{Q\} \cup P$ has a successful SLDNF-tree via the leftmost selection rule because $\texttt{path}(\texttt{d},\texttt{c})$ has a finitely failed SLDNF-tree via the leftmost selection rule, namely

$$\begin{tikzcd}
\underline{\texttt{path(d,c)}} \arrow[d, Rightarrow, "c_1"', "\{\texttt{X}_1 \to \texttt{d}, \texttt{Z}_2 \to \texttt{c}\}"] \\
\underline{\texttt{arc}(\texttt{d},\texttt{Y}_1)} \wedge \texttt{path}(\texttt{Y}_1,\texttt{c}) \\
\text{fail}
\end{tikzcd}$$

but $P \cup \{Q\}$ has an infinite unsuccessful SLDNF-tree via the rightmost selection rule because $\texttt{path}(\texttt{d},\texttt{c})$ has an infinite unsuccessful SLDNF-tree via the rightmost selection rule, namely

$$\begin{tikzcd}
 & & \underline{\texttt{path(d,c)}} \arrow[d, Rightarrow, "c_1"', "\{\texttt{X}_1 \to \texttt{d}, \texttt{Z}_2 \to \texttt{c}\}"] & \\
 & & \texttt{arc}(\texttt{d},\texttt{Y}_1) \wedge \underline{\texttt{path}(\texttt{Y}_1,\texttt{c})} \arrow[dl, Rightarrow, "c_1"', "\{\texttt{X}_2 \to \texttt{Y}_1, \texttt{Z}_2 \to \texttt{c}\}"] \arrow[dr, Rightarrow, "c_2", "\{\texttt{Y}_1 \to \texttt{c}\}"'] & \\
 & \texttt{arc}(\texttt{d},\texttt{Y}_1) \wedge \texttt{arc}(\texttt{Y}_1,\texttt{Y}_2) \wedge \underline{\texttt{path}(\texttt{Y}_2,\texttt{c})} \arrow[dl, Rightarrow, "c_1"] \arrow[dr, Rightarrow, "c_2", "\{\texttt{Y}_2 \to \texttt{c}\}"'] & & \underset{\text{fail}}{\underline{\texttt{arc}(\texttt{d},\texttt{c})}} \\
 \underset{\text{infinite branch}}{\ldots} & & \texttt{arc}(\texttt{d},\texttt{Y}_1) \wedge \underline{\texttt{arc}(\texttt{Y}_1,\texttt{c})} \arrow[d, Rightarrow, "c_3"', "\{\texttt{Y}_1 \to \texttt{b}\}"] & \\
 & & \underset{\text{fail}}{\underline{\texttt{arc(d,b)}}} &
\end{tikzcd}$$

For (A), Prolog implements the leftmost selection rule. A simple heuristics to avoid the generation of infinite unsuccessful SLD-trees is:

- Avoid left-recursion, by not placing the recursive calls to be the first ones in the body of clauses.

To appreciate this heuristics, see what happens if we change program P to be

```
path(X,Z) ← path(X,Y)∧arc(Y,Z).  % c1 is left-recursive
path(X,X) ← true.                % c2
arc(b,c) ← true.                 % c3
```

and we use the leftmost selection rule of Prolog for the query `path(d,c)`.

References

[AB94] K.R. Apt and R.N. Bol. Logic programming and negation: A survey. *Journal of Logic Programming*, 19,20:9–71, 1994. Available here.

[Kow74] R.A. Kowalski. Predicate logic as a programming language. *Information Processing*, pages 569–574, 1974. Available here.

[Kow14] R. Kowalski. Computational logic. In D. Gabbay and J. Woods, editors, *Computational Logic*, History of Logic, pages 523–569. Elsevier, 2014. Available here.

[MM82] A. Martelli and U. Montanari. An efficient unification algorithm. *ACM Transactions on Programming Languages and Systems*, 4:258–282, 1982. Available here.

[Rob65] J.A. Robinson. A machine-oriented logic based on the resolution principle. *Journal of the ACM*, 12(1):23–41, 1965. Available here.