5 Operative semantics of logic programs

The operative semantics of Prolog is given by the SLD-resolution calculus and a depth-first search strategy for finding SLD-derivations. The letters SLD stand for Selected, Linear and Definite and will be explained later. General resolution and its special form, SLD-resolution, were introduced by A Robinson.

5.1 SLD-resolution

1. Let $Q$ be the query

$$A_1, A_2, \ldots, A_n$$

(we do not write the ‘?-’) and $C$ a clause of the form

$$B ::= B_1, \ldots, B_m$$

such that $Q$ and $C$ have no variables in common. Assume that the first goal, $A_1$, and the head of the clause, $B$, are unifiable. Let $\theta$ be an mgu of $A_1$ and $B$, and let $Q'$ be the query

$$(B_1, \ldots, B_m, A_2, \ldots, A_n)\theta, \text{ (that is, } B_1\theta, \ldots, B_m\theta, A_2\theta, \ldots, A_n\theta)$$

Then $Q'$ is called a resolvent of $Q$ with input clause $C$ and mgu $\theta$. We call the passage from $Q$ to $Q'$ an SLD-resolution step and write it

$$Q \xrightarrow{C, \theta} Q'$$

When the clause $C$ is irrelevant we drop a reference to it.

2. An SLD-derivation w.r.t. a program $P$ is a finite or infinite sequence

$$Q \xrightarrow{C_1, \theta_1} Q_1 \xrightarrow{C_2, \theta_2} \ldots$$

where the input clauses $C_1$, $C_2$, are variants of program clauses in $P$.

An SLD-derivation is successful if it is finite and ends with the empty query, $\square$; it is failed if it is finite and the last query in the derivation is non-empty, but no further resolution step is possible.

4. Suppose we are given a program $P$ and a successful SLD-derivation
\[ Q \xrightarrow{C_1,\theta_1} Q_1 \xrightarrow{C_2,\theta_2} \ldots \xrightarrow{C_n,\theta_n} \square \]

Let \( \theta \) be the composition \( \theta_1\theta_2,\ldots,\theta_n \) restricted to the variables in \( Q \), that is, we retain from \( \theta_1,\theta_2,\ldots,\theta_n \) only those bindings \( X/t \) where \( X \) occurs in \( Q \). Then \( \theta \) is called a \textbf{computed answer substitution (c.a.s.)} for short of the query \( Q \) w.r.t. the program \( P \). Hence, if \( X_1,\ldots,X_k \) are the variables in \( Q \), the computed answer substitution is

\[ \theta = \{X_1/t_1, \ldots, X_k/t_k\} \]

where \( t_i \equiv X_i\theta_1\theta_2,\ldots,\theta_n \).

Prolog usually displays a computed answer substitution \( \theta = \{X_1/t_1, \ldots, X_n/t_n\} \) in the form

\[
\begin{align*}
X_1 & = t_1 \\
\cdots & \\
X_n & = t_n
\end{align*}
\]

**Example.** Consider again

\[
\% \text{ Program SUMMER}
\]

\[
\begin{array}{l}
\text{summer.} \quad \%1 \\
\text{warm : - summer.} \quad \%2 \\
\text{warm : - sunny.} \quad \%3 \\
\text{happy : - summer, warm.} \quad \%4
\end{array}
\]

Since this program doesn’t contain variables no substitutions will be involved in SLD-derivations.

We have the successful SLD-derivation

\[
\begin{array}{l}
\text{happy} \quad \xrightarrow{1} \text{summer, warm} \quad \xrightarrow{1} \text{warm} \quad \xrightarrow{2} \text{summer} \quad \xrightarrow{1} \square
\end{array}
\]

Hence \text{SUMMER} \vdash \text{happy}.

We also have the failed SLD-derivation
happy $\overset{4}{\Rightarrow}$ summer, warm $\overset{1}{\Rightarrow}$ warm $\overset{3}{\Rightarrow}$ sunny

We have `SUMMER` $\not\Rightarrow` sunny`, since obviously the only SLD-derivation starting with `sunny` is the trivial failed derivation of length one.

**Example.** Recall the program `SUM` from chapter 2.

% Program SUM

```
sum(X, 0, X). %1 X+0=X
sum(X, s(Y), s(Z)) :- sum(X, Y, Z). %2 X+(Y+1)=(X+Y)+1
```

For example, for the query

?- sum(s(s(0)), s(s(0)), Z).

we have the successful SLD-derivation

\[
\begin{align*}
\text{sum}(s(s(0)), s(s(0)), Z) & \overset{2}{\Rightarrow} \text{sum}(s(s(0)), s(0), Z_1) \\
& \overset{2}{\Rightarrow} \text{sum}(s(s(0)), 0, Z_2) \\
& \overset{1}{\Rightarrow} \square
\end{align*}
\]

where

\[
\begin{align*}
\theta_1 &= \{ X/s(s(0)), Y/s(0), Z/s(Z_1) \} \\
\theta_2 &= \{ X_2/s(s(0)), Y_2/s(0), Z_1/s(Z_2) \} \\
\theta_3 &= \{ X_3/s(s(0)), Z_2/s(s(0)) \}
\end{align*}
\]

In order to find the computed answer substitution we have to compute

\[
Z\theta_1\theta_2\theta_3 \equiv s(Z_1)\theta_2\theta_3 \equiv s(s(Z_2))\theta_3 \equiv s(s(s(s(0))))
\]

Therefore the computed answer substitution is

\[
\theta = \{ Z/s(s(s(s(0)))) \}
\]

and the corresponding computed instance is

\[
\text{sum}(s(s(0)), s(s(0)), s(s(s(s(0)))))
\]
Remarks. In a resolution step \( Q \overset{C}{\rightarrow} Q' \) the first goal of the query \( Q \) plays a distinguished role, since it is the one which is replaced by other atomic formulas. Therefore this first goal is called the selected goal. This explains the ‘S’ in SLD-resolution. We could equally well have decided to always select the last atomic formula. In fact, the fundamental properties of SLD-resolution are not affected by the particular choice of the selection rule. Prolog also always selects the first goal.

The letter ‘D’ in the acronym ‘SLD’ refers to the fact that the clauses of a Prolog program are also called definite clauses. More general forms of clauses are important in automated theorem proving, but will not be discussed here.

The letter ‘L’ stands for linear and refers to the linear structure of SLD-derivation. In general resolution a derivation is a tree.

5.2 Soundness and completeness of SLD-resolution

Theorem. For every program \( P \), query \( Q \) and substitution \( \tau \),

\[
\tau \text{ is a specialization of a computed answer substitution of } Q \text{ w.r.t. } P
\]

if and only if

\[
Q\tau \text{ holds in all models of } P
\]

For a proof see Apt’s book.

Corollary. If \( \tau \) is a specialization of a computed answer substitution of \( Q \) w.r.t. \( P \), then \( Q\tau \) holds in the least Herbrand-model, \( \mathcal{M}(P) \).

Example. For the program SUM we consider the following query:

\[
?- \text{sum}(X, s(0), Y).
\]

Prolog will return the answer

\[
X = U
\]

\[
Y = s(U)
\]
(instead of \( U \) any other variable different from \( X, Y \) could be used) that is, the answer substitution
\[
\theta := \{ X/U, Y/s(U) \}
\]

Hence \( \text{SUM} \vdash \theta \sum(X, s(0), Y) \). According to the theorem above (with \( \tau := \theta \)) it follows that \( \forall U \sum(U, s(0), s(U)) \) holds in all models of \( \text{SUM} \), and, by the corollary, this formula holds in particular in \( \mathcal{M}(P) \). Hence \( \sum(U, s(0), s(U)) \) is true in \( \mathcal{M}(P) \) for all possible values of \( U \), that is \( \sum(s^n(0), s(0), s^{n+1}(0) \) holds in \( \mathcal{M}(P) \) for all \( n \in \mathbb{N} \).

The converse of the corollary, however, does not hold, as the example at the end of chapter 3.3 showed.

Combining the theorem above and the theorem in chapter 3.3 we obtain:

**Theorem.** For any program \( P \), query \( Q \) and ground substitution \( \tau \) the following statements are equivalent:

(i) \( Q\tau \) holds in all models of \( P \).

(ii) \( Q\tau \) holds in the least Herbrand-model, \( \mathcal{M}(P) \).

(iii) \( \tau \) is a specialization of a computed answer substitution of \( Q \) w.r.t. \( P \).

Roughly speaking the implication \((iii) \Rightarrow (ii)\) in the theorem above says that every successful SLD-derivation for a query describes an inference tree for an instance of the query (see chapter 3.1 and the example there on the program FLIGHTS).

Since in many cases the least Herbrand model is the intended model of a program \( P \) one might be interested in an effective procedure that for a query \( Q \) generates all substitutions \( \tau \) such that \( Q\tau \) holds in \( \mathcal{M}(P) \) (we do have an effective procedure producing all ground substitutions \( \tau \) with that property, namely SLD-resolution). However, fundamental results in mathematical logic (Gödel’s incompleteness theorem) tell us that this is impossible:

**Theorem.** There is no effective procedure that for any program \( P \) and query \( Q \) would generate precisely all substitutions \( \tau \) such that \( Q\tau \) holds in \( \mathcal{M}(P) \).
5.3 Prolog’s search strategy

When we ask a query $Q$ for a given program $P$, Prolog searches the tree of all SLD-derivations starting with $Q$ (w.r.t. $P$) following the so-called depth-first search strategy. The advantage of depth-first search is its efficiency, its disadvantage is its incompleteness: it may happen that Prolog does not find an existing successful SLD-derivation.

**Definition.** Given a program $P$, the **SLD-tree** of a query $Q$ is the tree of all (successful, failed, or infinite) SLD-derivations, starting with $Q$ and using input clauses from $Q$.

**Exercise.** Draw the SLD-tree of the query happy w.r.t. the program SUMMER.

---

**Exercise.** Draw the SLD-tree of the query loop w.r.t. the following program:

```prolog
loop. %1
loop :- loop. %2
```

```
Depth-first search

We assume that we are given a program \( P \) and a query \( Q \). Prolog searches for a successful SLD-derivation for \( Q \) as follows:

- The search starts at the top node (= root), of the SLD-tree for \( Q \), that is the query \( Q \).

- When the node of the SLD-tree currently visited is not a leaf, that is, an SLD-step is applicable, then go to the left-most child, that is, choose the first clause in the program that can be used as an input clause and proceed to the corresponding resolvent.

- When the node of the SLD-tree currently visited is an unsuccessful leaf, that is, the query is nonempty, but no SLD-step is applicable, then go back (= toward the root) until a node is found where the child chosen was not the rightmost one, that is there is a clause in the program below the chosen clause, that can be used for an SLD-step. Chose that clause and perform an SLD-step. This process is called \textbf{backtracking}.

- When the node of the SLD-tree currently visited is an unsuccessful leaf, and no backtracking is possible, then the whole SLD-tree has been searched unsuccessfully. Hence stop and return the answer “\textbf{No}”.

- When the node of the SLD-tree currently visited is a successful leaf, then stop and return the answer “\textbf{Yes}”, or the computed answer substitution.

Examples.

1. Let us consider again

\begin{verbatim}
% Program SUMMER

summer. %1
warm :- summer. %2
warm :- sunny. %3
happy :- summer, warm. %4
\end{verbatim}

Draw the SLD-tree for the query \texttt{happy} and indicate, which, and in which order, nodes are visited by Prolog.
2. Now let us do the same with a modified program where the second and the third rule are swapped.

% Program SUMMER1

summer. %1
warm :- sunny. %2
warm :- summer. %3
happy :- summer, warm. %4

3. Consider again

%Program LOOP

loop. %1
loop :- loop. %2

and indicate Prolog’s search when asked the query loop.
4. Now we swap the clauses

Program LOOP1

loop :- loop. %1
loop. %2

and ask again the query loop.

5. Finally we ask the queries

fail, loop

and also

loop, fail

to the programs LOOP and LOOP1. Here we used the atomic formula fail, which by definition always fails.
Example 4 exhibits the incompleteness of depth-first search: although a successful SLD-derivation exists, Prolog is unable to find it. Therefore the Prolog programmer has to take care of the ‘right’ order of the clauses in his program.

**Breadth-first search**

Another search strategy, which is *not* used by Prolog, would be **breadth-first search**, in which the nodes of an SLD-tree are visited in order of their length, or distance to the root: The search starts with the root, then the children of the root, i.e. the nodes of length one, are visited (from left to right), then all grandchildren of the root, i.e. the nodes of length two, and so on, until a successful node is found.

It is easy to see that breadth-first search is complete, that is, if a successful node exists, it is found.

Breadth-first search is not used by Prolog, since in general it is less efficient: if the shortest successful SLD-derivation has length \( n \) and on average each node has two children, then approximately \( 2^n \) nodes have to be visited until the success is found.

**Exercise.** Indicate breadth-first search for the SLD-trees of the examples 1-5 above.