Formalising SLD Resolution with Variables

Let $G$ be the goal $\bot : - A_1, \ldots, A_m$ and let $C$ be the clause $H : - B_1, \ldots, B_n$. We say that a goal $G'$ is derived from $G$ and $C$ using most general unifier $\theta$ (and leftmost selection rule) if

1. $\theta$ is a most general unifier of $A_1$ and $H$,
2. $G'$ is $\bot : - (B_1, \ldots, B_n, A_2, \ldots, A_m)\theta$

Let $P$ be a logic program and $G$ a goal. An SLD derivation of $P \cup \{G\}$ consists of

1. a sequence $G_0 = G, G_1, \ldots, G_n$ of goals,
2. a sequence $C_1, \ldots, C_n$ of clauses in $P$,
3. a sequence $\theta_1, \ldots, \theta_n$ of substitutions

such that for each $i = 0 \ldots n - 1$, the goal $G_{i+1}$ is derived from $G_i$ and $C_{i+1}$ using most general unifier $\theta_{i+1}$.

If $G_n = \bot$ then the derivation is called an SLD-refutation of $P \cup \{G\}$. The computed answer substitution of an SLD refutation of $P \cup \{G\}$ is the substitution

$$\theta_1 \circ \theta_2 \circ \ldots \circ \theta_n$$

restricted to the set of variables in the goal.
Example:

```
append(emptylist, X, X).
append(1(X, Y), Z, 1(X, T)) :- append(Y, Z, T).
```

Prolog Operational Semantics

Prolog searches the tree of all SLD-derivations for the goal

1. using top to bottom selection of clauses to resolve against
2. using leftmost selection of literals to resolve against in each goal generated (this is how we just defined the goal generated by resolution).

and returns any answer substitutions it finds.

**However,** when computing the most general unifier, Prolog omits the occurs check, because it is seen as causing inefficiency.

These choices were made with the intention of producing an efficient programming language, but they have some detrimental consequences from the point of view of logical correctness....
1. Because the depth first search can get stuck in an infinite branch, Prolog may fail to return answers that are logical consequences of the program.

2. Because the occurs check is omitted, Prolog may return answers that are not logical consequences of the program.

Example:

\[
\text{someone_is_their_own_parent} : \neg \text{parent}(X, X).
\]

\[
\text{parent(father}(Y), Y).
\]

For the query “\text{someone_is_their_own_parent}?”, Prolog returns “yes.”

The query “\text{parent}(X, X)?”, returns an infinite size value: \[X \mapsto \text{father(father(father(\ldots))))]}\]