Chapter 3

SLD-Resolution

This chapter introduces the inference mechanism which is the basis of most logic programming systems. The idea is a special case of the inference rule called the *resolution principle* — an idea that was first introduced by J. A. Robinson in the mid-sixties for a richer language than definite programs. As a consequence, only a specialization of this rule, that applies to definite programs, is presented here. For reasons to be explained later, it will be called the *SLD-resolution principle*.

In the previous chapter the model-theoretic semantics of definite programs was discussed. The SLD-resolution principle makes it possible to draw correct conclusions from the program, thus providing a foundation for a logically sound *operational semantics* of definite programs. This chapter first defines the notion of SLD-resolution and then shows its correctness with respect to the model-theoretic semantics. Finally SLD-resolution is shown to be an instance of a more general notion involving the construction of proof trees.

3.1 Informal Introduction

Every inference rule of a logical system formalizes some natural way of reasoning. The presentation of the SLD-resolution principle is therefore preceded by an informal discussion about the underlying reasoning techniques.

The sentences of logic programs have a general structure of logical implication:

$$A_0 \leftarrow A_1, \ldots, A_n \qquad (n \geq 0)$$

where $A_0, \ldots, A_n$ are atomic formulas and where A_0 may be absent (in which case it is a goal clause). Consider the following definite program that describes a world where "parents of newborn children are proud", "Adam is the father of Mary" and "Mary is newborn":

$$\begin{array}{l} proud(X) \leftarrow parent(X, Y), newborn(Y). \\ parent(X, Y) \leftarrow father(X, Y). \\ parent(X, Y) \leftarrow mother(X, Y). \\ father(adam, mary). \\ newborn(mary). \end{array}$$

Notice that this program describes only "positive knowledge" — it does not state who is *not* proud. Nor does it convey what it means for someone not to be a parent. The problem of expressing negative knowledge will be investigated in detail in Chapter 4 when extending definite programs with negation.

Say now that we want to ask the question "Who is proud?". The question concerns the world described by the program P, that is, the intended model of P. We would of course like to see the answer "Adam" to this question. However, as discussed in the previous chapters predicate logic does not provide the means for expressing this type of *interrogative* sentences; only *declarative* ones. Therefore the question may be formalized as the goal clause:

$$\leftarrow proud(Z) \qquad (G_0)$$

which is an abbreviation for $\forall Z\, \neg proud(Z)$ which in turn is equivalent to:

$$\neg \exists Z\, proud(Z)$$

whose reading is "Nobody is proud". That is, a negative answer to the query above. The aim now is to show that this answer is a false statement in every model of P (and in particular in the intended model). Then by Proposition 1.13 it can be concluded that $P \models \exists Z\, proud(Z)$. Alas this would result only in a "yes"-answer to the original question, while the expected answer is "Adam". Thus, the objective is rather to find a substitution θ such that the set $P \cup \{\neg\, proud(Z)\theta\}$ is unsatisfiable, or equivalently such that $P \models proud(Z)\theta$.

The starting point of reasoning is the assumption G_0 — "For any Z, Z is not proud". Inspection of the program reveals a rule describing one condition for someone to be proud:

$$proud(X) \leftarrow parent(X, Y), newborn(Y). \qquad (C_0)$$

Its equivalent logical reading is:

$$\forall(\neg proud(X) \supset \neg(parent(X, Y) \wedge newborn(Y)))$$

Renaming X into Z, elimination of universal quantification and the use of *modus ponens* with respect to G_0 yields:

$$\neg\,(parent(Z, Y) \wedge newborn(Y))$$

or equivalently:

$$\leftarrow parent(Z, Y), newborn(Y). \qquad (G_1)$$

Thus, one step of reasoning amounts to replacing a goal G_0 by another goal G_1 which is true in any model of $P \cup \{G_0\}$. It now remains to be shown that $P \cup \{G_1\}$ is unsatisfiable. Note that G_1 is equivalent to:

$$\forall Z \, \forall Y \; (\neg \mathit{parent}(Z, Y) \vee \neg \mathit{newborn}(Y))$$

Thus G_1 can be shown to be unsatisfiable with P if in every model of P there is some individual who is a parent of a newborn child. Thus, check first whether there are any parents at all. The program contains a clause:

$$\mathit{parent}(X, Y) \leftarrow \mathit{father}(X, Y). \tag{C_1}$$

which is equivalent to:

$$\forall (\neg \mathit{parent}(X, Y) \supset \neg \mathit{father}(X, Y))$$

Thus, G_1 reduces to:

$$\leftarrow \mathit{father}(Z, Y), \mathit{newborn}(Y). \tag{G_2}$$

The new goal G_2 can be shown to be unsatisfiable with P if in every model of P there is some individual who is a father of a newborn child. The program states that "Adam is the father of Mary":

$$\mathit{father}(\mathit{adam}, \mathit{mary}). \tag{C_2}$$

Thus it remains to be shown that "Mary is not newborn" is unsatisfiable together with P:

$$\leftarrow \mathit{newborn}(\mathit{mary}). \tag{G_3}$$

But the program also contains a fact:

$$\mathit{newborn}(\mathit{mary}). \tag{C_3}$$

equivalent to $\neg \mathit{newborn}(\mathit{mary}) \supset \Box$ leading to a refutation:

$$\Box \tag{G_4}$$

The way of reasoning used in this example is as follows: to show existence of something, assume the contrary and use *modus ponens* and elimination of the universal quantifier to find a counter-example for the assumption. This is a general idea to be used in computations of logic programs. As illustrated above, a single computation (reasoning) step transforms a set of atomic formulas — that is, a *definite goal* — into a new set of atoms. (See Figure 3.1.) It uses a selected atomic formula $p(s_1, \ldots, s_n)$ of the goal and a selected program clause of the form $p(t_1, \ldots, t_n) \leftarrow A_1, \ldots, A_m$ (where $m \geq 0$ and $A_1, \ldots, A_m$ are atoms) to find a common instance of $p(s_1, \ldots, s_n)$ and $p(t_1, \ldots, t_n)$. In other words a substitution θ is constructed such that $p(s_1, \ldots, s_n)\theta$ and $p(t_1, \ldots, t_n)\theta$ are identical. Such a substitution is called a *unifier* and the problem of finding unifiers will be discussed in the next section. The new goal is constructed from the old one by replacing the selected atom by the set of body atoms of the clause and applying θ to all

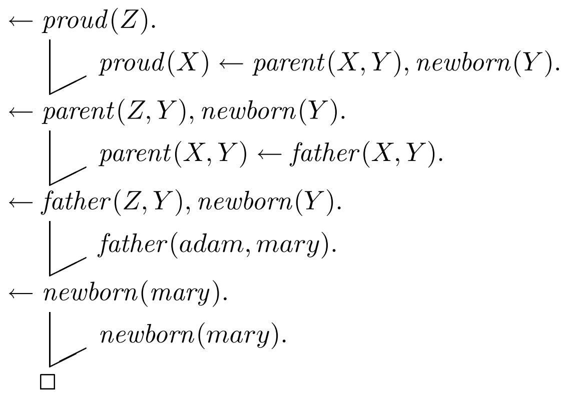

$\leftarrow proud(Z).$
$proud(X) \leftarrow parent(X,Y), newborn(Y).$
$\leftarrow parent(Z,Y), newborn(Y).$
$parent(X,Y) \leftarrow father(X,Y).$
$\leftarrow father(Z,Y), newborn(Y).$
$father(adam, mary).$
$\leftarrow newborn(mary).$
$newborn(mary).$
$\Box$

Figure 3.1: Refutation of $\leftarrow proud(Z)$**.**

atoms obtained in that way. This basic computation step can be seen as an inference rule since it transforms logic formulas. It will be called the resolution principle for definite programs or *SLD-resolution* principle. As illustrated above it combines in a special way *modus ponens* with the elimination rule for the universal quantifier.

At the last step of reasoning the empty goal, corresponding to falsity, is obtained. The final conclusion then is the negation of the initial goal. Since this goal is of the form $\forall\,\neg(A_1 \wedge \cdots \wedge A_m)$, the conclusion is equivalent (by DeMorgan's laws) to the formula $\exists(A_1 \wedge \cdots \wedge A_m)$. The final conclusion can be obtained by the inference rule known as *reductio ad absurdum.* Every step of reasoning produces a substitution. Unsatisfiability of the original goal $\leftarrow A_1, \ldots, A_m$ with P is demonstrated in k steps by showing that its instance:

$$\leftarrow (A_1, \ldots, A_m)\theta_1 \cdots \theta_k$$

is unsatisfiable, or equivalently that:

$$P \models (A_1 \wedge \cdots \wedge A_m)\theta_1 \cdots \theta_k$$

In the example discussed, the goal "Nobody is proud" is unsatisfiable with P since its instance "Adam is not proud" is unsatisfiable with P. In other words — in every model of P the sentence "Adam is proud" is true.

It is worth noticing that the unifiers may leave some variables unbound. In this case the universal closure of $(A_1 \wedge \cdots \wedge A_m)\theta_1 \cdots \theta_k$ is a logical consequence of P. Examples of such answers will appear below.

Notice also that generally the computation steps are not *deterministic* — any atom of a goal may be selected and there may be several clauses matching the selected atom. Another potential source of non-determinism concerns the existence of alternative unifiers for two atoms. These remarks suggest that it may be possible to construct (sometimes infinitely) many solutions, i.e. counter-examples for the initial goal. On the other hand it may also happen that the selected atom has no matching clause.

If so, it means that, using this method, it is not possible to construct any counter-example for the initial goal. The computation may also loop without producing any solution.

3.2 Unification

As demonstrated in the previous section, one of the main ingredients in the inference mechanism is the process of making two atomic formulas syntactically equivalent. Before defining the notion of SLD-resolution we focus on this process, called *unification*, and give an algorithmic solution — a procedure that takes two atomic formulas as input, and either shows how they can be instantiated to identical atoms or, reports a failure.

Before considering the problem of unifying atoms (and terms), consider an ordinary equation over the natural numbers ($\mathbb{N}$) such as:

$$2x + 3 \doteq 4y + 7 \tag{5}$$

The equation has a set of *solutions*; that is, valuations $\varphi\colon \{x, y\} \to \mathbb{N}$ such that $\varphi_{\Im}(2x+3) = \varphi_{\Im}(4y+7)$ where $\Im$ is the standard interpretation of the arithmetic symbols. In this particular example there are infinitely many solutions ($\{x \mapsto 2, y \mapsto 0\}$ and $\{x \mapsto 4, y \mapsto 1\}$ etc.) but by a sequence of syntactic transformations that preserve the set of all solutions the equation may be transformed into an new equation that compactly represents *all* solutions to the original equation:

$$x \doteq 2(y+1) \tag{6}$$

The transformations exploit domain knowledge (such as commutativity, associativity etc.) specific to the particular interpretation. In a logic program there is generally no such knowledge available and the question arises how to compute the solutions of an equation without *any* knowledge about the interpretation of the symbols. For example:

$$f(X, g(Y)) \doteq f(a, g(X)) \tag{7}$$

Clearly it is no longer possible to apply all the transformations that were applied above since the interpretation of $f/2$, $g/1$ is no longer fixed. However, any solution of the equations:

$$\{X \doteq a, g(Y) \doteq g(X)\} \tag{8}$$

must clearly be a solution of equation (7). Similarly, any solution of:

$$\{X \doteq a, Y \doteq X\} \tag{9}$$

must be a solution of equations (8). Finally any solution of:

$$\{X \doteq a, Y \doteq a\} \tag{10}$$

is a solution of (9). By analogy to (6) this is a compact representation of *some* solutions to equation (7). However, whether it represents *all* solution depends on how the symbols $f/2$, $g/1$ and a are interpreted. For example, if $f/2$ denotes integer addition, $g/1$ the successor function and a the integer zero, then (10) represents only one solution to equation (7). However, equation (7) has infinitely many integer solutions — any φ such that $\varphi(Y) = 0$ is a solution.

On the other hand, consider a Herbrand interpretation $\Im$; Solving of an equation $s \doteq t$ amounts to finding a valuation φ such that $\varphi_\Im(s) = \varphi_\Im(t)$. Now a valuation in the Herbrand domain is a mapping from variables of the equations to ground terms (that is, a substitution) and the interpretation of a ground term is the term itself. Thus, a solution in the Herbrand domain is a grounding substitution φ such that $s\varphi$ and $t\varphi$ are identical ground terms. This brings us to the fundamental concept of unification and unifiers:

Definition 3.1 (Unifier) Let s and t be terms. A substitution θ such that $s\theta$ and $t\theta$ are identical (denoted $s\theta = t\theta$) is called a *unifier* of s and t. ▮

The search for a unifier of two terms, s and t, will be viewed as the process of solving the equation $s \doteq t$. Therefore, more generally, if $\{s_1 \doteq t_1, \ldots, s_n \doteq t_n\}$ is a set of equations, then θ is called a unifier of the set if $s_i\theta = t_i\theta$ for all $1 \leq i \leq n$. For instance, the substitution $\{X/a, Y/a\}$ is a unifier of equation (7). It is also a unifier of (8)–(10). In fact, it is the only unifier as long as "irrelevant" variables are not introduced. (For instance, $\{X/a, Y/a, Z/a\}$ is also a unifier.) The transformations informally used in steps (7)–(10) preserve the set of all solutions in the Herbrand domain. (The full set of transformations will soon be presented.) Note that a solution to a set of equations is a (grounding) unifier. Thus, if a set of equations has a unifier then the set also has a solution.

However, not all sets of equations have a solution/unifier. For instance, the set $\{sum(1, 1) \doteq 2\}$ is not unifiable. Intuitively *sum* may be thought of as integer addition, but bear in mind that the symbols have no predefined interpretation in a logic program. (In Chapters 13–14 more powerful notions of unification are discussed.)

It is often the case that a set of equations have more than one unifier. For instance, both $\{X/g(Z), Y/Z\}$ and $\{X/g(a), Y/a, Z/a\}$ are unifiers of the set $\{f(X, Y) \doteq f(g(Z), Z)\}$. Under the first unifier the terms instantiate to $f(g(Z), Z)$ and under the second unifier the terms instantiate to $f(g(a), a)$. The second unifier is in a sense more restrictive than the first, as it makes the two terms ground whereas the first still provides room for some alternatives in that is does not specify how Z should be bound. We say that $\{X/g(Z), Y/Z\}$ is *more general* than $\{X/g(a), Y/a, Z/a\}$. More formally this can be expressed as follows:

Definition 3.2 (Generality of substitutions) A substitution θ is said to be *more general* than a substitution σ (denoted $\sigma \preceq \theta$) iff there exists a substitution ω such that $\sigma = \theta\omega$. ▮

Definition 3.3 (Most general unifier) A unifier θ is said to be a most general unifier (mgu) of two terms iff θ is more general than any other unifier of the terms. ▮

Definition 3.4 (Solved form) A set of equations $\{X_1 \doteq t_1, \ldots, X_n \doteq t_n\}$ is said to be in *solved form* iff $X_1, \ldots, X_n$ are distinct variables none of which appear in $t_1, \ldots, t_n$. ▮

There is a close correspondence between a set of equations in solved form and the most general unifier(s) of that set as shown by the following theorem:

Proposition 3.5 Let $\{X_1 \doteq t_1, \ldots, X_n \doteq t_n\}$ be a set of equations in solved form. Then $\{X_1/t_1, \ldots, X_n/t_n\}$ is an (idempotent) mgu of the solved form. ▮

Proof: First define:

$$\mathcal{E} := \{X_1 \doteq t_1, \ldots, X_n \doteq t_n\}$$
$$\theta := \{X_1/t_1, \ldots, X_n/t_n\}$$

Clearly θ is an idempotent unifier of $\mathcal{E}$. It remains to be shown that θ is more general than any other unifier of $\mathcal{E}$.

Thus, assume that σ is a unifier of $\mathcal{E}$. Then $X_i\sigma = t_i\sigma$ for $1 \leq i \leq n$. It must follow that $X_i/t_i\sigma \in \sigma$ for $1 \leq i \leq n$. In addition σ may contain some additonal pairs $Y_1/s_1, \ldots, Y_m/s_m$ such that $\{X_1, \ldots, X_n\} \cap \{Y_1, \ldots, Y_m\} = \varnothing$. Thus, σ is of the form:

$$\{X_1/t_1\sigma, \ldots, X_n/t_n\sigma, Y_1/s_1, \ldots, Y_m/s_m\}$$

Now $\theta\sigma = \sigma$. Thus, there exists a substitution ω (viz. σ) such that $\sigma = \theta\omega$. Therefore, θ is an idempotent mgu. ▮

Definition 3.6 (Equivalence of sets of equations) Two sets of equations $\mathcal{E}_1$ and $\mathcal{E}_2$ are said to be *equivalent* if they have the same set of unifiers. ▮

Note that two equivalent sets of equations must have the same set of solutions in any Herbrand interpretation.

The definition can be used as follows: to compute a most general unifier $\text{MGU}(s, t)$ of two terms s and t, first try to transform the equation $\{s \doteq t\}$ into an equivalent solved form. If this fails then $\text{MGU}(s, t) =$ **failure**. However, if there is a solved form $\{X_1 \doteq t_1, \ldots, X_n \doteq t_n\}$ then $\text{MGU}(s, t) = \{X_1/t_1, \ldots, X_n/t_n\}$.

Figure 3.2 presents a (non-deterministic) algorithm which takes as input a set of equations $\mathcal{E}$ and terminates returning either a solved form equivalent to $\mathcal{E}$ or **failure** if no such solved form exists. Note that constants are viewed as function symbols of arity 0. Thus, if an equation $c \doteq c$ gets selected, the equation is simply removed by case 1. Before proving the correctness of the algorithm some examples are used to illustrate the idea:

Example 3.7 The set $\{f(X, g(Y)) \doteq f(g(Z), Z)\}$ has a solved form since:

$$\begin{aligned} \{f(X, g(Y)) \doteq f(g(Z), Z)\} &\Rightarrow \{X \doteq g(Z), g(Y) \doteq Z\} \\ &\Rightarrow \{X \doteq g(Z), Z \doteq g(Y)\} \\ &\Rightarrow \{X \doteq g(g(Y)), Z \doteq g(Y)\} \end{aligned}$$

The set $\{f(X, g(X), b) \doteq f(a, g(Z), Z)\}$, on the other hand, does not have a solved form since:

$$\begin{aligned} \{f(X, g(X), b) \doteq f(a, g(Z), Z)\} &\Rightarrow \{X \doteq a, g(X) \doteq g(Z), b \doteq Z\} \\ &\Rightarrow \{X \doteq a, g(a) \doteq g(Z), b \doteq Z\} \end{aligned}$$

```
Input: A set $\mathcal{E}$ of equations.
Output: An equivalent set of equations in solved form or **failure**.

repeat
    select an arbitrary $s \doteq t \in \mathcal{E}$;
    case $s \doteq t$ of
        $f(s_1, \ldots, s_n) \doteq f(t_1, \ldots, t_n)$ where $n \geq 0 \Rightarrow$
                replace equation by $s_1 \doteq t_1, \ldots, s_n \doteq t_n$;        % case 1
        $f(s_1, \ldots, s_m) \doteq g(t_1, \ldots, t_n)$ where $f/m \neq g/n \Rightarrow$
                halt with failure;                                                  % case 2
        $X \doteq X \Rightarrow$
                remove the equation;                                                % case 3
        $t \doteq X$ where $t$ is not a variable $\Rightarrow$
                replace equation by $X \doteq t$;                                   % case 4
        $X \doteq t$ where $X \neq t$ and $X$ has more than one occurrence in $\mathcal{E} \Rightarrow$
                if $X$ is a proper subterm of $t$ then
                    halt with failure                                               % case 5a
                else
                    replace all other occurrences of $X$ by $t$;                    % case 5b
    esac
until no action is possible on any equation in $\mathcal{E}$;
halt with $\mathcal{E}$;
```

Figure 3.2: Solved form algorithm

$$\begin{aligned}
&\Rightarrow \{X \doteq a, a \doteq Z, b \doteq Z\}\\
&\Rightarrow \{X \doteq a, Z \doteq a, b \doteq Z\}\\
&\Rightarrow \{X \doteq a, Z \doteq a, b \doteq a\}\\
&\Rightarrow \textbf{failure}
\end{aligned}$$

The algorithm fails since case 2 applies to $b \doteq a$. Finally consider:

$$\begin{aligned}
\{f(X, g(X)) \doteq f(Z, Z)\} &\Rightarrow \{X \doteq Z, g(X) \doteq Z\}\\
&\Rightarrow \{X \doteq Z, g(Z) \doteq Z\}\\
&\Rightarrow \{X \doteq Z, Z \doteq g(Z)\}\\
&\Rightarrow \textbf{failure}
\end{aligned}$$

The set does not have a solved form since Z is a proper subterm of $g(Z)$. ∎

Theorem 3.8 The solved form algorithm in Figure 3.2 terminates and returns an equivalent solved form or **failure** if no such solved form exists. ∎

Proof: First consider termination: Note that case 5b is the only case that may increase the number of symbol occurrences in the set of equations. However, case 5b can be applied at most once for each variable X. Thus, case 5b can be applied only a finite number of times and may introduce only a finite number of new symbol occurrences. Case 2 and case 5a terminate immediately and case 1 and 3 strictly decrease the number of symbol occurrences in the set. Since case 4 cannot be applied indefinitely, but has to be intertwined with the other cases it follows that the algorithm always terminates.

It should be evident that the algorithm either returns **failure** or a set of equations in solved form. Thus, it remains to be shown that each iteration of the algorithm preserves equivalence between successive sets of equations. It is easy to see that if case 2 or 5a apply to some equation in:

$$\{s_1 \doteq t_1, \ldots, s_n \doteq t_n\} \tag{$\mathcal{E}_1$}$$

then the set cannot possibly have a unifier. It is also easy to see that if any of case 1, 3 or 4 apply, then the new set of equations has the same set of unifiers. Finally assume that case 5b applies to some equation $s_i \doteq t_i$. Then the new set is of the form:

$$\{s_1\theta \doteq t_1\theta, \ldots, s_{i-1}\theta \doteq t_{i-1}\theta, s_i \doteq t_i, s_{i+1}\theta \doteq t_{i+1}\theta, \ldots s_n\theta \doteq t_n\theta\} \tag{$\mathcal{E}_2$}$$

where $\theta := \{s_i/t_i\}$. First assume that σ is a unifier of $\mathcal{E}_1$ — that is, $s_j\sigma = t_j\sigma$ for every $1 \leq j \leq n$. In particular, it must hold that $s_i\sigma = t_i\sigma$. Since s_i is a variable which is not a subterm of t_i it must follow that $s_i/t_i\sigma \in \sigma$. Moreover, $\theta\sigma = \sigma$ and it therefore follows that σ is a unifier also of $\mathcal{E}_2$.

Next, assume that σ is a unifier of $\mathcal{E}_2$. Thus, $s_i/t_i\sigma \in \sigma$ and $\theta\sigma = \sigma$ which must then be a unifier also of $\mathcal{E}_1$. ∎

The algorithm presented in Figure 3.2 may be very inefficient. One of the reasons is case 5a; That is, checking if a variable X occurs inside another term t. This is often referred to as the *occur-check*. Assume that the time of occur-check is linear with respect to the size $|t|$ of t.[1] Consider application of the solved form algorithm to the equation:

$$g(X_1, \ldots, X_n) \doteq g(f(X_0, X_0), f(X_1, X_1), \ldots, f(X_{n-1}, X_{n-1}))$$

where $X_0, \ldots, X_n$ are distinct. By case 1 this reduces to:

$$\{X_1 \doteq f(X_0, X_0), X_2 \doteq f(X_1, X_1), \ldots, X_n \doteq f(X_{n-1}, X_{n-1})\}$$

Assume that the equation selected in step i is of the form $X_i = f(\ldots, \ldots)$. Then in the k-th iteration the selected equation is of the form $X_k \doteq \mathcal{T}_k$ where $\mathcal{T}_{i+1} := f(\mathcal{T}_i, \mathcal{T}_i)$ and $\mathcal{T}_0 := X_0$. Hence, $|\mathcal{T}_{i+1}| = 2|\mathcal{T}_i| + 1$. That is, $|\mathcal{T}_n| > 2^n$. This shows the exponential dependency of the unification time on the length of the structures. In this example the growth of the argument lengths is caused by duplication of subterms. As a matter of fact, the same check is repeated many times. Something that could be avoided by sharing various instances of the same structure. In the literature one can find linear algorithms but they are sometimes quite elaborate. On the other hand, Prolog systems usually "solve" the problem simply by omitting the occur-check during unification. Roughly speaking such an approach corresponds to a solved form algorithm where case 5a–b is replaced by:

[1]The size of a term is the total number of constant, variable and functor occurrences in t.

```
$X \doteq t$ where $X \neq t$ and $X$ has more than one occurrence in $\mathcal{E} \Rightarrow$
        replace all other occurrences of $X$ by $t$;            % *case 5*
```

A pragmatic justification for this solution is the fact that rule 5a (occur check) never is used during the computation of many Prolog programs. There are sufficient conditions which guarantee this, but in general this property is undecidable. The ISO Prolog standard (1995) states that the result of unification is undefined if case 5b can be applied to the set of equations. Strictly speaking, removing case 5a causes looping of the algorithm on equations where case 5a would otherwise apply. For example, an attempt to solve $X \doteq f(X)$ by the modified algorithm will produce a new equation $X \doteq f(f(X))$. However, case 5 is once again applicable yielding $X \doteq f(f(f(f(X))))$ and so forth. In practice many Prolog systems do not loop, but simply bind X to the infinite structure $f(f(f(\ldots)))$. (The notation $X/f(\infty)$ will be used to denote this binding.) Clearly, $\{X/f(\infty)\}$ is an infinite "unifier" of X and $f(X)$. It can easily be represented in the computer by a finite cyclic data structure. But this amounts to generalization of the concepts of term, substitution and unifier for the infinite case not treated in classical logic. Implementation of unification without occur-check may result in unsoundness as will be illustrated in Example 3.21.

Before concluding the discussion about unification we study the notion of most general unifier in more detail. It turns out that the notion of mgu is a subtle one; For instance, there is generally not a unique most general unifier of two terms s and t. A trivial example is the equation $f(X) \doteq f(Y)$ which has at least two mgu's; namely $\{X/Y\}$ and $\{Y/X\}$. Part of the confusion stems from the fact that $\preceq$ ("being more general than") is not an ordering relation. It is reflexive: That is, any substitution θ is "more general" than itself since $\theta = \theta\epsilon$. As might be expected it is also transitive: If $\theta_1 = \theta_2\omega_1$ and $\theta_2 = \theta_3\omega_2$ then obviously $\theta_1 = \theta_3\omega_2\omega_1$. However, $\preceq$ is not anti-symmetric. For instance, consider the substitution $\theta := \{X/Y, Y/X\}$ and the identity substitution ϵ. The latter is obviously more general than θ since $\theta = \epsilon\theta$. But θ is also more general than ϵ, since $\epsilon = \theta\theta$. It may seem odd that two distinct substitutions are more general than one another. Still there is a rational explanation. First consider the following definition:

Definition 3.9 (Renaming) A substitution $\{X_1/Y_1, \ldots, X_n/Y_n\}$ is called a *renaming substitution* iff $Y_1, \ldots, Y_n$ is a permutation of $X_1, \ldots, X_n$. ▮

A renaming substitution represents a *bijective* mapping between variables (or more generally terms). Such a substitution always preserves the structure of a term; if θ is a renaming and t a term, then $t\theta$ and t are equivalent but for the names of the variables. Now, the fact that a renaming represents a bijection implies that there must be an inverse mapping. Indeed, if $\{X_1/Y_1, \ldots, X_n/Y_n\}$ is a renaming then $\{Y_1/X_1, \ldots, Y_n/X_n\}$ is its inverse. We denote the inverse of θ by θ^{-1} and observe that $\theta\theta^{-1} = \theta^{-1}\theta = \epsilon$.

Proposition 3.10 Let θ be an mgu of s and t and assume that ω is a renaming. Then $\theta\omega$ is an mgu of s and t. ▮

The proof of the proposition is left as an exercise. So is the proof of the following proposition:

Proposition 3.11 Let θ and σ be substitutions. If $\theta \preceq \sigma$ and $\sigma \preceq \theta$ then there exists a renaming substitution ω such that $\sigma = \theta\omega$ (and $\theta = \sigma\omega^{-1}$). ∎

Thus, according to the above propositions, the set of all mgu's of two terms is closed under renaming.

3.3 SLD-Resolution

The method of reasoning discussed informally in Section 3.1 can be summarized as the following inference rule:

$$\frac{\forall\,\neg\,(A_1 \wedge \cdots \wedge A_{i-1} \wedge A_i \wedge A_{i+1} \wedge \cdots \wedge A_m) \qquad \forall\,(B_0 \leftarrow B_1 \wedge \cdots \wedge B_n)}{\forall\,\neg\,(A_1 \wedge \cdots \wedge A_{i-1} \wedge B_1 \wedge \cdots \wedge B_n \wedge A_{i+1} \wedge \cdots \wedge A_m)\theta}$$

or (using logic programming notation):

$$\frac{\leftarrow A_1, \ldots, A_{i-1}, A_i, A_{i+1}, \ldots, A_m \qquad B_0 \leftarrow B_1, \ldots, B_n}{\leftarrow (A_1, \ldots, A_{i-1}, B_1, \ldots, B_n, A_{i+1}, \ldots, A_m)\theta}$$

where

(i) $A_1, \ldots, A_m$ are atomic formulas;

(ii) $B_0 \leftarrow B_1, \ldots, B_n$ is a (renamed) definite clause in P ($n \geq 0$);

(iii) $\text{MGU}(A_i, B_0) = \theta$.

The rule has two premises — a goal clause and a definite clause. Notice that each of them is separately universally quantified. Thus the scopes of the quantifiers are disjoint. On the other hand, there is only one universal quantifier in the conclusion of the rule. Therefore it is required that the sets of variables in the premises are disjoint. Since all variables of the premises are bound it is always possible to *rename* the variables of the definite clause to satisfy this requirement (that is, to apply some renaming substitution to it).

The goal clause may include several atomic formulas which unify with the head of some clause in the program. In this case it may be desirable to introduce some deterministic choice of the selected atom A_i for unification. In what follows it is assumed that this is given by some function which for a given goal selects the subgoal for unification. The function is called the *selection function* or the *computation rule.* It is sometimes desirable to generalize this concept so that, in one situation, the computation rule selects one subgoal from a goal G but, in another situation, selects another subgoal from G. In that case the computation rule is not a function on goals but something more complicated. However, for the purpose of this book this extra generality is not needed.

The inference rule presented above is the only one needed for definite programs. It is a version of the inference rule called the *resolution principle*, which was introduced by J. A. Robinson in 1965. The resolution principle applies to clauses. Since definite clauses are restricted clauses the corresponding restricted form of resolution presented below is called *SLD-resolution* (Linear resolution for Definite clauses with Selection function).

Next the use of the SLD-resolution principle is discussed for a given definite program P. The starting point, as exemplified in Section 3.1, is a definite goal clause G_0 of the form:

$$\leftarrow A_1, \dots, A_m \qquad (m \geq 0)$$

From this goal a subgoal A_i is selected (if possible) by the computation rule. A new goal clause G_1 is constructed by selecting (if possible) some renamed program clause $B_0 \leftarrow B_1, \dots, B_n$ $(n \geq 0)$ whose head unifies with A_i (resulting in an mgu θ_1). If so, G_1 will be of the form:

$$\leftarrow (A_1, \dots, A_{i-1}, B_1, \dots, B_n, A_{i+1}, \dots, A_m)\theta_1$$

(According to the requirement above, the variables of the program clause are being renamed so that they are different from those of G_0.) Now it is possible to apply the resolution principle to G_1 thus obtaining G_2, etc. This process may or may not terminate. There are two cases when it is not possible to obtain G_{i+1} from G_i:

- the first is when the selected subgoal cannot be resolved (i.e. is not unifiable) with the head of any program clause;
- the other case appears when $G_i = \Box$ (i.e. the empty goal).

The process described above results in a finite or infinite sequence of goals starting with the initial goal. At every step a program clause (with renamed variables) is used to resolve the subgoal selected by the computation rule $\Re$ and an mgu is created. Thus, the full record of a reasoning step would be a pair $\langle G_i, C_i \rangle$, $i \geq 0$, where G_i is a goal and C_i a program clause with renamed variables. Clearly, the computation rule $\Re$ together with G_i and C_i determines (up to renaming of variables) the mgu (to be denoted θ_{i+1}) produced at the $(i+1)$-th step of the process. A goal G_{i+1} is said to be *derived* (*directly*) from G_i and C_i via $\Re$ (or alternatively, G_i and C_i *resolve* into G_{i+1}).

Definition 3.12 (SLD-derivation) Let G_0 be a definite goal, P a definite program and $\Re$ a computation rule. An *SLD-derivation* of G_0 (using P and $\Re$) is a finite or infinite sequence of goals:

$$G_0 \overset{C_0}{\rightsquigarrow} G_1 \cdots G_{n-1} \overset{C_{n-1}}{\rightsquigarrow} G_n \dots$$

where each G_{i+1} is derived directly from G_i and a renamed program clause C_i via $\Re$. ▮

Note that since there are usually infinitely many ways of renaming a clause there are formally infinitely many derivations. However, some of the derivations differ only in the names of the variables used. To avoid some technical problems and to make the renaming of variables in a derivation consistent, the variables in the clause C_i of a derivation are renamed by adding the subscript i to *every* variable in the clause. In what follows we consider only derivations where this renaming strategy is used.

Each finite SLD-derivation of the form:

$$G_0 \overset{C_0}{\rightsquigarrow} G_1 \cdots G_{n-1} \overset{C_{n-1}}{\rightsquigarrow} G_n$$

yields a sequence $\theta_1, \ldots, \theta_n$ of mgu's. The composition

$$\theta := \begin{cases} \theta_1\theta_2\cdots\theta_n & \text{if } n > 0 \\ \epsilon & \text{if } n = 0 \end{cases}$$

of mgu's is called the *computed substitution* of the derivation.

Example 3.13 Consider the initial goal $\leftarrow proud(Z)$ and the program discussed in Section 3.1.

$$\begin{array}{lcl} G_0 & : & \leftarrow proud(Z). \\ C_0 & : & proud(X_0) \leftarrow parent(X_0, Y_0), newborn(Y_0). \end{array}$$

Unification of $proud(Z)$ and $proud(X_0)$ yields e.g. the mgu $\theta_1 = \{X_0/Z\}$. Assume that a computation rule which always selects the leftmost subgoal is used (if nothing else is said, this computation rule is used also in what follows). Such a computation rule will occasionally be referred to as *Prolog's* computation rule since this is the computation rule used by most Prolog systems. The first derivation step yields:

$$\begin{array}{lcl} G_1 & : & \leftarrow parent(Z, Y_0), newborn(Y_0). \\ C_1 & : & parent(X_1, Y_1) \leftarrow father(X_1, Y_1). \end{array}$$

In the second resolution step the mgu $\theta_2 = \{X_1/Z, Y_1/Y_0\}$ is obtained. The derivation then proceeds as follows:

$$\begin{array}{lcl} G_2 & : & \leftarrow father(Z, Y_0), newborn(Y_0). \\ C_2 & : & father(adam, mary). \end{array}$$

$$\begin{array}{lcl} G_3 & : & \leftarrow newborn(mary). \\ C_3 & : & newborn(mary). \end{array}$$

$$G_4 \; : \; \square$$

The computed substitution of this derivation is:

$$\begin{array}{rcl} \theta_1\theta_2\theta_3\theta_4 & = & \{X_0/Z\}\{X_1/Z, Y_1/Y_0\}\{Z/adam, Y_0/mary\}\epsilon \\ & = & \{X_0/adam, X_1/adam, Y_1/mary, Z/adam, Y_0/mary\} \end{array}$$

A derivation like the one above is often represented graphically as in Figure 3.1. ▮

Example 3.14 Consider the following definite program:

$$\begin{array}{rl} 1: & grandfather(X, Z) \leftarrow father(X, Y), parent(Y, Z). \\ 2: & parent(X, Y) \leftarrow father(X, Y). \\ 3: & parent(X, Y) \leftarrow mother(X, Y). \\ 4: & father(a, b). \\ 5: & mother(b, c). \end{array}$$

$\leftarrow grandfather(a, X).$

$\quad grandfather(X_0, Z_0) \leftarrow father(X_0, Y_0), parent(Y_0, Z_0).$

$\leftarrow father(a, Y_0), parent(Y_0, X).$

$\quad father(a, b).$

$\leftarrow parent(b, X).$

$\quad parent(X_2, Y_2) \leftarrow mother(X_2, Y_2).$

$\leftarrow mother(b, X).$

$\quad mother(b, c).$

$\Box$

Figure 3.3: SLD-derivation

Figure 3.3 depicts a finite SLD-derivation of the goal $\leftarrow grandfather(a, X)$ (again using Prolog's computation rule). ▮

SLD-derivations that end in the empty goal (and the bindings of variables in the initial goal of such derivations) are of special importance since they correspond to refutations of (and provide answers to) the initial goal:

Definition 3.15 (SLD-refutation) A (finite) SLD-derivation:

$$G_0 \overset{C_0}{\leadsto} G_1 \cdots G_n \overset{C_n}{\leadsto} G_{n+1}$$

where $G_{n+1} = \Box$ is called an *SLD-refutation* of G_0. ▮

Definition 3.16 (Computed answer substitution) The computed substitution of an SLD-refutation of G_0 restricted to the variables in G_0 is called a *computed answer substitution* for G_0. ▮

In Examples 3.13 and 3.14 the computed answer substitutions are $\{Z/adam\}$ and $\{X/c\}$ respectively.

For a given initial goal G_0 and computation rule, the sequence $G_1, \ldots, G_{n+1}$ of goals in a finite derivation $G_0 \leadsto G_1 \cdots G_n \leadsto G_{n+1}$ is determined (up to renaming of variables) by the sequence $C_0, \ldots, C_n$ of (renamed) program clauses used. This is particularly interesting in the case of refutations. Let:

$$G_0 \overset{C_0}{\leadsto} G_1 \cdots G_n \overset{C_n}{\leadsto} \Box$$

be a refutation. It turns out that if the computation rule is changed there still exists another refutation:

$$G_0 \overset{C_0'}{\leadsto} G_1' \cdots G_n' \overset{C_n'}{\leadsto} \Box$$

$$\begin{tikzcd}[column sep=small]
\leftarrow \mathit{grandfather}(a, X). \arrow[d] & \mathit{grandfather}(X_0, Z_0) \leftarrow \mathit{father}(X_0, Y_0), \mathit{parent}(Y_0, Z_0). \\
\leftarrow \mathit{father}(a, Y_0), \mathit{parent}(Y_0, X). \arrow[d] & \mathit{father}(a, b). \\
\leftarrow \mathit{parent}(b, X). \arrow[d] & \mathit{parent}(X_2, Y_2) \leftarrow \mathit{father}(X_2, Y_2). \\
\leftarrow \mathit{father}(b, X). &
\end{tikzcd}$$

Figure 3.4: Failed SLD-derivation

of G_0 which has the same computed answer substitution (up to renaming of variables) and where the sequence $C'_0, \ldots, C'_n$ of clauses used is a permutation of the sequence $C_0, \ldots, C_n$. This property will be called *independence of the computation rule* and it will be discussed further in Section 3.6.

Not all SLD-derivations lead to refutations. As already pointed out, if the selected subgoal cannot be unified with any clause, it is not possible to extend the derivation any further:

Definition 3.17 (Failed derivation) A derivation of a goal clause G_0 whose last element is not empty and cannot be resolved with any clause of the program is called a *failed* derivation. ∎

Figure 3.4 depicts a failed derivation of the program and goal in Example 3.14. Since the selected literal (the leftmost one) does not unify with the head of any clause in the program, the derivation is failed. Note that a derivation is failed even if there is some other subgoal but the selected one which unifies with a clause head.

By a *complete derivation* we mean a *refutation*, a *failed derivation* or an *infinite derivation*. As shown above, a given initial goal clause G_0 may have many complete derivations via a given computation rule $\Re$. This happens if the selected subgoal of some goal can be resolved with more than one program clause. All such derivations may be represented by a possibly infinite tree called the SLD-tree of G_0 (using P and $\Re$).

Definition 3.18 (SLD-tree) Let P be a definite program, G_0 a definite goal and $\Re$ a computation rule. The SLD-tree of G_0 (using P and $\Re$) is a (possibly infinite) labelled tree satisfying the following conditions:

- the root of the tree is labelled by G_0;
- if the tree contains a node labelled by G_i and there is a renamed clause $C_i \in P$ such that G_{i+1} is derived from G_i and C_i via $\Re$ then the node labelled by G_i has a child labelled by G_{i+1}. The edge connecting them is labelled by C_i.

∎

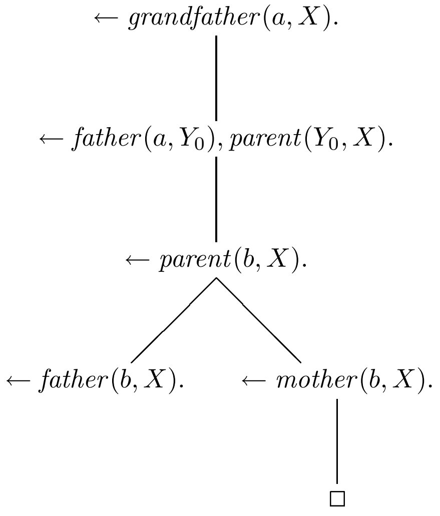

$\leftarrow \mathit{grandfather}(a, X).$
$\leftarrow \mathit{father}(a, Y_0), \mathit{parent}(Y_0, X).$
$\leftarrow \mathit{parent}(b, X).$
$\leftarrow \mathit{father}(b, X).$
$\leftarrow \mathit{mother}(b, X).$
$\Box$

Figure 3.5: SLD-tree of $\leftarrow \mathit{grandfather}(a, X)$

The nodes of an SLD-tree are thus labelled by goals of a derivation. The edges are labelled by the clauses of the program. There is in fact a one-to-one correspondence between the paths of the SLD-tree and the complete derivations of G_0 under a fixed computation rule $\Re$. The sequence:

$$G_0 \overset{C_0}{\rightsquigarrow} G_1 \cdots G_k \overset{C_k}{\rightsquigarrow} \cdots$$

is a complete derivation of G_0 via $\Re$ iff there exists a path of the SLD-tree of the form $G_0, G_1, \ldots, G_k, \ldots$ such that for every i, the edge $\langle G_i, G_{i+1} \rangle$ is labelled by C_i. Usually this label is abbreviated (e.g. by numbering the clauses of the program) or omitted when drawing the tree. Additional labelling with the mgu θ_{i+1} or some part of it may also be included.

Example 3.19 Consider again the program of Example 3.14. The SLD-tree of the goal $\leftarrow \mathit{grandfather}(a, X)$ is depicted in Figure 3.5. ∎

The SLD-trees of a goal clause G_0 are often distinct for different computation rules. It may even happen that the SLD-tree for G_0 under one computation rule is finite whereas the SLD-tree of the same goal under another computation rule is infinite. However, the independence of computation rules means that for every refutation path in one SLD-tree there exists a refutation path in the other SLD-tree with the same length and with the same computed answer substitution (up to renaming). The sequences of clauses labelling both paths are permutations of one another.

3.4 Soundness of SLD-resolution

The method of reasoning presented informally in Section 3.1 was formalized as the SLD-resolution principle in the previous section. As a matter of fact one more inference rule is used after construction of a refutation. It applies the computed substitution of the refutation to the body of the initial goal to get the final conclusion. This is the most interesting part of the process since if the initial goal is seen as a query, the computed substitution of the refutation restricted to its variables is an answer to this query. It is therefore called a computed answer substitution. In this context it is also worth noticing the case when no answer substitution exists for a given query. Prolog systems may sometimes discover this and deliver a "no" answer. The logical meaning of "no" will be discussed in the next chapter.

As discussed in Chapter 1, the introduction of formal inference rules raises the questions of their *soundness* and *completeness*. Soundness is an essential property which guarantees that the conclusions produced by the system are correct. Correctness in this context means that they are logical consequences of the program. That is, that they are true in every model of the program. Recall the discussion of Chapter 2 — a definite program describes many "worlds" (i.e. models), including the one which is meant by the user, the intended model. Soundness is necessary to be sure that the conclusions produced by any refutation are true in every world described by the program, in particular in the intended one.

This raises the question concerning the soundness of the SLD-resolution principle. The discussion in Section 3.1 gives some arguments which may by used in a formal proof. However, the intermediate conclusions produced at every step of refutation are of little interest for the user of a definite program. Therefore the soundness of SLD-resolution is usually understood as correctness of computed answer substitutions. This can be stated as the following theorem (due to Clark (1979)).

Theorem 3.20 (Soundness of SLD-resolution) Let P be a definite program, $\Re$ a computation rule and θ an $\Re$-computed answer substitution for a goal $\leftarrow A_1, \ldots, A_m$. Then $\forall\,((A_1 \wedge \cdots \wedge A_m)\theta)$ is a logical consequence of the program. ▮

Proof: Any computed answer substitution is obtained by a refutation of the goal via $\Re$. The proof is based on induction over the number of resolution steps of the refutation.

First consider refutations of length one. This is possible only if $m = 1$ and A_1 resolves with some fact A with the mgu θ_1. Hence $A_1\theta_1$ is an instance of A. Now let θ be θ_1 restricted to the variables in A_1. Then $A_1\theta = A_1\theta_1$. It is a well-known fact that the universal closure of an instance of a formula F is a logical consequence of the universal closure of F (cf. exercise 1.9, Chapter 1). Hence the universal closure of $A_1\theta$ is a logical consequence of the clause A and consequently of the program P.

Next, assume that the theorem holds for refutations with $n - 1$ steps. Take a refutation with n steps of the form:

$$G_0 \overset{C_0}{\leadsto} G_1 \cdots G_{n-1} \overset{C_{n-1}}{\leadsto} \Box$$

where G_0 is the original goal clause $\leftarrow A_1, \ldots, A_m$.

Now, assume that A_j is the selected atom in the first derivation step and that C_0 is a (renamed) clause $B_0 \leftarrow B_1, \ldots, B_k$ $(k \geq 0)$ in P. Then $A_j\theta_1 = B_0\theta_1$ and G_1 has to be of the form:

$$\leftarrow (A_1, \ldots, A_{j-1}, B_1, \ldots, B_k, A_{j+1}, \ldots, A_m)\theta_1$$

By the induction hypothesis the formula:

$$\forall\,(A_1 \wedge \ldots \wedge A_{j-1} \wedge B_1 \wedge \ldots \wedge B_k \wedge A_{j+1} \wedge \ldots \wedge A_m)\theta_1 \cdots \theta_n \tag{11}$$

is a logical consequence of the program. It follows by definition of logical consequence that also the universal closure of:

$$(B_1 \wedge \ldots \wedge B_k)\theta_1 \cdots \theta_n \tag{12}$$

is a logical consequence of the program. By (11):

$$\forall\,(A_1 \wedge \ldots \wedge A_{j-1} \wedge A_{j+1} \wedge \ldots \wedge A_m)\theta_1 \cdots \theta_n \tag{13}$$

is a logical consequence of P. Now because of (12) and since:

$$\forall\,(B_0 \leftarrow B_1 \wedge \ldots \wedge B_k)\theta_1 \cdots \theta_n$$

is a logical consequence of the program (being an instance of a clause in P) it follows that:

$$\forall\, B_0\theta_1 \cdots \theta_n \tag{14}$$

is a logical consequence of P. Hence by (13) and (14):

$$\forall\,(A_1 \wedge \ldots \wedge A_{j-1} \wedge B_0 \wedge A_{j+1} \wedge \ldots \wedge A_m)\theta_1 \cdots \theta_n \tag{15}$$

is also a logical consequence of the program. But since θ_1 is a most general unifier of B_0 and A_j, B_0 can be replaced by A_j in (15). Now let θ be $\theta_1 \cdots \theta_n$ restricted to the variables in $A_1, \ldots, A_m$ then:

$$\forall\,(A_1 \wedge \ldots \wedge A_m)\theta$$

is a logical consequence of P, which concludes the proof. ▮

It should be noticed that the theorem does not hold if the unifier is computed by a "unification" algorithm without occur-check. For illustration consider the following example.

Example 3.21 A term is said to be *f-constructed* with a term T if it is of the form $f(T, Y)$ for any term Y. A term X is said to be *bizarre* if it is f-constructed with itself. (As discussed in Section 3.2 there are no "bizarre" terms since no term can include itself as a proper subterm.) Finally a term X is said to be *crazy* if it is the second direct substructure of a bizarre term. These statements can be formalized as the following definite program:

$$\begin{array}{l} \mathit{f\_constructed}(f(T,Y),T). \\ \mathit{bizarre}(X) \leftarrow \mathit{f\_constructed}(X,X). \\ \mathit{crazy}(X) \leftarrow \mathit{bizarre}(f(Y,X)). \end{array}$$

Now consider the goal $\leftarrow crazy(X)$ — representing the query "Are there any crazy terms?". There is only one complete SLD-derivation (up to renaming). Namely:

$$\begin{aligned}
G_0 &: \leftarrow crazy(X)\\
C_0 &: crazy(X_0) \leftarrow bizarre(f(Y_0, X_0))\\
\\
G_1 &: \leftarrow bizarre(f(Y_0, X))\\
C_1 &: bizarre(X_1) \leftarrow f\_constructed(X_1, X_1)\\
\\
G_2 &: \leftarrow f\_constructed(f(Y_0, X), f(Y_0, X))
\end{aligned}$$

The only subgoal in G_2 does not unify with the first program clause because of the occur-check. This corresponds to our expectations: Since, in the intended model, there are no bizarre terms, there cannot be any crazy terms. Since SLD-resolution is sound, if there were any answers to G_0 they would be correct also in the intended model.

Assume now that a "unification" algorithm without occur-check is used. Then the derivation can be extended as follows:

$$\begin{aligned}
G_2 &: \leftarrow f\_constructed(f(Y_0, X), f(Y_0, X))\\
C_2 &: f\_constructed(f(T_2, Y_2), T_2)\\
\\
G_3 &: \square
\end{aligned}$$

The "substitution" obtained in the last step is $\{X/Y_2, Y_0/f(\infty, Y_2), T_2/f(\infty, Y_2)\}$ (see Section 3.2). The resulting answer substitution is $\{X/Y_2\}$. In other words the conclusion is that every term is crazy, which is not true in the intended model. Thus it is not a logical consequence of the program which shows that the inference is no longer sound. ∎

3.5 Completeness of SLD-resolution

Another important problem is whether all correct answers for a given goal (i.e. all logical consequences) can be obtained by SLD-resolution. The answer is given by the following theorem, called the completeness theorem for SLD-resolution (due to Clark (1979)).

Theorem 3.22 (Completeness of SLD-resolution) Let P be a definite program, $\leftarrow A_1, \ldots, A_n$ a definite goal and $\Re$ a computation rule. If $P \models \forall(A_1 \wedge \cdots \wedge A_n)\sigma$, there exists a refutation of $\leftarrow A_1, \ldots, A_n$ via $\Re$ with the computed answer substitution θ such that $(A_1 \wedge \cdots \wedge A_n)\sigma$ is an instance of $(A_1 \wedge \cdots \wedge A_n)\theta$. ∎

The proof of the theorem is not very difficult but is rather long and requires some auxiliary notions and lemmas. It is therefore omitted. The interested reader is referred to e.g. Apt (1990), Lloyd (1987), Stärk (1990) or Doets (1994).

Theorem 3.22 shows that even if all correct answers cannot be computed using SLD-resolution, every correct answer is an instance of some computed answer. This is

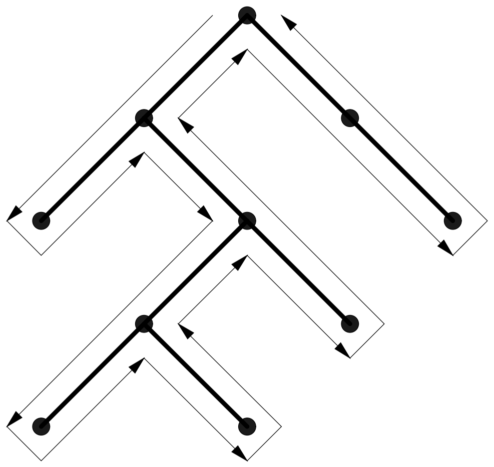


Figure 3.6: Depth-first search with backtracking

due to the fact that only most general unifiers — not arbitrary unifiers — are computed in derivations. However every particular correct answer is a special instance of some computed answer since all unifiers can always be obtained by further instantiation of a most general unifier.

Example 3.23 Consider the goal clause $\leftarrow p(X)$ and the following program:

$$p(f(Y)).$$
$$q(a).$$

Clearly, $\{X/f(a)\}$ is a correct answer to the goal — that is:

$$\{p(f(Y)), q(a)\} \models p(f(a))$$

However, the only computed answer substitution (up to renaming) is $\{X/f(Y_0)\}$. Clearly, this is a more general answer than $\{X/f(a)\}$. ∎

The completeness theorem confirms *existence* of a refutation which produces a more general answer than any given correct answer. However the problem of how to *find* this refutation is still open. The refutation corresponds to a complete path in the SLD-tree of the given goal and computation rule. Thus the problem reduces to a systematic search of the SLD-tree. Existing Prolog systems often exploit some ordering on the program clauses, e.g. the textual ordering in the source program. This imposes the ordering on the edges descending from a node of the SLD-tree. The tree is then traversed in a *depth-first* manner following this ordering. For a finite SLD-tree this strategy is complete. Whenever a leaf node of the SLD-tree is reached the traversal continues by *backtracking* to the last preceding node of the path with unexplored branches (see Figure 3.6). If it is the empty goal the answer substitution of the completed refutation is reported before backtracking. However, as discussed in Section 3.3 the SLD-tree may be infinite. In this case the traversal of the tree will never

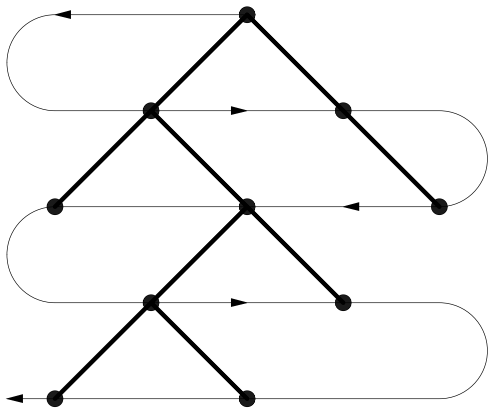


Figure 3.7: Breadth-first search

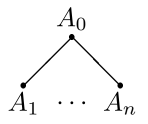
terminate and some existing answers may never be computed. This can be avoided by a different strategy of tree traversal, like for example the *breadth-first* strategy illustrated in Figure 3.7. However this creates technical difficulties in implementation due to very complicated memory management being needed in the general case. Because of this, the majority of Prolog systems use the depth-first strategy for traversal of the SLD-tree.

3.6 Proof Trees

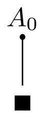
The notion of SLD-derivation resembles the notion of derivation used in formal grammars (see Chapter 10). By analogy to grammars a derivation can be mapped into a graph called a *derivation tree*. Such a tree is constructed by combining together elementary trees representing renamed program clauses. A definite clause of the form:

$$A_0 \leftarrow A_1, \ldots, A_n \qquad (n \geq 0)$$

is said to have an *elementary tree* of one of the forms:



if $n > 0$



if $n = 0$

Elementary trees from a definite program P may be combined into *derivation trees* by combining the root of a (renamed) elementary tree labelled by $p(s_1, \ldots, s_n)$ with the leaf of another (renamed) elementary tree labelled by $p(t_1, \ldots, t_n)$. The joint node is labelled by an equation $p(t_1, \ldots, t_n) \doteq p(s_1, \ldots, s_n)$.[2] A derivation tree is said to be *complete* if it is a tree and all of its leaves are labelled by ■. Complete derivation trees are also called *proof trees.* Figure 3.8 depicts a proof tree built out of the following elementary trees from the program in Example 3.14:

[2]Strictly speaking equations may involve terms only. Thus, the notation $p(t_1, \ldots, t_n) \doteq p(s_1, \ldots, s_n)$ should be viewed as a shorthand for $t_1 \doteq s_1, \ldots, t_n \doteq s_n$.

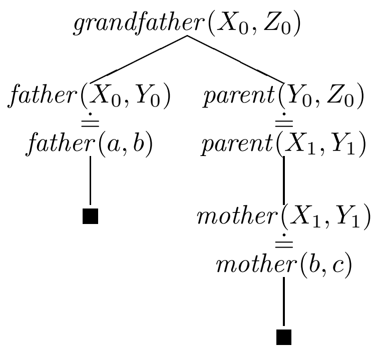
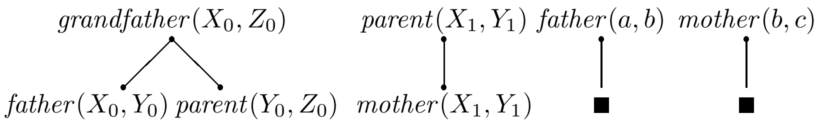


Figure 3.8: Consistent proof tree



A derivation tree or a proof tree can actually be viewed as a collection of equations. In the particular example above:

$$\{X_0 \doteq a, Y_0 \doteq b, Y_0 \doteq X_1, Z_0 \doteq Y_1, X_1 \doteq b, Y_1 \doteq c\}$$

In this example the equations can be transformed into solved form:

$$\{X_0 \doteq a, Y_0 \doteq b, X_1 \doteq b, Z_0 \doteq c, Y_1 \doteq c\}$$

A derivation tree or proof tree whose set of equations has a solution (i.e. can be transformed into a solved form) is said to be *consistent*. Note that the solved form may be obtained in many different ways. The solved form algorithm is not specific as to what equation to select from a set — any selection order yields an equivalent solved form.

Not all derivation trees are consistent. For instance, the proof tree in Figure 3.9 does not contain a consistent collection of equations since the set:

$$\{X_0 \doteq a, Y_0 \doteq b, Y_0 \doteq X_1, Z_0 \doteq c, X_1 \doteq a, Y_1 \doteq b\}$$

does not have a solved form.

The idea of derivation trees may easily be extended to incorporate also atomic *goals*. An atomic goal $\leftarrow A$ may be seen as an elementary tree with a single node, labelled by A, which can only be combined with the root of other elementary trees. For instance, proof tree (a) in Figure 3.10 is a proof tree involving the goal $\leftarrow grandfather(X, Y)$. Note that the solved form of the associated set of equations provides an answer to the initial goal — for instance, the solved form:

$$\{X \doteq a, Y \doteq c, X_0 \doteq a, Y_0 \doteq b, Y_0 \doteq X_1, Z_0 \doteq Y_1, X_1 \doteq b, Y_1 \doteq c\}$$

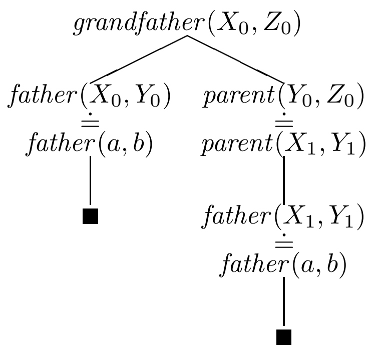


Figure 3.9: Inconsistent proof tree

of the equations associated with proof tree (a) in Figure 3.10 provides an answer substitution $\{X/a, Y/c\}$ to the initial goal.

The solved form of the equations in a consistent derivation tree can be used to simplify the derivation tree by instantiating the labels of the tree. For instance, applying the substitution $\{X/a, Y/c, X_0/a, Y_0/b, Z_0/c, X_1/b, Y_1/c\}$ (corresponding to the solved form above) to the nodes in the proof tree yields a new proof tree (depicted in Figure 3.11). However, nodes labelled by equations of the form $A \doteq A$ will usually be abbreviated A so that the tree in Figure 3.11 is instead written as the tree (d) in Figure 3.10. The equations of the simplified tree are clearly consistent.

Thus the search for a consistent proof tree can be seen as two interleaving processes: The process of combining elementary trees and the simplification process working on the equations of the already constructed part of the derivation tree. Note in particular that it is not necessary to simplify the whole tree at once — the tree (a) has the following associated equations:

$$\{X \doteq X_0, Y \doteq Z_0, X_0 \doteq a, Y_0 \doteq b, \underline{Y_0 \doteq X_1}, \underline{Z_0 \doteq Y_1}, X_1 \doteq b, Y_1 \doteq c\}$$

Instead of solving all equations only the underlined equations may be solved, resulting in an mgu $\theta_1 = \{Y_0/X_1, Z_0/Y_1\}$. This may be applied to the tree (a) yielding the tree (b). The associated equations of the new tree can be obtained by applying θ_1 to the previous set of equations after having removed the previously solved equations:

$$\{X \doteq X_0, Y \doteq Y_1, X_0 \doteq a, X_1 \doteq b, \underline{X_1 \doteq b}, \underline{Y_1 \doteq c}\}$$

Solving of the new underlined equations yields a mgu $\theta_2 = \{X_1/b, Y_1/c\}$ resulting in the tree (c) and a new set of equations:

$$\{\underline{X \doteq X_0}, \underline{Y \doteq c}, \underline{X_0 \doteq a}, \underline{b \doteq b}\}$$

Solving all of the remaining equations yields $\theta_3 = \{X/a, Y/c, X_0/a\}$ and the final tree (d) which is trivially consistent.

Notice that we have not mentioned *how* proof trees are to be constructed or in which order the equations are to be solved or checked for consistency. In fact, a whole spectrum of strategies is possibile. One extreme is to first build a complete proof

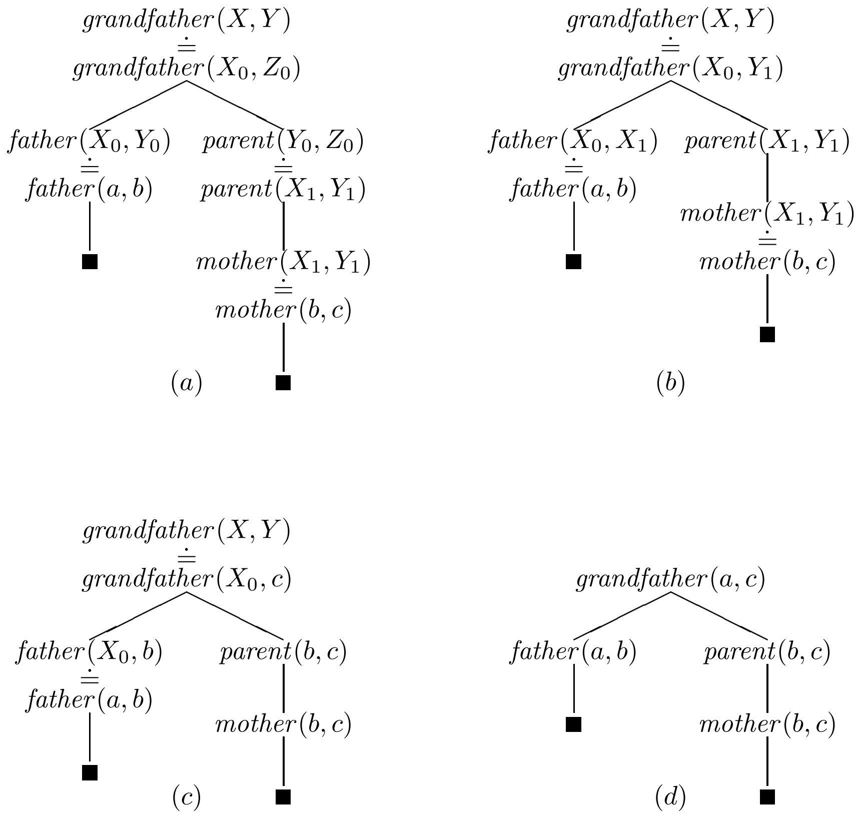


Figure 3.10: Simplification of proof tree

tree and then check if the equations are consistent. At the other end of the spectrum equations may be checked for consistency while building the tree. In this case there are two possibilities — either the whole set of equations is checked every time a new equation is added or the tree is simplified by trying to solve equations as soon as they are generated. The latter is the approach used in Prolog — the tree is built in a depth-first manner from left to right and each time a new equation is generated the tree is simplified.

From the discussion above it should be clear that many derivations may map into the same proof tree. This is in fact closely related to the intuition behind the independence of the computation rule — take "copies" of the clauses to be combined together. Rename each copy so that it shares no variables with the other copies. The clauses are then combined into a proof tree. A computation rule determines the order in which the equations are to be solved but the solution obtained is independent of this order (up to renaming of variables).

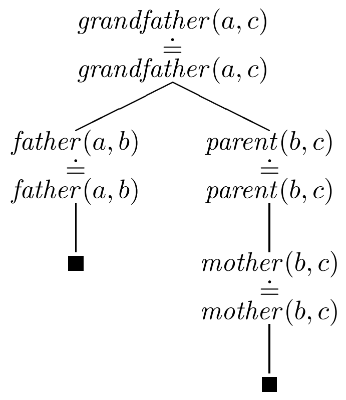


Figure 3.11: Resolved proof tree

Exercises

3.1 What are the mgu's of the following pairs of atoms:

$p(X, f(X))$	$p(Y, f(a))$
$p(f(X), Y, g(Y))$	$p(Y, f(a), g(a))$
$p(X, Y, X)$	$p(f(Y), a, f(Z))$
$p(a, X)$	$p(X, f(X))$

3.2 Let θ be an mgu of s and t and ω a renaming substitution. Show that $\theta\omega$ is an mgu of s and t.

3.3 Let θ and σ be substitutions. Show that if $\theta \preceq \sigma$ and $\sigma \preceq \theta$ then there exists a renaming substitution ω such that $\sigma = \theta\omega$.

3.4 Let θ be an idempotent mgu of s and t. Prove that σ is a unifier of s and t iff $\sigma = \theta\sigma$.

3.5 Consider the following definite program:

$$\begin{array}{l} p(Y) \leftarrow q(X, Y), r(Y). \\ p(X) \leftarrow q(X, X). \\ q(X, X) \leftarrow s(X). \\ r(b). \\ s(a). \\ s(b). \end{array}$$

Draw the SLD-tree of the goal $\leftarrow p(X)$ if Prolog's computation rule is used. What are the computed answer substitutions?

3.6 Give an example of a definite program, a goal clause and two computation rules where one computation rule leads to a finite SLD-tree and where the other computation rule leads to an infinite tree.

3.7 How many consistent proof trees does the goal $\leftarrow p(a, X)$ have given the program:

$$\begin{array}{l} p(X, Y) \leftarrow q(X, Y). \\ p(X, Y) \leftarrow q(X, Z), p(Z, Y). \\ q(a, b). \\ q(b, a). \end{array}$$

3.8 Let θ be a renaming substitution. Show that there is only one substitution σ such that $\sigma\theta = \theta\sigma = \epsilon$.

3.9 Show that if $A \in B_P$ and $\leftarrow A$ has a refutation of length n then $A \in T_P \uparrow n$.