2. Foundations of Loop Checking

In this chapter we systematically study the foundations of loop checking mechanisms. To this end, we provide in Section 2.1 a general definition of a loop check. We also introduce a natural subclass of loop checks, called simple loop checks: their definition does not depend on the analyzed logic program.

In Section 2.2 we define some important properties of loop checks, like soundness (no computed answer to a goal is missed) and completeness (all resulting derivations are finite). We study the effect of adding loop checks to top-down interpreters. Finally we prove that no sound and complete simple loop check exists even in the absence of function symbols.

In Section 2.3 we study some nonsimple loop checks: loop checks that take the program into account. We show that such a loop check can be sound and complete for the class of function-free programs. However, their value for practical purposes appears to be limited: nonsimple loop check are in a sense too powerful.

2.1. What is a Loop Check?

Definitions

One might define a loop check as a function from SLD-trees to unfinished SLD-trees. However, this would be a very general definition, allowing practically everything. The purpose of a loop check is to prune an SLD-tree to an initial subtree of it. Moreover, we shall use here a more restricted definition: given a program P and a goal G, the decision to prune a node is based only upon its ancestors in the SLD-tree of PÆ{G}, that is on the SLD-derivation from G up to this node.

Thus we exclude here more complicated pruning mechanisms, for which the decision whether a node in a tree is pruned depends on the so far traversed fragment of the considered tree. Such mechanisms are for example studied by Vieille [V] and Tamaki & Sato [TS] (see Chapter 8).

Due to this restriction we could define a loop check as a function which, given a program and an SLD-derivation, returns it unchanged if it is not pruned, and otherwise returns the proper initial subderivation of it that ends in the pruned
node. Of course, if a derivation $D$ is pruned at the goal $G$, then every derivation $D'$ that is the same as $D$ until and including $G$ must also be pruned at $G$: the ancestors of $G$ are the same in $D$ and $D'$.

This means that it is better to define a loop check as a set of derivations (depending on the program): the derivations that are pruned exactly at their last node. Thus a program $P$ and a loop check $L$ determine a set of (unfinished) SLD-derivations $L(P)$. Such a loop check $L$ can be extended in a canonical way to a function $f_L$ from SLD-trees to unfinished SLD-trees by pruning in an SLD-tree $T$ for $P \cup \{G_0\}$ the nodes in $\{G \mid$ the SLD-derivation from $G_0$ to $G$ in $T$ is in $L(P)\}$. We shall usually make this conversion implicitly.

We shall mainly study an even more restricted form of a loop check, called simple loop check, in which the set of pruned derivations is independent of the program. Thus a loop check is a function with a program as input and a set of derivations, being a simple loop check, as output. This leads us to the following definitions.

**Definition 2.1.1.**
Let $L$ be a set of SLD-derivations. $\text{Initials}(L) = \{D \in L \mid L$ does not contain a proper initial subderivation of $D\}$. $L$ is subderivation free if $L = \text{Initials}(L)$. □

In order to render the intuitive meaning of a loop check $L$: ‘every derivation $D \in L$ is pruned exactly at its last node’, we need that $L$ is subderivation free. Note that $\text{Initials}(\text{Initials}(L)) = \text{Initials}(L)$.

**Definition 2.1.2 (Simple loop check).**
A simple loop check is a computable set $L$ of finite SLD-derivations such that $L$ is closed under variants and subderivation free. □

The first condition here ensures that the choice of variables in the input clauses in an SLD-derivation does not influence its pruning. This is a reasonable demand since we are not interested in the choice of the names of these variables.

**Definition 2.1.3 (Loop check).**
A loop check is a computable function $L$ from programs to sets of SLD-derivations such that for every program $P$, $L(P)$ is a simple loop check. □
Of course, we can treat a simple loop check \( L \) as a loop check, namely as the constant function \( \lambda P. L \).

**Definition 2.1.4.**

Let \( L \) be a loop check. An SLD-derivation \( D \) of \( P \cup \{ G \} \) is pruned by \( L \) if \( L(P) \) contains \( D \) or a proper initial subderivation of \( D \).

An example: the Variant of Atom check

A first attempt to formulate the *Variant of Atom (VA)* check might be: ‘A derivation is pruned at the first goal that contains a variant \( A \) of an atom \( A' \) that occurred in an earlier goal.’ Note that we have to allow here that \( A \) and \( A' \) are variants: if we required \( A = A' \) then we would violate the first condition in Definition 2.1.2.

The intuition behind this loop check is the following. We wish to prove \( A' \) by resolution. If we find out after some resolution steps that in order to prove \( A' \) we need to prove a variant \( A \) of \( A' \), then there are two possibilities. One is that there is a proof for \( A \). Then this proof could also be used as a proof for \( A' \), by applying an appropriate renaming on it. So we do not need the proof of \( A' \) that goes via \( A \). The other possibility is that there is no proof for \( A \). In that case, the attempt to prove \( A' \) via \( A \) cannot be successful. So in both cases there is no reason to continue the attempt to prove \( A' \) via \( A \).

The derivation step \( \leftarrow B,A \Rightarrow_{B \leftarrow \leftarrow} \leftarrow A \) shows that the first formulation of the VA check is not precise enough: it does not capture the intuition that the proof of \( A' \) goes via \( A \). The atom \( A \) should be the result (after one or more derivation steps) of resolving \( A' \), or a further instantiated version of \( A' \) (if \( A' \) is not immediately selected). Therefore we arrive at the following definition.

**Definition 2.1.5 (Variant of Atom check).**

The *Variant of Atom check* is the set of SLD-derivations

\[
VA = \text{Initials}(\{ D \mid D = (G_0 \Rightarrow C_{i,0} G_1 \Rightarrow \ldots \Rightarrow G_{k-1} \Rightarrow C_{k,0} G_k) \text{ such that for some } i \text{ and } j, 0 \leq i \leq j < k, G_k \text{ contains an atom } A \text{ that is}
\]

- a variant of an atom \( A' \) in \( G_i \) and
- the result of an attempt to resolve \( A' \theta_{i+1} \ldots \theta_j \), the further instantiated version of \( A' \), that is selected in \( G_j \)).

\[\square\]
We now illustrate the use of this loop check.

**Example 2.1.6.**
(This example is based on Example 8 in [B], see also [vG1]).

Let \( P = \{ \quad p(0) \leftarrow. \quad (C1), \quad q(1) \leftarrow. \quad (C2), \quad p(x) \leftarrow p(y). \quad (C3), \quad r \leftarrow p(x).q(x). \quad (C4) \quad \} \),

let \( G = \leftarrow r. \)

That the informal justification of the loop check VA is incorrect, is shown by applying it to two SLD-trees of \( P \cup \{ G \} \), via the leftmost and rightmost selection rule respectively, which gives us Figure 2.1.1. (In this figure and elsewhere a failed node, i.e., a node without a successor in the SLD-tree, is marked by a box around it. \( C' \) denotes the program clause \( C \), where every variable \( v \) is renamed to \( v' \).)

![Figure 2.1.1](image)

A detailed analysis shows why the goal \( G_3 = \leftarrow p(y') \) in the rightmost tree is pruned by the VA check. Clearly, a variant of \( p(y') \) occurs in an earlier goal: \( p(x) \) in \( G_1 \). So we take \( i = 1 \). In \( G_1 \), \( p(x) \) is not yet selected, so \( j > i \). In fact
Indeed, this loop check has not worked properly here: all successful derivations have been pruned. Clearly, this is an undesirable property for loop checks. On the other hand, all infinite derivations are pruned, as intended. In the next section, we shall give formal definitions of these and related properties of loop checks.

2.2. Properties of Loop Checks

In this section some basic properties of loop checks are introduced and some natural results concerning them are established.

**Soundness and completeness**

The most important property is that using a loop check does not result in a loss of success: the answer to the query \( \exists G^\sim \) (which is simply ‘yes’ or ‘no’) must not change. Since we intend to use pruned trees instead of the original ones, we need at least that pruning a successful tree yields again a successful tree.

Even stronger, often we do not want to lose any individual solution. That is, if the original tree contains a successful branch, giving some computed answer \( \theta \) (thus proving \( \forall G^\sim \theta \)), then we require that the pruned tree contains a successful branch giving a more general answer than \( \theta \), thus proving (a formula trivially implying) \( \forall G^\sim \theta \). In this way every correct answer is still ‘represented’ by a more general computed answer in the pruned tree, thus ensuring the completeness of SLD-resolution with loop checking.

Finally, we would like to retain only shorter derivations and prune the longer ones that give the same result. This leads to the following definitions.

**Definition 2.2.1 (Soundness).**

i) A loop check \( L \) is **weakly sound** if for every program \( P \), goal \( G \), and SLD-tree \( T \) of \( P \cup \{ G \} \): if \( T \) is successful, then \( f_L(T) \) is successful.
ii) A loop check \( L \) is \textit{sound} if for every program \( P \), goal \( G \), and SLD-tree \( T \) of \( P \cup \{G\} \): if \( T \) contains a successful branch with a computed answer \( G^\sigma \), then \( f_L(T) \) contains a successful branch with a computed answer \( G^\sigma' \leq G^\sigma \).

iii) A loop check \( L \) is \textit{shortening} if for every program \( P \), goal \( G \), and SLD-tree \( T \) of \( P \cup \{G\} \): if \( T \) contains a successful branch \( D \) with a computed answer \( G^\sigma \), then either \( f_L(T) \) contains \( D \) or \( f_L(T) \) contains a successful branch \( D' \) with a computed answer \( G^\sigma' \leq G^\sigma \) such that \( |D'| < |D| \).

The following lemma is an immediate consequence of these definitions.

**Lemma 2.2.2.** Let \( L \) be a loop check.

1. If \( L \) is shortening, then \( L \) is sound.
2. If \( L \) is sound, then \( L \) is weakly sound.

The purpose of a loop check is to reduce the search space for top-down interpreters. Although impossible in general, we would like to end up with a finite search space. This is the case if every infinite derivation is pruned.

**Definition 2.2.3 (Completeness).**

A loop check \( L \) is \textit{complete w.r.t. a selection rule} \( R \) for a class of programs \( \mathcal{C} \), if for every program \( P \in \mathcal{C} \) and goal \( G \) in \( L_P \), every infinite SLD-derivation of \( P \cup \{G\} \) via \( R \) is pruned by \( L \).

We must point out here that by these definitions we have overloaded the terms ‘soundness’ and ‘completeness’. These terms do not only refer to loop checks, but also to interpreters for logic programs (with or without a loop check). As explained in Section 1.3, such an interpreter is sound if any answer it gives is correct w.r.t. the intended model or the intended theory of the program. An interpreter is complete if it finds every correct answer within a finite time.

**Interpreters and loop checks**

When a top-down interpreter is augmented with a loop check, we obtain a new interpreter. The soundness and completeness of this new interpreter depends on the soundness and completeness of the old one, as well as on the soundness and completeness of the loop check. However, these relations are not trivial. For
example, it is not true that adding a complete loop check to a complete interpreter yields a complete interpreter (recall that the notion of soundness of a loop check was introduced to ensure the completeness of the interpreter equipped with it).

The relationships between soundness and completeness of loop checks and the interpreters augmented with them are expressed in the following lemmas. We refer here to two interpreters: one searching the SLD-tree depth-first left-to-right (as the PROLOG interpreter does), and one searching breadth-first. Recall that without a loop check, both interpreters are sound and sound w.r.t. CWA. The breadth-first interpreter is also complete, but not complete w.r.t. CWA.

**Lemma 2.2.4.** Let $P$ be a program, $A$ a ground atom and $L$ a weakly sound loop check. Then for every SLD-tree $T$ of $P \cup \{\leftarrow A\}, \neg A \in \text{CWA}(P)$ iff $f_L(T)$ contains no successful branches.

**Proof.** We know by the Soundness and strong completeness Theorem 1.3.2 that $\neg A \in \text{CWA}(P) \iff T$ contains no successful branches.

$\Rightarrow$ $T$ contains no successful branches and $f_L(T)$ is a subtree of $T$, so $f_L(T)$ contains no successful branches either.

$\Leftarrow$ Since $L$ is weakly sound, a successful branch in $T$ would yield a successful branch in $f_L(T)$. But $f_L(T)$ contains no successful branches, hence $T$ contains no successful branches either.

Thus an interpreter augmented with a weakly sound loop check remains sound w.r.t. CWA. Since $f_L(T)$ may be infinite, nothing can be said about completeness.

**Lemma 2.2.5.** Let $P$ be a program, $G$ a goal and $T$ an SLD-tree of $P \cup \{G\}$. Let $L$ be a sound loop check. Then $G^\sim \theta$ is a correct answer for $P \cup \{G\}$ iff $f_L(T)$ contains a successful branch with a computed answer $G^\sim \tau \leq G^\sim \theta$.

**Proof.** We have by the strong completeness of SLD-resolution $P \models G^\sim \theta \iff T$ contains a successful branch with a computed answer $G^\sim \sigma \leq G^\sim \theta$.

$\Rightarrow$ $T$ contains this successful branch, and since $L$ is sound, $f_L(T)$ contains a successful branch with a computed answer substitution $\tau$ such that $G\tau \leq G\sigma$.

Now $G^\sim \tau \leq G^\sim \sigma \leq G^\sim \theta$.

$\Leftarrow$ $f_L(T)$ contains a successful branch with a computed answer $G^\sim \tau \leq G^\sim \theta$, so $T$ contains this branch as well.
Thus an interpreter augmented with a sound loop check remains sound. Moreover, a breadth-first interpreter remains complete.

**COROLLARY 2.2.6.** Let $P$ be a program, $A$ a ground atom and $L$ a weakly sound and complete loop check. Then for every SLD-tree $T$ of $P \cup \{ \leftarrow A \}$, $\neg A \in CWA(P)$ iff $f_L(T)$ is finite and contains no successful branches.

**PROOF.** By Lemma 2.2.4 and the Completeness Definition 2.2.3. □

Thus an interpreter augmented with a weakly sound and complete loop check becomes complete w.r.t. CWA.

**COROLLARY 2.2.7.** Let $P$ be a program, $G$ a goal and $L$ a sound and complete loop check. Then for every correct answer $G \sim \emptyset$ for $P \cup \{ G \}$ and for every SLD-tree $T$ of $P \cup \{ G \}$, $f_L(T)$ is finite and contains a successful branch with a computed answer $G \sim \tau \leq G \sim \emptyset$.

**PROOF.** By Lemma 2.2.5 and the Completeness Definition 2.2.3. □

Thus a depth-first interpreter augmented with a sound and complete loop check becomes complete. This also means that a sound and complete loop check can be used to implement query processing as defined in the Introduction. Indeed, given a program $P$ and an atom $A$ with an SLD-tree $T$ of $P \cup \{ \leftarrow A \}$, it suffices to traverse the finite tree $f_L(T)$ and to collect all (computed) answers.

*Comparing and combining loop checks*

After studying the relationships between loop checks and interpreters, we shall now analyze a relationship between loop checks. In general, it can be quite difficult to compare loop checks. However, some of them can be compared in a natural way: if every loop that is detected by one loop check, is detected at the same derivation step or earlier by another loop check, then the latter one is stronger than the former.

**DEFINITION 2.2.8.**

Let $L_1$ and $L_2$ be loop checks. $L_1$ is stronger than $L_2$ if for every program $P$ and goal $G$, every SLD-derivation $D_2 \in L_2(P)$ of $P \cup \{ G \}$ that is not itself contained in $L_1(P)$ has a proper initial subderivation $D_1 \in L_1(P)$. □
In other words, $L_1$ is stronger than $L_2$ if every SLD-derivation that is pruned by $L_2$ is also pruned by $L_1$. Note that the definition implies that every loop check is stronger than itself.

When an interpreter is augmented with a loop check, we obtain a new interpreter. This means that we could iterate the process, adding several different loop checks in order to detect more loops or to detect loops earlier. Another way to obtain this result is to combine these loop checks into one new loop check, which is added to the interpreter. This leads to the following definition.

**Definition 2.2.9 (Sum of loop checks).**

Let $L_1$ and $L_2$ be loop checks. For every program $P$, the union of $L_1$ and $L_2$ (denoted by $L_1+L_2$) is defined as: $(L_1+L_2)(P) = \text{Initials}(L_1(P) \cup L_2(P))$.

Note that we can not take simply $L_1(P) \cup L_2(P)$, since one loop check might contain a proper initial subderivation of the other. A number of nice, easily provable properties hold for sums of loop checks.

**Theorem 2.2.10.** Let $L_1$, $L_2$ and $L_3$ be loop checks. Then:

i) $L_1+L_2$ is a loop check.

ii) $L_1+L_1 = L_1$.

iii) $L_1+L_2 = L_2+L_1$.

iv) $(L_1+L_2)+L_3 = L_1+(L_2+L_3)$.

v) $L_1$ is stronger than $L_2$ iff $L_1+L_2 = L_1$.

vi) If $L_1$ and $L_2$ are simple, then $L_1+L_2$ is simple.

vii) If $L_1$ and $L_2$ are shortening, then $L_1+L_2$ is shortening.

**Proof.** i)-vi). Straightforward.

vii). For every successful derivation $D$ with computed answer $G^\sim \sigma$, the shortest derivation(s) with a computed answer more general than $G^\sim \sigma$ is (are) neither pruned by $L_1$ nor by $L_2$, hence it is (they are) not pruned by $L_1+L_2$.  

**Remark 2.2.11.** Even if $L_1$ and $L_2$ are sound, $L_1+L_2$ can be unsound. The following example shows that this is still true if $L_1$ and $L_2$ are both simple.
Let \( P = \{ p(x,1) \leftarrow p(x,0). \quad (C1), \quad p(1,x) \leftarrow p(0,x). \quad (C2), \quad p(0,0) \leftarrow. \quad (C3) \} \)

Consider the SLD-tree of \( P \cup \{ \leftarrow p(1,1) \} \) in Figure 2.2.1.

Let \( L_1 \) be the set of variants of \( D_1 = (\leftarrow p(1,1) \Rightarrow (C1) \leftarrow p(1,0) \Rightarrow (C2) \leftarrow p(0,0) ) \) and let \( L_2 \) be the set of variants of \( D_2 = (\leftarrow p(1,1) \Rightarrow (C2) \leftarrow p(0,1) \Rightarrow (C1) \leftarrow p(0,0) ) \).

Both \( L_1 \) and \( L_2 \) are sound: every SLD-tree that contains (a variant of) \( D_1 \) must contain (a variant of) \( D_2 \) and vice versa. Clearly \( L_1 + L_2 \) is unsound.

The following theorem will prove to be very useful. It will enable us to obtain soundness and completeness results for loop checks which are related by the ‘stronger than’ relation, by proving soundness and completeness for only one of them.

**THEOREM 2.2.12 (Relative Strength).** Let \( L_1 \) and \( L_2 \) be loop checks, and let \( L_1 \) be stronger than \( L_2 \).

i) If \( L_1 \) is weakly sound, then \( L_2 \) is weakly sound.

ii) If \( L_1 \) is sound, then \( L_2 \) is sound.

iii) If \( L_1 \) is shortening, then \( L_2 \) is shortening.

iv) If \( L_2 \) is complete (w.r.t. a selection rule \( R \) for a class of programs \( \varphi \)), then \( L_1 \) is complete (w.r.t. \( R \) for the class of programs \( \varphi \)).

**PROOF.**

i)-iii) If an SLD-tree \( T \) contains a successful branch, then \( f_{L_1}(T) \) contains a successful branch that satisfies the conditions of Definition 2.2.8. Since \( L_1 \) is stronger than \( L_2 \), \( f_{L_1}(T) \) is a subtree of \( f_{L_2}(T) \), so this branch is also contained in \( f_{L_2}(T) \).

iv) Every infinite SLD-derivation is pruned by \( L_2 \), so it is also pruned by \( L_1 \).
some infinite derivations (in other words, it might be incomplete). Of course, the ‘stronger than’ relation is not linear. Moreover, loop checks exist that are neither sound nor complete.

The existence of sound and complete loop checks
A question now arises: do there exist sound and complete loop checks? Obviously, there cannot be such a loop check for logic programs in general, as logic programming has the full power of recursion theory. (Remember that according to the definition, a loop check is computable.) So when studying completeness we shall rule out programs that compute over an infinite domain. We do so by restricting our attention to programs without function symbols, so called function-free programs. This restriction leads to a finite Herbrand Universe, but other solutions (typed functions, bounded term-size property [vG2]) are also possible here.

Now our question can be reformulated as: is there a sound and complete loop check for function-free programs? Before answering this question for loop checks in general, we answer it for simple loop checks.

Theorem 2.2.13. There is no weakly sound and complete simple loop check for function-free programs.

Proof. The proof is similar to the proof of Theorem 4.7 in [BW] for sound loop checks. Let L be a simple loop check that is complete for function-free programs. Consider the following infinite SLD-derivation D, obtained by repeatedly using the clause \( p(x) \leftarrow p(y), s(y, x) \) (using the leftmost selection rule).

\[
\begin{align*}
\leftarrow & p(x_0), q(x_0) \\
\Downarrow \\
\leftarrow & p(x_1), s(x_1, x_0), q(x_0) \\
\Downarrow \\
\leftarrow & p(x_2), s(x_2, x_1), s(x_1, x_0), q(x_0) \\
\Downarrow \\
\leftarrow & p(x_3), s(x_3, x_2), s(x_2, x_1), s(x_1, x_0), q(x_0) \\
\Downarrow \\
\quad & \ldots
\end{align*}
\]

Figure 2.2.2
Since L is a complete loop check, this derivation is pruned by L and since L is simple, the goal at which pruning takes place is independent of the program used for this derivation. Suppose this derivation is pruned by L at the goal $\leftarrow p(x_n), s(x_n,x_{n-1}), \ldots, s(x_1,x_0), q(x_0)$.

Now let $P = \{ s(i,i+1) \leftarrow \ | 0 \leq i < n \} \cup \{ p(0) \leftarrow, p(x) \leftarrow p(y), s(y,x), q(n) \leftarrow \}$. Extending the above derivation to an SLD-tree of $P \cup \{ G \}$ (still using the leftmost selection rule, see Figure 2.2.3), we see that every goal of the derivation has two descendants, obtained by applying the clauses $p(0) \leftarrow$ and $p(x) \leftarrow p(y), s(y,x)$ respectively. The derivation of Figure 2.2.2 shows the effect of repeatedly applying $p(x) \leftarrow p(y), s(y,x)$. After applying $p(0) \leftarrow$ at some goal, a derivation becomes deterministic: if there are initially $m$ s-atoms, then these atoms are resolved from left to right by the clauses $s(0,1), \ldots, s(m-1,m) \leftarrow$.

Finally, the goal $\leftarrow q(m)$ is left. Since of all goals of the form $\leftarrow q(i)$ ($i \geq 0$) only the goal $\leftarrow q(n)$ can be refuted, exactly $n$ s-atoms are needed. Therefore the only successful branch of this SLD-tree of $P \cup \{ G \}$ goes via the goal $\leftarrow p(x_n), s(x_n,x_{n-1}), \ldots, s(x_1,x_0), q(x_0)$. As exactly this goal is pruned by L, L has pruned the only successful branch of this SLD-tree. Hence L is not weakly sound.

$\leftarrow p(x_0), q(x_0) \quad \Rightarrow \quad \leftarrow q(0)$

$\downarrow$

$\leftarrow p(x_1), s(x_1,x_0), q(x_0) \quad \Rightarrow \quad \leftarrow s(0,x_0), q(x_0) \quad \Rightarrow \quad \leftarrow q(1)$

$\downarrow$

$\leftarrow p(x_2), s(x_2,x_1), s(x_1,x_0), q(x_0) \quad \Rightarrow \quad \leftarrow s(0,x_1), s(x_1,x_0), q(x_0)$

$\downarrow \quad \Rightarrow \quad \leftarrow s(1,x_0), q(x_0) \quad \Rightarrow \quad \leftarrow q(2)$

$\downarrow$

$\vdots$

$\leftarrow p(x_n), s(x_n,x_{n-1}), \ldots, s(x_1,x_0), q(x_0) \Rightarrow \ldots$ n intermediate goals $\ldots \Rightarrow \leftarrow q(n)$

$\downarrow$

$\vdots$

**FIGURE 2.2.3**
2.3. Nonsimple loop checks

In this section we investigate two nonsimple loop checks. We show that these loop checks are sound and complete for programs with a finite number of ground atoms in their language. To enforce this restriction in a simple way, we assume throughout this section that programs are function-free.

Proof Tree Redundancy

The example of Theorem 2.2.13 suggests that a sound and complete (but not simple) loop check might exist depending only on the language of the program. We shall prove that such a loop check indeed exists. Given a derivation, the loop check first constructs the associated proof tree. It prunes the derivation if this proof tree contains ‘too much’ repetition, where ‘too much’ depends on the initial goal and the language of the program. Our definition of a proof tree is an adapted version of the one in [Cl1].

**Definition 2.3.1 (Proof tree).**

Let \( D = (G_0 \Rightarrow_{C_1, \theta_1} G_1 \Rightarrow \ldots \Rightarrow G_{k-1} \Rightarrow_{C_k, \theta_k} G_k) \) be an SLD-derivation. The proof tree associated to \( D \) is constructed as follows.

First \( \theta_1 \ldots \theta_k \) is applied to every goal and clause in \( D \). This new structure is not an SLD-derivation: instances of program clauses are used and there is no standardizing apart. Also unifiers are not needed: the head of the input clause is already syntactically equal to the selected atom in the goal. So a step consists only of the replacement of this selected atom by the body of the clause. This means that we can regard these replacements as being carried out in parallel (no instantiation of shared variables). This yields the proof tree associated to \( D \). In this tree every node, consisting of an atom \( A \), has as descendants nodes consisting of the atoms \( A \) was replaced by. In the resulting proof tree, a ‘special’ root node is needed: otherwise, a goal of more than one atom would yield a forest instead of a tree.

Figure 2.3.1 shows an example of this conversion of an SLD-derivation \( D \) via \( D\theta_1 \ldots \theta_k \) (where \( \theta_1 \ldots \theta_k = \{x/x', y/1, y'/0, z'/z\} \)) into its proof tree.
A goal in $D$ corresponds to a ‘horizontal layer’ through the proof tree. A derivation step corresponds to the replacement of a node (representing the selected atom) in such a layer by its children. A simple induction argument shows that the length of an SLD-refutation equals the number of nodes in its proof tree, not counting the root.

For a program $P$ and a goal $G$ we denote by $L_{P,G}$ the language that is obtained from $L_P$ by adding the variables of $G$ to the set of constants. We can now define the intended loop check and show its effect on the derivation of Theorem 2.2.13.

**Definition 2.3.2 (Proof Tree Redundancy check).**

For a function-free program $P$, the *Proof Tree Redundancy check* is defined as $\text{PTR}(P) = \text{Initials}(\{D \mid \text{for some } G, D \text{ is an SLD-derivation of } P \cup \{G\} \text{ and for some predicate symbol } p, \text{ a branch of the proof tree associated to } D \text{ contains more } p\text{-atoms than there are ground } p\text{-atoms in } L_{P,G}\})$. □
EXAMPLE 2.3.3.
In Theorem 2.2.13 we considered the program \( P = \{ s(i,i+1) \leftarrow . \mid 0 \leq i < n \} \cup \{ p(0) \leftarrow . p(x) \leftarrow p(y), s(y,x). q(n) \leftarrow . \} \) and the resulting infinite SLD-derivation \( D \) of \( P \cup \{ \leftarrow p(x_0), q(x_0) \} \) shown in Figure 2.2.2. The proof tree associated to the first three steps of \( D \) is depicted in Figure 2.3.2.

```
root
  p(x0) q(x0)
  |
  p(x1) s(x1,x0)
  |
  p(x2) s(x2,x1)
  |
  p(x3) s(x3,x2)
```

**Figure 2.3.2**

The number of ground \( p \)-atoms in \( L_{P,G} \) is \( n+2: p(x_0), p(0), p(1), p(2), \ldots, p(n) \). The leftmost branch of a proof tree associated to a proper initial subderivation of \( D \) consists of the atoms \( p(x_0), p(x_1), p(x_2), \ldots \). Thus when the \((n+3)\)rd \( p \)-atom of this branch is generated, \( D \) is pruned, notably at the goal \( p(x_n+2), s(x_{n+2},x_{n+1}), s(x_{n+1},x_n), \ldots, s(x_1,x_0), q(x_0) \). Recall that the derivation was needed up to and including the goal \( p(x_n), s(x_n,x_{n-1}), s(x_{n-1},x_{n-2}), \ldots, s(x_1,x_0), q(x_0) \) in order to preserve the refutation of \( \leftarrow p(x_0), q(x_0) \). \( \square \)

For a convenient notation in the following proofs, we write \( \text{Succ}(P,G,\sigma) \) for the set of SLD-refutations of \( P \cup \{ G \} \) with a computed answer \( G \models \tau \leq G \models \sigma \). We say that a refutation \( D \) is a shortest refutation in \( \text{Succ}(P,G,\sigma) \) if \( D \in \text{Succ}(P,G,\sigma) \) and \( |D| = \min \{|D'| | D' \in \text{Succ}(P,G,\sigma)\} \).

**THEOREM 2.3.4.** \( \text{PTR} \) is shortening (so a fortiori sound).

**PROOF.** Let \( P \) be a program, \( G \) a goal in \( L_P \), \( \sigma \) a substitution and \( D \) a shortest derivation in \( \text{Succ}(P,G,\sigma) \). We must show that \( D \) is not pruned by \( \text{PTR} \). To this end we prove that for every predicate symbol \( p \) in \( L_P \), no branch in the proof
tree $T$ associated to $D$ contains more $p$-atoms than there are ground $p$-atoms in $L_{P,G}$.

We prove this claim by contradiction: suppose that for the predicate symbol $p$ there is such a branch in $T$. Then there exists a ground instance of $T$ (w.r.t. $L_{P,G}$) in which some node consists of the same $p$-atom as one of its ancestors. Now a proof tree with less nodes than $T$ can be constructed:

![Figure 2.3.3](image)

For any selection rule, this smaller proof tree can be converted back into an SLD-refutation with the same computed answer substitution as $D$ (because the ground instantiation of $T$ did not affect the variables of $G$). Thus $D$ is not a shortest refutation in $\text{Succ}(P,G,\sigma)$. Contradiction.

**Theorem 2.3.5.** $\text{PTR}$ is complete.

**Proof.** Let $P$ be a program, $G$ a goal and $D$ an infinite SLD-derivation of $P \cup \{G\}$. As the proof tree associated to $D$ is infinite\(^1\), but finitely branching, it follows from König’s Lemma that it has an infinite branch. $L_{P,G}$ contains only a finite number of ground atoms. Thus for at least one predicate symbol $p$, this infinite branch contains more $p$-atoms than there are ground $p$-atoms in $L_{P,G}$.

---

\(^1\) Strictly speaking we have not defined the proof tree associated to an infinite derivation (in order to avoid an infinite composition of substitutions). Here it is sufficient to consider only the predicate symbols of the atoms, forgetting the arguments and substitutions altogether.
PTR may be shortening and complete, in practice it is often useless because it allows very long derivations. For a program $P$ and an atom $A$, and an SLD-derivation $D$ of $P \cup \{ \leftarrow A \}$ that is not pruned by PTR, the maximum length of a branch in the proof tree associated to $D$ is the number of ground atoms in $L_{P, \leftarrow A}$, say $N$ (the root is not needed here). The maximum branching factor $B$ is the maximum number of atoms in the body of a clause used in $D$. A simple calculation shows that $|D|$ can be as much as $\sum_{i=0}^{N-3} B^i$. The problem is not that PTR is too cautious: even for small languages, a shortest refutation can indeed be extremely long.

**Example 2.3.6.**
In the program of Theorem 2.2.13, the length of the (longest) successful branch is only $2n + 2$. But if we take $P_n =$

\[
\begin{align*}
&\{ s(i,i+1) \leftarrow \mid 0 \leq i < n \} \cup \\
&\{ p(x_1,x_2,x_3) \leftarrow p(x_1,x_2,y),p(x_1,x_2,y),s(y,x_3). \\
&\quad \quad \quad p(x_1,x_2,0) \leftarrow p(x_1,y,n),p(x_1,y,n),s(y,x_2). \\
&\quad \quad \quad p(x_1,0,0) \leftarrow p(y,n,n),p(y,n,n),s(y,x_1). \\
&\quad \quad \quad p(0,0,0) \leftarrow. \}
\end{align*}
\]

then a successful derivation of $P_n \cup \{ \leftarrow p(n,n,n) \}$ takes $3 \cdot 2^{(n+1)3-1} - 2$ steps. This can be seen by considering the three arguments of $p$ as representing a three-digit number in base $(n+1)$. If proving $p(x,y,z)$ takes $T(xyz)$ steps, then we have $T(xyz) = 2 \cdot T(xyz-1) + 2$ and $T(0) = 1$. This yields $T(x) = 2^{x+1} + 2^x - 2 = 3 \cdot 2^x - 2$. Now take $x = nnn$ in base $(n+1)$, that is $x = (n+1)^3 - 1$. □

Because PTR takes only the language of the program into account, it will sometimes prune derivations much later than necessary. In the next section, we investigate a stronger loop check, that takes the whole program into account.

**The strongest loop check**
Taking the whole program into account gives us an opportunity to define a shortening loop check which is stronger than *every* other shortening loop check (hence it is complete). Strange as it may seem, this loop check is also impractical.

The aim of generating an SLD-tree is to find all solutions to a problem. For a function-free program, this set of solutions is finite. Once this set is known, a
finite unfinished SLD-tree can be constructed that contains only the shortest derivation(s) for every solution. The other derivations are pruned as soon as possible. This loop check is obviously as strong as possible: every derivation that is not pruned is really needed. It is also useless for practical purposes, as there is no point in generating the pruned SLD-tree when the set of solutions is already known.

**DEFINITION 2.3.7 (STRONG check).**

\[ \text{STRONG}(P) = \text{Initials} \{ \{ D = G \Rightarrow \ldots \mid \text{for no } \sigma, \ D \text{ is an initial fragment of a shortest derivation in } \text{Succ}(P,G,\sigma) \} \} \].

Note that an SLD-tree pruned by STRONG consists not only of the shortest refutation(s) of \( P \cup \{ G \} \) for any computed answer substitution \( \sigma \), but also of the derivations that follow the path of such a derivation but ‘make a wrong decision’, that is a step deviating from such a refutation. After such a step, the derivation is immediately pruned by STRONG. This effect is caused by the fact that pruning a node in a tree implies removing all descendants, so we cannot remove the descendants caused by a ‘wrong step’ while retaining the others. The following example shows the effect of pruning an SLD-tree by STRONG.

**EXAMPLE 2.3.8.**

Let \( P = \{ p(1) \leftarrow \ldots \mid (C1), \]
\[ p(y) \leftarrow q(y,z),p(z). \quad (C2), \]
\[ q(w,0) \leftarrow . \quad (C3), \]
\[ q(0,1) \leftarrow . \quad (C4) \}, \]
and let \( G = \leftarrow p(x) \).

Consider an SLD-tree of \( P \cup \{ G \} \) displayed in Figure 2.3.4. In \( \text{Succ}(P,G,\{x/1\}) \) a minimal length derivation has 2 goals, in \( \text{Succ}(P,G,\{x/0\}) \) a minimal length derivation has 4 goals and in \( \text{Succ}(P,G,\varepsilon) \) a minimal length derivation has 6 goals. These derivations are retained by STRONG in the considered SLD-tree, the others are pruned (at the horizontal lines in the figure). Among these are successful ones, but not minimal length successful ones. (The tree in Figure 2.3.4 is extended beyond the sixth level to show this effect.)
We can now prove the claims we made in the beginning of this section.
**THEOREM 2.3.9.** For function-free programs:

i) **STRONG** is a loop check.

ii) **STRONG** is shortening.

iii) **STRONG** is stronger than any shortening loop check.

iv) **STRONG** is complete.

**PROOF.** i) **STRONG** is a loop check. The nontrivial point here is to prove that for every function-free program P, STRONG(P) is computable. Can we, given a derivation D = G ⇒ ..., decide whether or not D is pruned by STRONG and if so, at which node? Indeed we can, using the following procedure.

1. Compute the set of correct answers for P ∪ \{G\} modulo renamings (e.g. bottom-up). Since P has no function symbols, this set is finite. Construct (breadth-first) an initial subtree of an SLD-tree of P ∪ \{G\} that contains (a proper subderivation of) D and for each correct answer a successful branch with a more general computed answer.

2. For each correct answer G~σ, mark the nodes of the shortest refutations in Succ(P,G,σ).

3. Prune D at the first node in the tree that is not marked. If such a node does not exist, then D is a subderivation of a minimal length refutation.

   ii) **STRONG** is shortening. If a successful derivation D of P ∪ \{G\} with computed answer substitution σ is pruned by STRONG, then it is not a shortest derivation in Succ(P,G,σ). Obviously, there is a shortest derivation D′ ∈ Succ(P,G,σ) in the SLD-tree. D′ is shorter than D and not pruned by STRONG.

   iii) **STRONG** is stronger than any shortening loop check. Let L be a loop check and let D be a derivation of P ∪ \{G\} that is pruned by L. If D is a subderivation of a shortest refutation D′, then L is not shortening. Otherwise, D is pruned by STRONG.

   iv) **STRONG** is complete. STRONG is stronger than PTR and by Theorem 2.3.5 PTR is complete. Now apply the Relative Strength Theorem 2.2.12.

So far, we have not been very successful in defining useful sound and complete loop checks. In the next chapter, we shall restrict our attention to simple loop checks. They will be shortening (or at least weakly sound), but as shown in Theorem 2.2.13 they cannot be complete (not even for function-free programs). Nevertheless, for each of these loop checks we shall introduce one or more natural classes of programs for which they are complete.