Centrum voor Wiskunde en Informatica
Centre for Mathematics and Computer Science

R.N. Bol

Towards more efficient loop checks

Computer Science/Department of Software Technology Report CS-R9026 June

---

The Centre for Mathematics and Computer Science is a research institute of the Stichting Mathematisch Centrum, which was founded on February 11, 1946, as a nonprofit institution aiming at the promotion of mathematics, computer science, and their applications. It is sponsored by the Dutch Government through the Netherlands Organization for the Advancement of Research (N.W.O.).

Towards More Efficient Loop Checks

Roland N. Bol

Centre for Mathematics and Computer Science
P.O.Box 4079, 1009 AB Amsterdam, The Netherlands.
Phone: (+31) - 20 - 592 4080. E-mail: bol@cwi.nl.

Abstract

Loop checking is a mechanism for pruning infinite SLD-derivations. Most loop checks essentially compare the goals in a derivation: a derivation is pruned if 'sufficiently similar' goals are detected. In theory a goal is usually compared with every previous goal in the derivation, but in practice such loop checks are too expensive.

Here we investigate how to alter such loop checks to obtain less expensive ones (notably such that the number of comparisons performed is linear in the number of goals generated) while retaining the soundness and completeness results of the original loop check. To this end we modify Van Gelder's [vG] 'tortoise-and-hare' technique and study in detail the number of comparisons performed by a loop check whose checkpoints are placed in accordance with the triangular numbers.

Key Words and Phrases: deductive databases, logic programming, loop checking, termination, implementation, efficiency.
1985 Mathematical Subject Classification: 68Q40, 68T15
1987 CR Categories: F.3.2, F.4.1, H.3.3, I.2.3.
Notes: This research was partly supported by Esprit BRA-project 3020 Integration. This paper will appear in the Proceedings of the North American Conference on Logic Programming 1990.

1. Introduction

Most loop checking mechanisms for logic programming proposed in the literature ([B], [BAK], I_G in [BW], [C], [vG], [KT], [PG], [SGG], [SI] and 'redundancy' in [V]) are based on comparing goals. In theory, a goal is usually compared with every previous goal in the derivation. For such loop checks, the number of comparisons performed is quadratic in the number of goals generated. An interpreter equipped with such a loop check would not be very useful in practice: the longer a derivation gets, the more time is spent on loop checking instead of generating new goals. For a practical loop check, the number of comparisons should be at most linear in the number of goals generated.

Report CS-R9026
Centre for Mathematics and Computer Science
P.O. Box 4079, 1009 AB Amsterdam, The Netherlands

We discuss in this paper two methods for adapting existing loop checks to meet this requirement. Both methods describe which carefully selected pairs of goals are to be compared, using the comparison criterion of the original loop check. For the new loop checks thus obtained we investigate the soundness and completeness (as defined in [BAK], see also section 2.2) and the number of comparisons performed (relative to the number of goals generated).

The first method, originally proposed by Van Gelder [vG], is called the 'tortoise-and-hare technique'. It is discussed in section 3. Roughly speaking, this method compares every newly generated goal in a derivation with only one previous goal, namely the goal that is currently 'halfway' in the derivation. In this way the number of comparisons performed is equal to the number of goals generated. Unfortunately, a loop check thus obtained is generally incomplete.

Then two other closely related techniques are introduced. In both methods an (infinite) number of 'checkpoints' is selected; then every goal that is at such a checkpoint (on account of its level in the SLD-derivation or -tree) is compared with

- every previous goal ('single selected' loop checks), or
- the previous goals at checkpoints ('double selected' loop checks).

The use of single selected loop checks is already suggested in [C]. In section 4 the soundness and (for most cases) completeness of selected loop checks are proved, independently of the selection.

The 'density' of the selection determines the efficiency of a selected loop check. (The original loop check can be described as the selected loop check for which every goal is selected as a checkpoint, which is the most dense selection possible.) A 'linear' loop check is obtained if the increasing number of comparisons at the checkpoints is compensated by a decreasing density of the occurrence of checkpoints among other goals: the further the derivation is developed, the more comparisons are performed at a checkpoint, but the less checkpoints occur.

In [C], Covington argues informally that a single selected loop check with a selection of the form $\{n,n^2,n^3,\ldots\}$ (for some constant $n > 1$) is linear. In section 5 we prove in detail that for a double selected loop check this effect is obtained with the selection $\{\frac{1}{2}i(i+1) \mid i \in \mathbb{N}\}$ (the initial goal is defined to be at level 0). So for single *and* double selected loop checks, an appropriate selection renders the number of comparisons performed linear in the number of goals generated.

2. Preliminaries

In this section we recall the basic notions concerning loop checking, as presented in [BAK]. Throughout this paper we assume familiarity with the concepts and notations of logic programming as described in [L]. For two substitutions σ and τ, we write $\sigma \leq \tau$ when σ is more general than τ and for two expressions E and F, we write $E \leq F$ if F is an instance of E. An SLD-derivation step from a goal G, using a clause C and an mgu θ, to a goal H is denoted as $G \Rightarrow_{C,\theta} H$. By an SLD-derivation we mean an SLD-derivation in the sense of [L] *or an initial segment (subderivation) of it.*

2.1 Loop checks

The purpose of a loop check is to prune every infinite SLD-tree to a finite subtree of it containing the root. We define a loop check as a set of SLD-derivations: the derivations that are pruned exactly at their last node. Such a set of SLD-derivations L can be extended in a canonical way to a function f_L from SLD-trees to SLD-trees by pruning in an SLD-tree the nodes in { G | the SLD-derivation from the root to G is in L }. We shall usually make this conversion implicitly.

DEFINITION 2.1.
Let L be a set of SLD-derivations.
RemSub(L) = { D ∈ L | L does not contain a proper subderivation of D }.
L is *subderivation free* if L = RemSub(L). □

In order to render the intuitive meaning of a loop check L: 'every derivation D ∈ L is pruned *exactly* at its last node', we need that L is subderivation free. Note that RemSub(RemSub(L)) = RemSub(L).

In the following definition, by a *variant* of a derivation D we mean a derivation D' in which in every derivation step, atoms in the same positions are selected and the same program clauses are used. D' may differ from D in the renaming that is applied to these program clauses for reasons of standardizing apart and in the mgu used. Thus (see [LS]) variants differ only in the choice of the names of the variables.

DEFINITION 2.2.
A *simple loop check* is a computable set L of finite SLD-derivations such that L is closed under variants and subderivation free. □

In [BAK], loop checks are treated in a more general way. There non-simple loop checks occur: their behaviour may depend on the *program* the interpreter is confronted with. For the topics addressed in this paper, the distinction between simple and non-simple loop checks does not play a role. For simplicity in the presentation, we shall only consider simple loop checks, but usually omit the qualification 'simple'.

DEFINITION 2.3.
Let L be a loop check. An SLD-derivation D of $P \cup \{G\}$ is *pruned by L* if L contains a subderivation D' of D. □

2.2 Soundness and completeness

The most important property is definitely that using a loop check does not result in a loss of success. Even stronger, because we use here a PROLOG-like interpreter augmented with a loop check as the *only* inference mechanism, we may not want to lose any individual solution. That is, if the original tree contains a successful branch (giving some computed answer), then we require that the pruned tree contains a successful branch giving a more general answer.

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, where for a derivation D, $|D|$ stands for its length, i.e. the number of goals in it.

DEFINITION 2.4 (Soundness).

i) A loop check L is *weakly sound* if for every program P and goal G, and SLD-tree T of $P \cup \{G\}$: if T contains a successful branch, then $f_L(T)$ contains a successful branch.

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

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

The following lemma is an immediate consequence of these definitions.

LEMMA 2.5. *Let L be a loop check.*

i) If L is shortening, then L is sound.

ii) 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 2.6 (Completeness).

A loop check L is *complete w.r.t. a selection rule* $\mathbf{R}$ *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 $\mathbf{R}$ is pruned by L. □

In general, comparing loop checks is difficult. The following relation comparing loop checks is not very general: most loop check will be incomparable with respect to it. Nevertheless it turns out to be very useful.

DEFINITION 2.7.

Let L_1 and L_2 be loop checks. L_1 is *stronger than* L_2 if every SLD-derivation $D_2 \in L_2$ contains a subderivation $D_1 \in L_1$. □

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. The following theorem enables 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 2.8 (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 then L_1 is complete.

PROOF. Straightforward. □

2.3. Loop checks comparing goals

After these basic definitions, three groups of simple loop checks are presented in [BAK]: equality checks, subsumption checks and context checks; a summary of this presentation is given in the Appendix. These loop checks are all based on the comparison of goals: a derivation is pruned as soon as a goal occurs in it that 'is sufficiently similar' to a previous goal.

Obviously the exact criterion for 'being sufficiently similar' is the essence of a loop check. This criterion, in addition to the two goals that are compared, may use some further information about the derivation D, such as the mgu's used, the initial goal (for the resultant-based checks) and the ancestry relation among atoms (for the context checks). However, when too much extra information is used, one may doubt if the loop check really 'compares goals'. It is difficult, if at all possible, to give a precise limit on the amount and the nature of 'other information' that may be used by the criterion. Therefore we refrain from giving a fully exact definition, relying instead on the intuition of the reader.

DEFINITION 2.9.

A *full-comparison* loop check is a loop check of the form

$$L(\varphi) = \mathrm{RemSub}(\{\, D \mid D = (\, G_0 \Rightarrow_{C_1,\theta_1} G_1 \Rightarrow \ldots \Rightarrow G_{k-1} \Rightarrow_{C_k,\theta_k} G_k) \text{ and } \exists\, i < k \text{ such that } \varphi(G_i,G_k,D) \,\}),$$

where $\varphi(G_i,G_k,D)$ 'essentially compares the goals G_i and G_k':
$\varphi(G_i,G_k,D)$ = true if and only if G_i and G_k 'are sufficiently similar'.
The relation φ is called the *loop checking criterion* of $L(\varphi)$. □

The condition that φ 'essentially compares goals' implies for example that the effort of computing $\varphi(G_i,G_k,D)$ is independent of $|D|$. Therefore the number of φ-computations (*comparisons*) performed by a loop check is a good measure of the overhead caused by the loop check, as was tacitly assumed in the introduction.

LEMMA 2.10 [C]. *On a finite SLD-derivation D, a full-comparison loop check performs* $\frac{1}{2}|D|(|D|-1)$ *comparisons.*

PROOF. Obvious. □

3. The tortoise-and-hare technique

A first attempt to reduce the number of comparisons performed by a loop check is presented in [vG]. There every goal G_k is compared with exactly one previous goal, namely the goal $G_{k/2}$ ($G_{(k-1)/2}$ if k is odd) 'halfway' the derivation. The name of the method originates from the technique used to keep track of the goals G_k and $G_{k/2}$ in the derivation: a fast (every derivation step) moving pointer (the hare) points at the 'current' goal G_k, a slow (every other step) moving pointer (the tortoise) points at the goal $G_{k/2}$ 'halfway'. We now formalize this technique.

DEFINITION 3.1 (Tortoise-and-hare technique).
Let φ be a loop checking criterion. The *tortoise-and-hare* loop check of φ is the loop check $L^{th}(\varphi) = \text{RemSub}(\{ D \mid D = (G_0 \Rightarrow_{C_1,\theta_1} G_1 \Rightarrow \ldots \Rightarrow G_{k-1} \Rightarrow_{C_k,\theta_k} G_k)$
and $(k = 2i \text{ or } k = 2i+1)$ and $\varphi(G_i,G_k,D)$ and $k > 0$ $\})$. □

The following theorem is an immediate consequence of our previous results.

THEOREM 3.2 (Soundness). *Let φ be a loop checking criterion. If $L(\varphi)$ is weakly sound (sound,shortening) then $L^{th}(\varphi)$ is weakly sound (sound,shortening).*
PROOF. This follows from the Relative Strength Theorem 2.8, as $L(\varphi)$ is obviously stronger than $L^{th}(\varphi)$. □

Van Gelder justifies the use of the tortoise-and-hare technique by the observation that due to the use of the leftmost selection rule and the fixed order of clauses in PROLOG every loop must have a fixed length, say l (assuming no side-effects occur). As the distance between the tortoise and the hare continuously increases by 1, the loop is detected (after the tortoise enters the looping part of the derivation) as soon as the distance between the tortoise and the hare is a multiple of l.

In [vG] a looping derivation is not pruned automatically: it is suggested that control should be returned to the user, once a loop is detected. (Which makes sense there: the initial loop checking criterion that is proposed in [vG], and to which the tortoise-and-hare technique is added, is not even weakly sound, so it is up to the user to determine whether the derivation is *really* in a loop.) In our setting, a pruned goal is handled as a failed one, giving rise to backtracking. As is implicit in Definition 3.1 (and explicitly mentioned in [vG]), during backtracking the tortoise and hare motions are simply 'undone'.

This entails however that the fixed order of clauses in PROLOG cannot be relevant for a demonstration of the completeness of the method: no distinction is made between the application of a clause as a first attempt to solve a goal, or its application as a later attempt after backtracking from previous (failed) attempts. Indeed the tortoise-and-hare technique does not preserve completeness, as the following counterexample shows.

COUNTEREXAMPLE 3.3.
Let $P = \{ p \leftarrow p ; p \leftarrow q ; q \leftarrow p ; q \leftarrow q \}$. Let φ be a loop checking criterion such that $\varphi(p,p,D) = \varphi(q,q,D) = \text{true}$ and $\varphi(p,q,D) = \varphi(q,p,D) = \text{false}$ for every derivation D (as one would expect). Let T be the SLD-tree of $P \cup \{\leftarrow p\}$ pruned by $L^{th}(\varphi)$.
CLAIM. T contains one infinite branch, so $L^{th}(\varphi)$ is incomplete for P.
PROOF. Let G be a goal in T that is not pruned. We prove that G has two immediate descendants, of which only one is pruned. Regardless of G being $\leftarrow p$ or $\leftarrow q$, G has two descendants $G_1 = \leftarrow p$ and $G_2 = \leftarrow q$, which are both compared with the same 'halfway' goal H. If $H = \leftarrow p$, then G_1 is pruned but G_2 is not; if $H = \leftarrow q$, then G_2 is pruned but G_1 is not. □

4. Selected loop checks

An easy generalization of Counterexample 3.3 shows that a loop check cannot be complete if there exists a maximum N such that every goal is compared with at most N other goals (at least not if N is smaller than the number of ground atoms in the language). Therefore we adopt a different strategy here: an infinite selection S of natural numbers is made, and a pair of goals (G_i,G_k) is compared if and only if
- $i < k$ and $k \in S$ (single selected loop checks), respectively
- $i < k$ and $i,k \in S$ (double selected loop checks).

DEFINITION 4.1 (Selected Loop Checks).
Let φ be a loop checking criterion and let S be an infinite subset of $\mathbb{N}$.
The *single selected* loop check of φ and S is the loop check

$$L^1(\varphi,S) = \text{RemSub}(\{ D \mid D = (G_0 \Rightarrow_{C_1,\theta_1} G_1 \Rightarrow \ldots \Rightarrow G_{k-1} \Rightarrow_{C_k,\theta_k} G_k) \text{ and } \exists\, i < k \text{ such that } \varphi(G_i,G_k,D), \text{ and } k \in S \}).$$

The *double selected* loop check of φ and S is the loop check

$$L^2(\varphi,S) = \text{RemSub}(\{\ D \mid D = (\ G_0 \Rightarrow_{C_1,\theta_1} G_1 \Rightarrow \ldots \Rightarrow G_{k-1} \Rightarrow_{C_k,\theta_k} G_k)$$
$$\text{and } \exists\ i < k \text{ such that } \varphi(G_i,G_k,D), \text{ and } i,k \in S\ \}).$$

S is called the *selection* of $L^1(\varphi,S)$ or $L^2(\varphi,S)$. □

Clearly, the number of comparisons performed by a selected loop check depends on the selection S. For $S = \mathbb{N}$, we obtain the full-comparison loop checks again, for which the number of comparisons is quadratic in the number of goals generated. In section 5, the efficiency of double selected loop checks with $S = \{\frac{1}{2}i(i+1) \mid i \in \mathbb{N}\}$ is studied in detail. In the rest of this section, we do not consider any specific selection, but rather study the soundness and completeness of selected loop checks in general.

The following lemma enables the use of the Relative Strength Theorem 2.8.

LEMMA 4.2. *Let φ be a loop checking criterion and let S_1 and S_2 be selections.*

Then, *i) $L^1(\varphi,S_1)$ is stronger than $L^2(\varphi,S_1)$ and*

if $S_1 \supseteq S_2$ then *ii) $L^1(\varphi,S_1)$ is stronger than $L^1(\varphi,S_2)$ and*

iii) $L^2(\varphi,S_1)$ is stronger than $L^2(\varphi,S_2)$.

PROOF. Obvious. □

In particular, the full-comparison loop check $L(\varphi) = L^1(\varphi,\mathbb{N}) = L^2(\varphi,\mathbb{N})$ is stronger than any selected loop check using the criterion φ. This enables us to derive the soundness of a selected loop check from the soundness of the corresponding full-comparison loop check.

THEOREM 4.3 (Soundness of Selection). *Let φ be a loop checking criterion. If $L(\varphi)$ is weakly sound (sound, shortening) then for every selection S: $L^1(\varphi,S)$ and $L^2(\varphi,S)$ are weakly sound (sound, shortening).*

PROOF. By Lemma 4.2 and the Relative Strength Theorem 2.8. □

Combining this theorem with the soundness results for the simple loop checks presented in [BAK] (see the Appendix) yields the following results.

COROLLARY 4.4. *For every selection used,*

i) the (single and double) selected equality, subsumption and context checks based on goals are weakly sound and

ii) the (single and double) selected equality, subsumption and context checks based on resultants are shortening.

PROOF. By Theorem 4.3 and Theorem 7.1. □

Unfortunately, an equally general completeness result cannot be obtained using Lemma 4.2. Instead, generalizing the completeness results from the simple loop checks of [BAK] to the corresponding selected loop checks requires a detailed analysis of the completeness proofs in [BAK]. (However, by Lemma 4.2 it suffices to consider only double selected loop checks.)

For the equality checks and subsumption checks, this generalization is straightforward. By definition, a loop check is complete if every 'possible' infinite derivation (given an initial goal and a program satisfying the restrictions) contains two goals that 'are sufficiently similar' for the loop check. However, in the relevant proofs in [BAK] (notably Theorem 4.18, Lemma 5.15 and Theorem 5.20) a stronger result is proven: every infinite sequence of *unrelated* goals contains two 'similar' goals. Although in [BAK] this sequence is always taken $\{G_i \mid i \in \mathbb{N}\}$, the sequence $\{G_i \mid i \in S\}$ can be used for any selection S. Hence the completeness results for equality and subsumption checks (see Theorem 7.3) generalize immediately to selected equality and subsumption checks. The classes of programs mentioned below are defined in the Appendix, in Definition 7.2.

COROLLARY 4.5.

i) All (single and double) selected equality checks are complete w.r.t. the leftmost selection rule for function-free restricted programs.

ii) All (single and double) selected subsumption checks are complete w.r.t. the leftmost selection rule for function-free restricted programs.

iii) All (single and double) selected subsumption checks are complete for function-free nvi programs.

iv) All (single and double) selected subsumption checks are complete for function-free svo programs.

PROOF. By the arguments given above. □

For the context checks, the generalization of the completeness results is less straightforward, but still possible.

THEOREM 4.6. *All (single and double) selected context checks are complete for function-free nvi programs and for function-free svo programs.*

PROOF. Let S be a selection. In the completeness proofs for the full-comparison context checks (Theorem 6.14 and 6.16 in the revised version of [BAK]), an infinite sequence of goals $G_{m_0},G_{m_1},\ldots$ $(0 \le m_0 < m_1 < \ldots)$ is constructed in which 'similar' goals are shown to occur. The selection $M = (m_0 < m_1 < \ldots)$ can be adapted to the selection $S = (s_0 < s_1 < \ldots)$: we define the new selection $T = (t_0 < t_1 < \ldots)$ by:

- $t_0 = s_0$,
- $t_i = \min\{s \in S \mid \exists\, m \in M$ such that $t_{i-1} \le m < s\}$ for $i > 0$.

As in the sequence $G_{m_0},G_{m_1},\ldots$, in the goals of the sequence $G_{t_0},G_{t_1},\ldots$ atoms $A_0,A_1,\ldots$ occur such that A_{i+1} is the result (directly or indirectly) of resolving A_i. The 'interleaving' with the selection M is needed to ensure that A_{i+1} is not just an instantiated version of A_i, but indeed the result of at least one resolution step performed on A_i. (This follows from the observation that A_i is the selected atom in G_{m_i}; so in Theorem 6.14 exactly one resolution step occurs between A_i and A_{i+1}.)

In the rest of the proof of Theorem 6.14 (and 6.16), the sequence M can be replaced by T without any difficulty. Therefore the double selected CVR check using the selection T is complete for function-free nvi programs and for function-free svo programs. By Lemma 4.2 ($T \subseteq S$) and the Relative Strength Theorem 2.8, the same holds for all (single and double) selected context checks using the selection S. □

Surprisingly, selected context checks are not necessarily complete w.r.t. the leftmost selection rule for function-free restricted programs, as the following counterexample shows.

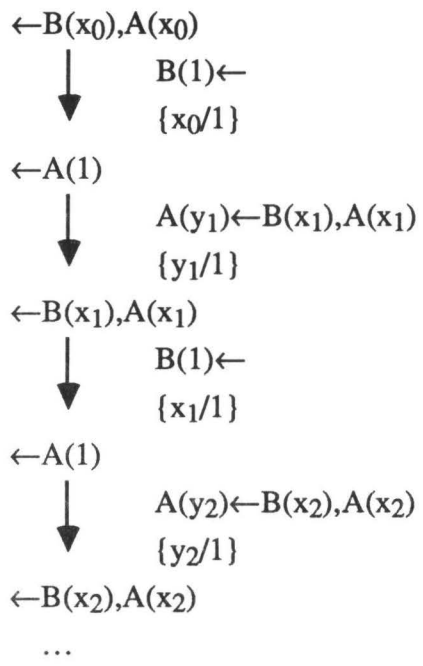
COUNTEREXAMPLE 4.7.

Let $P = \{\ A(y) \leftarrow B(x),A(x),$

$\qquad B(1) \leftarrow \qquad\qquad \}$

and let $G = \leftarrow B(x_0),A(x_0)$.

Consider the following derivation D of $P \cup \{G\}$ via the leftmost selection rule:



D is not pruned by the single selected context checks using the selection $S = \{2i \mid i \in \mathbb{N}\}$. First compare $G_{2j} = \leftarrow B(x_j),A(x_j)$ with $\leftarrow A(1)$. But $A(x_j)$ is not an instance of $A(1)$. Then compare it with $G_{2i} = \leftarrow B(x_i),A(x_i)$ $(i < j)$. $B(x_j)$ is not the result of resolving $B(x_i)$, so we are forced to take $A = A(x_i)$ and $\tau = \{x_i/x_j, x_j/x_i\}$. Then $\theta_{2i+1} = \{x_i/1\}$ and $A(x_j)$ is indeed the result of resolving $A(1) = A(x_i)\theta_{2i+1}$ in G_{2i+1}. But τ and $\theta_{2i+1}\ldots\theta_{2j}$ should agree on x_i, which they do not. □

Concluding, in most cases a selected loop check can be used instead of the corresponding full-comparison loop check, without losing its benefits such as soundness and completeness. In the next section we take a more constructive attitude towards selected loop checks and we investigate how much is gained by using them.

5. Triangular loop checks: a case study

This section presents a detailed study of double selected loop checks with the selection $S = \{\frac{1}{2}i(i+1) \mid i \in \mathbb{N}\}$. Numbers of the form $\frac{1}{2}i(i+1)$ are usually called *triangular* numbers, therefore we call such loop checks *triangular* loop checks. In this section the loop checking criterion is not relevant, as we focus solely on the number of comparisons performed.

THEOREM 5.1. *Let D be a finite SLD-derivation. The number of comparisons performed on D by a triangular loop check is less than |D|.*

PROOF. Let $D = (G_0 \Rightarrow_{C_1,\theta_1} G_1 \Rightarrow \ldots \Rightarrow G_{k-1} \Rightarrow_{C_k,\theta_k} G_k)$. For every triangular number $n = \frac{1}{2}i(i+1)$ $(0 \le n \le k)$, the goal G_n is compared to i previous goals. We may assume that k is a triangular number, say $k = \frac{1}{2}j(j+1)$. The number of comparisons performed on D is then $\sum_{i=0}^{j} i = \frac{1}{2}j(j+1) = k < k + 1 = |D|$. □

The following arrangement of goals may help the intuition:

$G_0 : G_1\ G_2\ G_4\ G_7\ \ldots$ $G_3\ G_5\ G_8$ $G_6\ G_9$ G_{10}	Every column contains one 'triangular' goal G (a goal with a triangular number as its index; this index is exactly the level of the goal). The number of goals in the column of G equals the number of columns preceding it (for $G \neq G_0$), which in turn equals the number of comparisons performed at G.

So the presence of G_7, G_8, G_9 and G_{10} 'justifies' the four comparisons performed at G_{10}. This arrangement of goals also explains the word 'triangular'.

When SLD-trees are considered, the situation gets more complicated: two goals G_{10} and G_{10}' may have common ancestors G_7, G_8 and G_9; these five goals cannot completely justify the eight comparisons performed at G_{10} and G_{10}'. We now show that for SLD-trees with a constant (average) branching factor and containing a 'reasonable' number of goals (say $\leq 10^{10}$), the number of comparisons performed is less than five times the number of goals generated.

THEOREM 5.2. *Let T consist of the levels 0,...,k of an SLD-tree, have a constant average branching factor b and contain $n \leq 10^{10}$ goals. Then a triangular loop check performs less than $5 \cdot n$ comparisons on T.*

PROOF. We may assume that $b > 1$ (for $b = 1$ (or $b < 1$), see Theorem 5.1) and that the depth of T is a triangular number, say $k = \frac{1}{2}j(j+1)$. Then for $0 \leq m \leq k$, the number of

- goals at level m is b^m,
- goals in T is $\sum_{i=0}^{k} b^i = \frac{b^{k+1}-1}{b-1} = n \leq 10^{10}$,
- comparisons at level $m = \frac{1}{2}i(i+1)$ is $i \cdot b^m$,
- comparisons in T is $\sum_{i=1}^{j} i \cdot b^{1/2 \cdot i(i+1)}$.

We consider two cases.

CASE 1: $b^{j-1} \leq 5$.

In this case the number of goals at level $\frac{1}{2}i(i+1)$ $(1 \leq i \leq j)$ can be at most 5 times the number of goals at level $\frac{1}{2}i(i-1)+1$. Therefore the goals between level $\frac{1}{2}i(i-1)+1$ and level $\frac{1}{2}i(i+1)$ justify at least one fifth of the comparisons at level $\frac{1}{2}i(i+1)$.

Formally, $\sum_{i=1}^{j} i \cdot b^{1/2 \cdot i(i+1)} = \sum_{i=1}^{j} b^{i-1} \cdot i \cdot b^{1/2 \cdot i(i-1)+1} \leq b^{j-1} \cdot \sum_{i=1}^{j} i \cdot b^{1/2 \cdot i(i-1)+1} \leq$

$5 \cdot \sum_{i=1}^{j} \sum_{r=1}^{i} b^{1/2 \cdot i(i-1)+1} \leq 5 \cdot \sum_{i=1}^{j} \sum_{r=1}^{i} b^{1/2 \cdot i(i-1)+r} = 5 \cdot \sum_{l=1}^{k} b^{l} < 5n$. The final equation is justified by the observation that every level-number l $(1 \leq l \leq k)$ can be written as $l = t+r$, where $t = \frac{1}{2}i(i-1)$ is the largest triangular number smaller than l and $1 \leq r \leq i$.

CASE 2: $b^{j-1} > 5$.

In this case the total number of comparisons can be estimated at $\frac{5}{4}$ times the number of comparisons at the last level, since the number of comparisons at level $\frac{1}{2}j(j+1)$ is $j \cdot b^{1/2 \cdot j(j+1)} > b^{j} \cdot (j-1) \cdot b^{1/2 \cdot j(j-1)} > 5$ times the number of comparisons at level $\frac{1}{2}j(j-1)$ (which is in turn > 5 times the number of comparisons at level $\frac{1}{2}(j-1)(j-2)$; now we have $1 + \frac{1}{5} + \frac{1}{25} + \ldots = \frac{5}{4}$).

Therefore $\frac{\text{the number of comparisons in T}}{\text{the number of goals in T}} = \frac{5 \cdot j \cdot b^{k}}{4 \cdot n} = \frac{5 \cdot j \cdot b^{k} \cdot (b-1)}{4 \cdot (b^{k+1}-1)} \approx \frac{5j(b-1)}{4b}$.

(Notice that $b^{j-1} > 5$ implies $b^{k+1} > 125 \gg 1$.)

Finally $k = \frac{1}{2}j(j+1)$ and $\frac{b^{k+1}-1}{b-1} \leq 10^{10}$ implies $j \leq -\frac{1}{2} + \sqrt{-\frac{7}{4} + 2 \cdot {}^{b}\log(10^{10}(b-1))}$. A numeric analysis[1] of the function $\frac{5(b-1)}{4b}\left(-\frac{1}{2} + \sqrt{-\frac{7}{4} + 2 \cdot {}^{b}\log(10^{10}(b-1))}\right)$ shows that its maximum is almost 5 (≈ 4.95 for $b \approx 3.21$). □

Finally we consider SLD-trees which do not have a constant average branching factor, but exhibit a kind of 'worst case' behaviour. In these trees only the parents of the 'triangular' goals have more than one descendant. More formally, if b_k is the number of descendants of a goal at level k ($k \geq 0$), then

$$b_k = \begin{cases} b & \text{if } k = \frac{1}{2}j(j+1)-1 \ (j > 0) \\ 1 & \text{otherwise,} \end{cases}$$

for some constant branching factor b.

---

[1] Perfomed using the 'Maple' package, developed by the Symbolic Computation Group of the University of Waterloo, Ontario, Canada.

THEOREM 5.3. *Let T consist of the levels 0,...,k of a 'worst case' SLD-tree with branching factor b, and contain n goals. Then a triangular loop check performs less than b·n comparisons on T. Moreover, if $n \leq 10^{10}$, then a triangular loop check performs less than 6·n comparisons on T.*

PROOF. Let $k = \frac{1}{2}j(j+1)$. The number of goals at level k is then $\prod_{i=0}^{k} b_k = b^j$. Hence the number of comparisons performed at level k is $j \cdot b^j$. Each of the levels $\frac{1}{2}j(j-1)+1,\ldots,\frac{1}{2}j(j+1)-1$ consists of b^{j-1} goals, giving $(j-1)\cdot b^{j-1}$ goals together. So for the $j \cdot b^j$ comparisons at level k, there are $(j-1)\cdot b^{j-1}+b^j$ 'justifying' goals, giving $\frac{j \cdot b}{j-1+b}$ comparisons per goal. It is easy to show that $b > 1$ implies $\frac{(j-1)\cdot b}{(j-1)-1+b} < \frac{j \cdot b}{j-1+b}$, so the overall ratio in T is less than $\frac{j \cdot b}{j-1+b}$ comparisons per goal. First notice that $b > 1$ implies $\frac{j \cdot b}{j-1+b} < b$, which proves the first claim.

Now $n \leq 10^{10}$ implies $b^j < 10^{10}$, so $j < {}^b\log(10^{10})$. A numeric analysis of the function $\frac{{}^b\log(10^{10})\cdot b}{{}^b\log(10^{10})-1+b}$ shows that its maximum is almost 6 (≈ 5.76 for $b \approx 21$), which proves the second claim. □

6. Conclusions

The obvious conclusion is that the number of comparisons performed by a triangular loop check is (almost) linear in the number of goals generated. For any realistic number of generated goals n, the number of comparisons performed is at most 6·n. So triangular loop checks satisfy the requirement stated in the introduction. Moreover, unlike the tortoise-and-hare technique, which was motivated by the same requirement, the 'triangular' technique retains the completeness of the corresponding full-comparison loop checks (with the exception of Counterexample 4.7). The only minor disadvantage of the 'triangular' technique might be that the comparisons are not distributed smoothly over the goals, which makes the timing of the interpreter less predictable.

Other selections give rise to other efficiency results: a more sparse selection yields a more efficient loop check, relative to the number of goals generated. However, using a sparse selection is not necessarily the best thing to do: loops are detected later, so the overall effort of generating goals and loop checking may well become larger than with a less sparse selection loop check. A further analysis which selection might be most favorable (or even an investigation of the circumstances on which this depends) is beyond the scope of this paper.

7. Appendix

Here we recall the three groups of simple loop checks that are introduced in [BAK], together with their respective soundness and completeness results.

7.1 Definitions and soundness results

First we present the weakly sound loop checks of each group.

The first group consists of the equality checks. Their loop checking criterion has the form 'for some substitution τ: $G_k = G_i\tau$' (or in words: 'G_k is an instance of G_i'). Small variations on this criterion give rise to various loop checks within this group. These variations are notably the two interpretations of '=' that are considered (goals can be treated as lists or as multisets) and the possible addition of the requirement 'τ is a renaming' (in other words: 'G_k is a variant of G_i'). Such variations can also be made within the other groups of loop checks, but as it appears that these variations have not much effect on soundness and completeness, we shall not mention them any more.

The second group consists of the subsumption checks. Their loop checking criterion has the form 'for some substitution τ: $G_k \subseteq G_i\tau$' (or in words: 'G_k is subsumed by an instance of G_i'). Although the replacement of = by $\subseteq$ seems to be yet another small variation, it appears that subsumption checks are really more powerful than equality checks.

The third group consists of the context checks, introduced by Besnard [B]. Their loop checking condition is more complicated: 'For some atom A in G_i, $A\theta_{i+1}\ldots\theta_j$ is selected in G_j to be resolved. As the (direct or indirect) result of resolving $A\theta_{i+1}\ldots\theta_j$, an instance $A\tau$ of A occurs in G_k ($0 \le i \le j < k$). Finally, for every variable x that occurs both inside and outside of A in G_i, $x\theta_{i+1}\ldots\theta_k = x\tau$.'

For all these weakly sound loop checks, a shortening counterpart is obtained by adding the condition '$G_0\theta_1\ldots\theta_k = G_0\theta_1\ldots\theta_i\tau$' to the loop checking criterion. For reasons explained in [BAK], the loop checks thus obtained are called 'based on resultants' as opposed to the weakly sound ones, which are 'based on goals'.

Thus the following results were proved in [BAK].

THEOREM 7.1.

i) The equality, subsumption and context checks based on goals are weakly sound.

ii) The equality, subsumption and context checks based on resultants are shortening. □

7.2 Completeness results

Due to the undecidability of the halting problem, a weakly sound loop check cannot be complete for all programs. In [BAK] it was shown that a weakly sound *simple* loop check cannot even be complete for all *function-free* programs. Therefore three classes of function-free programs were isolated for which the completeness of (some of) the loop checks mentioned above could be proved. We now present those classes of programs and completeness results.

DEFINITION 7.2.

A program P is *restricted* if for every clause $H \leftarrow A_1,\ldots,A_n$ in P, the definitions of the predicates in $A_1,\ldots,A_{n-1}$ do not depend on the the predicate of H in P. (So recursion is allowed, namely through A_n, but double recursion is not; almost similar to [ŠŠ].)

A program P is *non-variable introducing* (*nvi*) if for every clause $H \leftarrow A_1,\ldots,A_n$ in P, every variable that occurs in $A_1,\ldots,A_n$ occurs also in H.

A program P has the *single variable occurrence* property (*is svo*) if for every clause $H \leftarrow A_1,\ldots,A_n$ in P, no variable occurs more than once in $A_1,\ldots,A_n$. □

THEOREM 7.3.

i) All equality, subsumption and context checks are complete w.r.t. the leftmost selection rule for function-free restricted programs.

ii) All subsumption and context checks are complete for function-free nvi programs.

iii) All subsumption and context checks are complete for function-free svo programs.

□

References

[B] Ph. BESNARD, *On Infinite Loops in Logic Programming*, Internal Report 488, IRISA, Rennes, 1989.

[BAK] R.N. BOL, K.R. APT and J.W. KLOP, *An Analysis of Loop Checking Mechanisms for Logic Programs*, Technical Report CS-R8942, Centre for Mathematics and Computer Science, Amsterdam; Technical Report TR-89-32, University of Texas at Austin, 1989. To appear in Theoretical Computer Science.

[BW] D.R. BROUGH and A. WALKER, *Some Practical Properties of Logic Programming Interpreters*, in: Proceedings of the International Conference on Fifth Generation Computer Systems, (ICOT eds.), 1984, 149-156.

[C] M.A. COVINGTON, *Eliminating Unwanted Loops in PROLOG*, SIGPLAN Notices, Vol. 20, No. 1, 1985, 20-26.

[vG] A. VAN GELDER, *Efficient Loop Detection in PROLOG using the Tortoise-and-Hare Technique*, J. Logic Programming 4, 1987, 23-31.

[KT] D.B. KEMP and R.W. TOPOR, *Completeness of a Top-Down Query Evaluation Procedure for Stratified Databases*, in: Proceedings of the Fifth International Conference on Logic Programming, (R. Kowalski and K. Bowen eds.), MIT Press, Cambridge Massachusetts, 1988, 178-194.

[L] J.W. LLOYD, *Foundations of Logic Programming*, Second Edition, Springer-Verlag, Berlin, 1987.

[LS] J.W. LLOYD and J.C. SHEPHERDSON, *Partial Evaluation in Logic Programming*, Technical Report CS-87-09, Dept. of Computer Science, University of Bristol, 1987.

[PG] D. POOLE and R. GOEBEL, *On Eliminating Loops in PROLOG*, SIGPLAN Notices, Vol. 20, No. 8, 1985, 38-40.

[SGG] D.E. SMITH, M.R. GENESERETH and M.L. GINSBERG, *Controlling Recursive Inference*, Artificial Intelligence 30, 1986, 343-389.

[SI] H. SEKI and H. ITOH, *A Query Evaluation Method for Stratified Programs under the Extended CWA*, in: Proceedings of the Fifth International Conference on Logic Programming, (R. Kowalski and K. Bowen eds.), MIT Press, Cambridge Massachusetts, 1988, 195-211.

[ŠŠ] O. ŠTĚPÁNKOVÁ and P. ŠTĚPÁNEK, *A Complete Class of Restricted Logic Programs*, in: Logic Colloquium '86, (F.R. Drake and J.K. Truss eds.), North Holland, Amsterdam, 1988, 319-324.

[V] L. VIEILLE, *Recursive Query Processing: The Power of Logic*, Theoretical Computer Science 69, No. 1, 1989, 1-53.