1991

R.N. Bol

Loop checking in partial deduction

Computer Science/Department of Software Technology Report CS-R9134 July

CWI, nationaal instituut voor onderzoek op het gebied van wiskunde en informatica

CWI is the research institute of the Stichting Mathematisch Centrum, which was founded on February 11, 1946, as a non-profit institution aiming at the promotion of mathematics, computer science, and their applications. It is sponsored by the Dutch Government through the Netherlands organization for scientific research (NWO).

Loop Checking in Partial Deduction

Roland N. Bol

CWI
P.O. Box 4079, 1009 AB Amsterdam, The Netherlands
Phone: (+31) - 20 - 592 4080. E-mail: bol@cwi.nl.

Abstract. In the framework of [LS], partial deduction involves the creation of SLD-trees for a given program and some goals, up to certain halting points. This paper identifies the relation between halting criteria for partial deduction and loop checking (as formalized in [BAK]). It appears that loop checks for partial deduction must be complete, whereas traditionally the soundness of a loop check is more important. However, it is also shown that sound loop checks can contribute to improve partial deduction. Finally, a class of complete loop checks suitable for partial deduction is identified.

Key Words and Phrases: Logic programming, loop checking, termination, program transformation, partial evaluation.
1985 Mathematics Subject Classification: 68Q40, 68T15.
1987 CR Categories: D.3.4, F.3.2, F.4.1, H.3.3, I.2.2, I.2.3.
Notes: This research was partly supported by Esprit BRA-project 3020 Integration.
This paper will appear in the Journal of Logic Programming.

1. INTRODUCTION

Although partial evaluation dates back to the 1970's, and was introduced into logic programming in the early 1980's ([Ko]), the topic only recently has attracted more substantial attention (e.g. [BEJ]). The foundations of partial evaluation in pure logic programming have been thoroughly studied in [LS]; we follow their framework here. Their method is more appropriately called *partial deduction* nowadays, leaving the term partial evaluation for works taking into account certain non-logical features of PROLOG, as is done in e.g. [S1, S2].

The following intuitive description of partial deduction is given in [LS]: 'Given a program P and a goal G, partial evaluation produces a new program P', which is P "specialized" to the goal G. The intention is that G should have the same answers w.r.t. P and P', and that G should run more efficiently for P' than for P. The basic technique for obtaining P' from P is to construct "partial" search

trees for P and suitably chosen atoms as goals, and then extract P' from the definitions associated with the leaves of these trees.'

Thus for some atoms A a finite part of an SLD-tree of $P \cup \{\leftarrow A\}$ must be constructed. This paper will not address the choice of the atoms, but concentrate on the question which part of the SLD-tree must be constructed, or, conversely, where the construction of the SLD-tree must be stopped. This question is of fundamental importance, as a sufficient stopping criterion is necessary to prevent the partial deduction from looping.

In the literature it is often noted that these stopping criteria are 'very closely related to the problems of loop trapping' ([LS]). But a precise connection was never made, probably because there was no formal theory of loop trapping to connect to. Either the problem was only identified as being 'difficult', or for practical purposes ad hoc solutions were used.

Recently a framework for the analysis of loop checking mechanisms has been presented in [BAK], together with some particular loop checks intended to be incorporated in a PROLOG-like interpreter for use at run-time. One of the aims of this paper is to show that the framework of [BAK] is sufficiently general for describing loop checks suitable for partial deduction as well.

The plan of this paper is as follows. Section 2 contains all preliminaries regarding partial deduction, illustrated by an example. Section 3 recalls the basic definitions of loop checking, as presented in [BAK]. A group of loop checks, the equality checks, is studied as an example.

In Section 4, it is shown that the termination criteria for partial deduction can indeed be described as loop checks, but that their characteristics are different from the 'ordinary' loop checks that can be used at run-time. More precisely, a loop check used for partial deduction must first of all remove all infinite derivations (completeness), whereas for ordinary loop checks the most important requirement is that no solutions are lost (soundness). Due to the unsolvability of the halting problem, these two requirements are generally incompatible.

We also show in this section how in conjunction with a complete loop check (which enforces termination) a sound loop check can be used to remove some loops from the program P' obtained by partial deduction. To this end the example of Section 2 is reconsidered. Adding a sound loop check to the partial deduction procedure is probably less costly than adding it to a PROLOG-like

interpreter, as most information needed for it (such as previous goals) must be maintained for the complete loop check anyway.

The importance of sound loop checks has been sufficiently stressed in the literature (e.g. in [BAK]). Complete loop checks have not yet received that much attention. Section 5 contains some general observations about complete loop checks (notably their relation with selection rules) and describes a class of complete loop checks that is inspired by some typical examples proposed in [S1]. Furthermore, the relationship with [BdSM] is discussed.

In [LS], programs with negation are considered and SLDNF-resolution is used, making the distinction between finite and infinite failure significant. As shown in [B2], the use of a sound loop check can be combined with (stratified) negation, but because infinite failure can be turned into finite failure by such a loop check, it was more natural to use SLS-resolution ([P]) there. In order to avoid unnecessary complications, we restrict ourselves here to programs without negation and the use of SLD-resolution.

2. PARTIAL DEDUCTION

In this section we recall the basic concepts of partial deduction, as introduced in [LS]. Knowledge of the basic theory and terminology of logic programming, as can be found in [L], is assumed. 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 $E \leq F$ when F is an instance of E. An SLD-derivation step from a goal G, using a clause C and an idempotent mgu θ, to a goal H is denoted as $G \Rightarrow_{C,\theta} H$.

In contrast with [L], where SLD-derivations must always be finished (being successful, failed or infinite), here SLD-derivations can also be *unfinished*, in the sense that a goal can have no selected literal, in which case the derivation ends at that goal. Similarly, SLD-trees can be unfinished too. An unfinished SLD-derivation or -tree that consists solely of the initial goal is called *trivial*. If an SLD-derivation D is finite, then $|D|$ denotes its length, i.e. the number of resolution steps in it.

Given a goal $G = \leftarrow A_1,\ldots,A_n$, $G^{\sim}$ denotes the formula $A_1 \wedge \ldots \wedge A_n$. With each goal in an SLD-derivation $G_0 \Rightarrow_{C_1,\theta_1} G_1 \Rightarrow \ldots \Rightarrow G_{k-1} \Rightarrow_{C_k,\theta_k} G_k \Rightarrow \ldots$ we associate a *resultant*: for $i \geq 0$, the *resultant associated to* G_i is $G_0{}^{\sim}\theta_1\theta_2\ldots\theta_i$ if $G_i = \Box$, $G_0{}^{\sim}\theta_1\theta_2\ldots\theta_i \leftarrow G_i{}^{\sim}$ otherwise. Notice that if G_0 consists of a single atom, such a resultant is a program clause.

DEFINITION 2.1 (Partial deduction).
Let P be a program, A an atom and T a finite non-trivial SLD-tree of $P \cup \{\leftarrow A\}$. Let $G_1,\ldots,G_r$ be the leaves of T that are not failed (r = 0 is possible). Let $R_1,\ldots,R_r$ be the corresponding resultants. The set $\{R_1,\ldots,R_r\}$ is called a *partial deduction for A in P*.

For a set of atoms $A = \{A_1,\ldots,A_s\}$, a *partial deduction for A in P* is the union of partial deductions for $A_1,\ldots,A_s$ in P.

A *partial deduction for P w.r.t. A* is a program obtained from P by replacing the set of clauses in P whose head contains one of the predicate symbols appearing in A by a partial deduction for A in P. □

DEFINITION 2.2 (Soundness and completeness of partial deduction).
Let P be a program and A a finite set of atoms. Let P' be a partial deduction for P w.r.t. A. Let G be a goal.

i) P' is *sound* w.r.t. P and G if every correct answer for $P' \cup \{G\}$ is correct for $P \cup \{G\}$.

ii) P' is *complete* w.r.t. P and G if every correct answer for $P \cup \{G\}$ is correct for $P' \cup \{G\}$. □

As SLD-resolution is sound and complete (w.r.t. the least Herbrand model semantics), one could equally well express these criteria by means of computed answers of SLD-refutations. In [LS] programs with negation are considered, using SLDNF-resolution ([Cl]) and completion semantics. Consequently their approach is more complicated in two ways.

First of all, SLDNF-resolution is generally not complete w.r.t. the completion semantics. So a distinction between declarative soundness and completeness of partial deduction (considering correct answers) and operational soundness and completeness (considering computed answers) must be made.

Secondly, they require more elaborate notions of soundness and completeness. In terms of semantics, having no correct answers for $P \cup \{\leftarrow A\}$ allows for two situations that must be distinguished, namely '$comp(P) \models \neg A$' and '$comp(P) \not\models A$ and $comp(P) \not\models \neg A$'. In terms of SLDNF-derivations, this relates to the distinction between finite and infinite failure.

Both in [LS] and in the more limited case studied here, it appears that partial deduction is always sound, but only complete under a certain condition.

DEFINITION 2.3.
Let S be a set of first order formulas and A a finite set of atoms. S is *A-closed* if each atom in S that contains a predicate symbol occurring in an atom in A is an instance of an atom in A. □

THEOREM 2.4. ([LS]) *Let P be a program, G a goal, A a finite set of atoms and P' a partial deduction for P w.r.t. A.*
i) P' is sound w.r.t. P and G.
ii) If $P' \cup \{G\}$ is A-closed, then P' is complete w.r.t. P and G. □

The following example shows a case in which partial deduction is traditionally useful: a meta-interpreter is specialized to a certain object program. The resulting program bears similarity to this object program: the meta-interpreter is 'compiled away'. Thus one level of interpretation is removed, an operation that usually leads to a considerable gain in efficiency.

The example also shows that the closedness condition is needed. In Section 4 this example reoccurs in combination with loop checking.

EXAMPLE 2.5.
Let P be the following variant of the 'vanilla'-interpreter, interpreting a small transitive closure program (translated in such a way that the PROLOG system predicate 'clause' has become a purely logical predicate; the predicate symbols denoting the base relation r and its transitive closure tc have become function symbols). Goals are represented as lists and the leftmost selection rule is always used. Notice that the addition of x_2 in the third clause for 'solve' avoids an infinite loop (or the use of a cut).

solve([]) ←.
solve([x]) ← clause(x,y),solve(y).
solve([x_1,x_2|y]) ← solve([x_1]),solve([x_2|y]).

clause(r(a,a),[]) ←.
clause(r(a,b),[]) ←.
clause(r(b,c),[]) ←.
clause(tc(x,y),[r(x,y)]) ←.
clause(tc(x,y),[r(x,z),tc(z,y)]) ←.

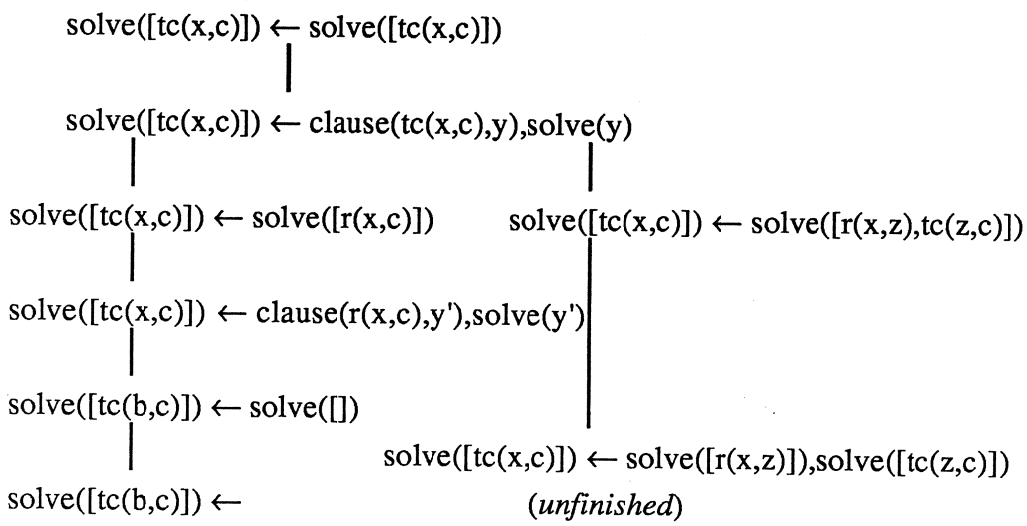

FIGURE 2.1

Taking **A** = {solve([tc(x,c)])}, the SLD-tree of Figure 2.1 can be constructed (the resultants are given). The partial deduction P_1 for P w.r.t. **A** is now obtained by replacing the clauses for 'solve' in P by:

solve([tc(b,c)]) ←.
solve([tc(x,c)]) ← solve([r(x,z)]),solve([tc(z,c)]).

The resulting program is *not* complete w.r.t. P∪{←solve([tc(x,c)])}: every call to solve([r(x,z)]) fails, only the answer substitution {x/b} is found. This is due to the fact that P_1 is not **A**-closed: the atom solve([r(x,z)]) occurs in P_1 and is not an instance of solve([tc(x,c)]). Thus solve([r(x,z)]) must be included in **A** and an SLD-tree of P∪{←solve([r(x,z)])} must be constructed (see Figure 2.2).

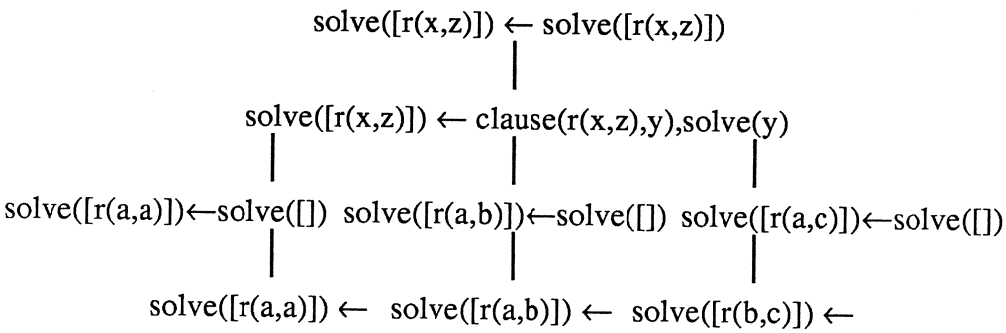

FIGURE 2.2

Thus the new partial deduction P_2 for P w.r.t. **A** contains for 'solve' the clauses

solve([r(a,a)]) ←.

solve([r(a,b)]) ←.

solve([r(b,c)]) ←.

solve([tc(b,c)]) ←.

solve([tc(x,c)]) ← solve([r(x,z)]),solve([tc(z,c)]).

Now $P_2 \cup \{\leftarrow solve([tc(x,c)])\}$ is **A**-closed and indeed P_2 is complete w.r.t. $P \cup \{\leftarrow solve([tc(x,c)])\}$. □

This short introduction to partial deduction leaves two questions unanswered (although the example gives some hints), namely:

- which set $\mathbf{A} = \{A_1 \ldots, A_s\}$ is best to be used, and
- how deep the SLD-trees of $P \cup \{\leftarrow A_1\}, \ldots, P \cup \{\leftarrow A_s\}$ should be expanded.

In this paper the second question is addressed, by relating the stopping criteria for the expansion of the SLD-trees used in partial deduction to loop checking.

3. LOOP CHECKING

In this section we recall some of the basic notions concerning loop checking, as introduced in [BAK]. At the end of this section, the concepts introduced are illustrated by an example.

3.1. 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 $P \cup \{G\}$, 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 in [TS, V].

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 | 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 3.1.
Let L be a set of SLD-derivations. *Initials(L)* = {D ∈ L | L does not contain a proper initial subderivation of D}. L is *subderivation free* if L = Initials(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 Initials(Initials(L)) = Initials(L).

DEFINITION 3.2 (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 3.3 (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 3.4.
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. □

3.2. Soundness and Completeness

In this section some basic properties of loop checks are introduced and some natural results concerning them are established. Here we concentrate on the use of a loop check at run-time. That is, the effect of adding a loop checking mechanism to a standard PROLOG-like interpreter is considered.

When used in this way, the most important property of a loop check is that using it 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 θ (thus proving $\forall G\sim\theta$), then we require that the pruned tree contains a successful branch giving a more general answer than θ, 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 3.5 (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 *sound* if for every program P, goal G, and SLD-tree T of P∪{G}: if T contains a successful branch with a computed answer G~σ, then $f_L(T)$ contains a successful branch with a computed answer G~σ' ≤ G~σ.

iii) A loop check L is *shortening* if for every program P, goal G, and SLD-tree T of P∪{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 G~σ' ≤ G~σ such that |D'| < |D|. □

The following lemma is an immediate consequence of these definitions.

LEMMA 3.6. *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 3.7 (Completeness).

A loop check L is *complete w.r.t. a selection rule* **R** *for a class of programs* $\mathcal{C}$, if for every program P ∈ $\mathcal{C}$ and goal G in L_P, every infinite SLD-derivation of P∪{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). 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.

A logic program can express (by semantical implication) that certain facts hold, but it cannot express that certain other facts do *not* hold. To overcome this shortcoming Reiter ([Re]) introduced the *closed world assumption* (CWA). Given a program P, CWA(P) = {¬A | A is a ground atom and P ⊭ A}. The soundness and completeness of SLD-resolution imply that CWA(P) = {¬A | A is a ground atom and every SLD-tree of P∪{←A} is failed}.

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.

The relationships between soundness and completeness of loop checks and the interpreters augmented with them are expressed in the following lemma's. 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. 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 3.8. *Let P be a program, A a ground atom and L a weakly sound loop check. Then for every SLD-tree T of* $P \cup \{\leftarrow A\}, \neg A \in CWA(P)$ *iff* $f_L(T)$ *contains no successful branches.*

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 3.9. *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* $G\tilde{}\theta$ *is a correct answer for* $P \cup \{G\}$ *iff* $f_L(T)$ *contains a successful branch with a computed answer* $G\tilde{}\tau \leq G\tilde{}\theta$.

Thus an interpreter augmented with a sound loop check remains sound. Moreover, a breadth-first interpreter remains complete .

COROLLARY 3.10. *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.*

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

COROLLARY 3.11. *Let P be a program, G a goal and L a sound and complete loop check. Then for every correct answer $G^{\sim}\theta$ 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 $G^{\sim}\tau \leq G^{\sim}\theta$.*

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.

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 3.12.
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.

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 3.13 (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 $\mathcal{C}$), then L_1 is complete (w.r.t. **R** for the class of programs $\mathcal{C}$).*

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 3.12. 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. □

Now we have a more clear view of the situation. Very strong loop checks prune derivations in an 'early stage'. If they prune too early, then they are unsound. Since this is undesirable, we must look for weaker loop checks. But a loop check should preferably be not too weak, for then it might fail to prune 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 undecidability of the halting problem implies that there cannot be a weakly sound and complete loop check for logic programs in general, as logic programming has the full power of recursion theory. It was shown in [BAK] that weakly sound and complete non-simple loop checks exist for programs without function symbols, so called *function-free* programs, for which the Herbrand Universe is finite. However, it was also shown that there is no weakly sound and complete *simple* loop check for function-free programs. Therefore, we found it useful to develop some simple loop checks, and to find classes of programs for which these loop checks are complete.

3.3. An example: the equality checks

In this subsection we introduce the equality checks. First we give a definition of the weakly sound versions. Then we introduce an additional condition that makes these checks shortening. Finally, we define the class of *restricted* programs: the equality checks are complete w.r.t. the leftmost selection rule for function-free restricted programs.

In fact, we should give a definition for each equality check. This would yield a number of almost identical definitions. Therefore we compress them into two definitions, trusting that the reader is willing to understand our notation. The

equality relation between goals (regarded as *lists*) is denoted by $=_L$. (In [BAK], also variants of these loop checks are considered, regarding goals as *multisets*.)

DEFINITION 3.14 (Equality checks based on Goals).
The *Equals Variant/Instance of* $Goal_{List}$ check is the set of SLD-derivations EVG/EIG_L = Initials({ D | D = ($G_0 \Rightarrow_{C_1,\theta_1} G_1 \Rightarrow \ldots \Rightarrow G_{k-1} \Rightarrow_{C_k,\theta_k} G_k$) such that for some i, $0 \leq i < k$, there is a renaming/substitution τ such that $G_k =_L G_i\tau$ }). □

The informal justification for these loop checks is as follows. Suppose that we want to refute a goal G. If we find that in order to refute G we need to refute a variant or instance of G, say $G\tau$, then two cases arise. If there is no solution for $G\tau$, then pruning $G\tau$ is clearly safe. On the other hand, if there is a solution for $G\tau$, then the derivation giving this solution might be used (possibly in a more general form) directly from G.

These loop checks are indeed weakly sound. However, they are not sound. To see this, suppose that we find for $G\tau$ a successful derivation D with a computed answer substitution σ. Then using D directly from G gives a computed answer substitution $\tau\sigma$ (maybe a more general substitution, but not necessarily). Therefore success is not lost. However, the derivation $G = G_i \Rightarrow_{C_{i+1},\theta_{i+1}} \cdots \Rightarrow_{C_k,\theta_k} G_k = G\tau$, followed by D, yields a possibly different computed answer substitution: $\theta_{i+1}\ldots\theta_k\sigma$, thus possibly affecting soundness. (In Example 3.16, we show a specific program and goal for which this difference arises.) Of course, we are only interested in computed answers, i.e. the resultants $G_0\theta_1\ldots\theta_i\theta_{i+1}\ldots\theta_k\sigma$ and $G_0\theta_1\ldots\theta_i\tau\sigma$, where G_0 is the initial goal. So τ and $\theta_{i+1}\ldots\theta_k$ should coincide on the variables of $G_0\theta_1\ldots\theta_i$.

Hence we can make these loop checks sound, and even shortening, by adding the condition $G_0\theta_1\ldots\theta_k = G_0\theta_1\ldots\theta_i\tau$. (Note that in this equality it is irrelevant whether goals are lists or multisets.)

Finally, note that adding this condition is equivalent to the replacement of the condition $G_k =_L G_i\tau$ by the condition $R_k =_L R_i\tau$, where R_k and R_i are the resultants associated to the goals G_k and G_i.

DEFINITION 3.15 (Equality checks based on Resultants).
The *Equals Variant/Instance of Resultant*$_{List}$ check is the set of SLD-derivations
EVR/EIR_L = Initials({D | D = ($G_0 \Rightarrow_{C_1,\theta_1} G_1 \Rightarrow \ldots \Rightarrow G_{k-1} \Rightarrow_{C_k,\theta_k} G_k$) such that for some i, $0 \leq i < k$, there is a renaming/substitution τ such that $G_k =_L G_i\tau$ and $G_0\theta_1\ldots\theta_k = G_0\theta_1\ldots\theta_i\tau$ }). □

The following example shows the difference between the goal-based and resultant-based equality checks. It is so chosen that the distinction between variants and instances does not play a role.

EXAMPLE 3.16.
Let P = { p(a) ←. (C1),
p(y) ← p(z). (C2) },
let G_0 = ← p(x).

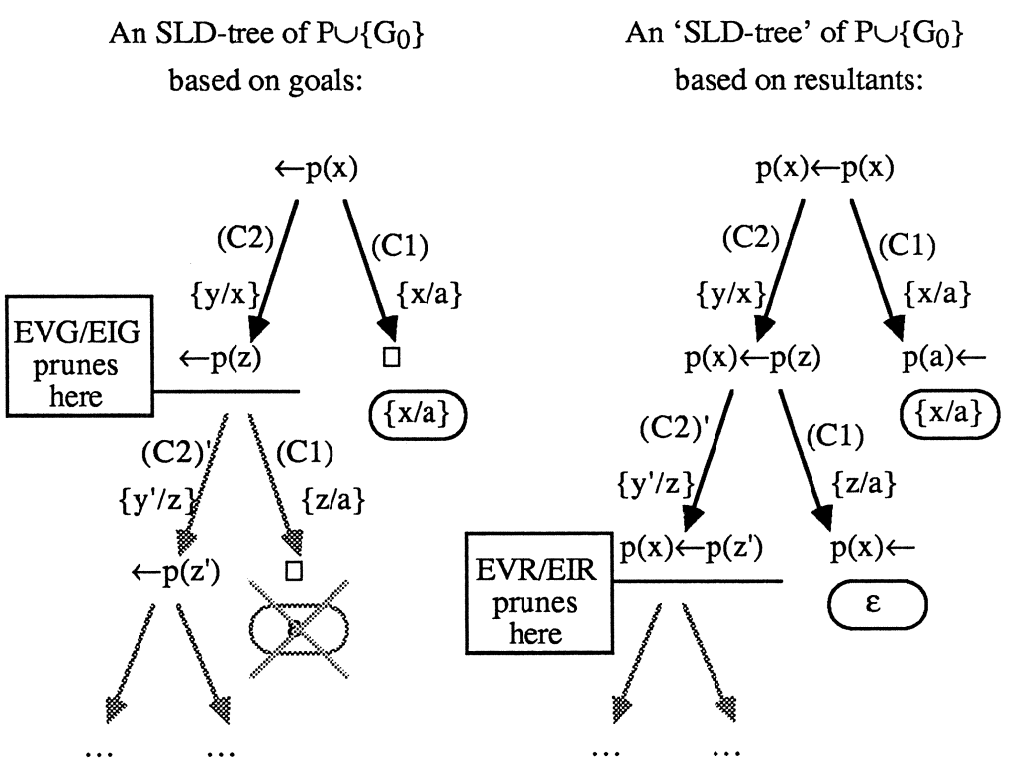

FIGURE 3.1

Without the condition $G_0\theta_1 \ldots \theta_k = G_0\theta_1 \ldots \theta_i\tau$ we would only obtain the computed answer substitution {x/a}, whereas we should also obtain the empty substitution. This shows that the EVG and EIG loop checks are not sound.

In the leftmost tree in Figure 3.1, ←p(z) is a variant of ←p(x), so the derivation is pruned by EVG at that goal. However, the corresponding resultant p(x)←p(z) is clearly not a variant of p(x)←p(x), therefore the derivation is not yet pruned by EVR. After another application of (C2), the resultant p(x)←p(z') occurs, which is a variant of p(x)←p(z). There the derivation is pruned by EVR.

The rightmost tree shows an 'SLD-tree' in which the goals are replaced by the corresponding resultants. Note that a successful branch in a resultant-based SLD-tree does not end by □, but by the computed answer of this branch. □

The following results are straightforward to prove.

LEMMA 3.17. *All equality checks are simple loop checks.*

The equality checks based on goals are stronger than the corresponding checks based on resultants.

The equality checks based on instances are stronger than the corresponding checks based on variants.

An informal motivation for the (weak) soundness of the equality checks has already been given. A formal proof of this result can be found in [BAK].

THEOREM 3.18 (Equality Soundness).

i) All equality checks based on resultants are shortening. A fortiori they are sound.

ii) All equality checks based on goals are weakly sound.

For completeness issues, it is sufficient to consider the weakest of the equality checks: the EVR_L check. We now introduce a class of programs for which EVR_L is complete. The necessary restriction is obtained by allowing at most one recursive call per clause and allowing such a call only after all other atoms in the body of the clause have been completely resolved. In order to avoid unnecessary complications, we shall place the atom that causes the recursive call (if present) at the right end of the body of the clause, and consider only

derivations via the leftmost selection rule. For a formal definition, we use the notion of the *dependency graph* D_P of a program P.

DEFINITION 3.19.
The *dependency graph* D_P of a program P is a directed graph whose nodes are the predicate symbols appearing in P and

$(p,q) \in D_P$ iff there is a clause in P using p in its head and q in its body.

D_P^* is the reflexive, transitive closure of D_P. When $(p,q) \in D_P^*$, we say that p *depends on* q in P. For a predicate symbol p, the *class of p* is the set of predicate symbols p 'mutually depends' on: $cl_P(p) = \{q \mid (p,q) \in D_P^* \text{ and } (q,p) \in D_P^*\}$. □

DEFINITION 3.20 (Restricted program).
Given an atom A, let *rel(A)* denote its predicate symbol. Let P be a program. In a clause $H \leftarrow A_1,\ldots,A_n$ $(n \geq 0)$ of P, an atom A_i $(1 \leq i \leq n)$ is called *recursive* if $rel(A_i)$ depends on rel(H) in P. Otherwise, the atom is called *non-recursive*.
A clause $H \leftarrow A_1,\ldots,A_n$ is *restricted w.r.t. P* if $A_1,\ldots,A_{n-1}$ are non-recursive.
A program P is called *restricted* if every clause in P is restricted w.r.t. P. □

Note that this definition allows at most one recursive call per clause. Thus (disregarding the order of atoms in the bodies) restricted programs include so called linear programs, which contain only one recursive clause and in this clause only a single recursive call occurs. The name *restricted program* originates from [ŠŠ], where essentially the same class of programs is defined and investigated, although a more rigid format is used. Now we can formulate the desired completeness result. For its proof, we refer again to [BAK].

THEOREM 3.21. *The loop check EVR_L is complete w.r.t. the leftmost selection rule for function-free restricted programs.*

COROLLARY 3.22 (Equality Completeness). *All equality checks are complete w.r.t. the leftmost selection rule for function-free restricted programs.*

PROOF. By Theorem 3.21 and the Relative Strength Theorem 3.13. □

4. THE USE OF LOOP CHECKING IN PARTIAL DEDUCTION

In this section the relation between partial deduction and loop checking is established. It appears that loop checks can be used in two different ways, each requiring special characteristics of the loop check.

Suppose a program P and a finite set of atoms **A** are given. For every atom $A \in \mathbf{A}$, a finite (unfinished) SLD-tree of $P \cup \{\leftarrow A\}$ must be constructed. When constructing these SLD-trees, two loop checks can be applied at the same time.

1. A sound, but not necessarily complete loop check is applied as in standard SLD-resolution. Goals that are pruned by this loop check are treated as failure leaves.
2. A complete, but not necessarily sound loop check is used for loop prevention (as it is called in [S2]). It ensures that the constructed tree is finite, thus enforcing termination of the partial deduction procedure. The resultants corresponding to the goals pruned by this loop check become part of the partial deduction for P w.r.t. **A**.

In order to avoid trivial SLD-trees, these loop checks must be *non-trivial*, i.e. it must not prune SLD-trees at their root. We now formalize this way of using loop checks in partial deduction and we prove that the soundness and completeness results of partial deduction persist.

DEFINITION 4.1 (Partial deduction with loop checking).
Let P be a program, A an atom and T a (completed) SLD-tree of $P \cup \{\leftarrow A\}$. Let L_S and L_C be two non-trivial loop checks such that L_C is complete. Let $G_1,\ldots,G_r$ be the leaves of $f_{L_C}(f_{L_S}(T))$[1] that are neither failed nor pruned by L_S. Let $R_1,\ldots,R_r$ be the corresponding resultants. The set $\{R_1,\ldots,R_r\}$ is called a *partial deduction for A in P w.r.t. L_S and L_C*.

For a set of atoms $\mathbf{A} = \{A_1,\ldots,A_s\}$, a *partial deduction for A in P w.r.t. L_S and L_C* is the union of partial deductions for $A_1,\ldots,A_s$ in P w.r.t. L_S and L_C.

A *partial deduction for P w.r.t. A, L_S and L_C* is a program obtained from P by replacing the set of clauses in P whose head contains one of the predicate symbols appearing in **A** by a partial deduction for **A** in P w.r.t. L_S and L_C. □

[1] This unfinished SLD-tree is obviously finite and non-trivial.

THEOREM 4.2. *Let P be a program, G a goal and A a finite set of atoms. Let L_S and L_C be two non-trivial loop checks such that L_C is complete. Let P' be a partial deduction for P w.r.t. A, L_S and L_C. Then*

i) P' is sound w.r.t. P and G.

ii) If P'∪{G} is A-closed and L_S is sound, then P' is complete w.r.t. P and G.

PROOF. i) The tree $f_{L_C}(f_{L_S}(T))$ in Definition 4.1 is precisely the finite non-trivial SLD-tree required in Definition 2.1. The only difference is that the resultants corresponding to the goals pruned by L_S are not included in P'. In other words, there exists a program $P'' \supseteq P'$ such that P'' is a partial deduction for P w.r.t. A. Thus, due to the absence of negation, a correct answer for P'∪{G} is also a correct answer for P''∪{G}, and hence by Theorem 2.4 also for P∪{G}.

ii) (This proof closely follows the proof of Theorem 4.1(b.i) in [LS]). Suppose that θ is a correct answer substitution for P∪{G}. Then there is an SLD-refutation D of P∪{G} giving a computed answer $G\tilde{}\sigma \leq G\tilde{}\theta$. We prove by induction on |D| that there is an SLD-refutation D* of P'∪{G} giving a computed answer $G\tilde{}\sigma^* \leq G\tilde{}\sigma$.

For |D| = 0, i.e. G = □, the claim is trivial. If the clause applied in the first step of D is (a variant of) a clause in P', then the induction step is also trivial. Otherwise the selected atom A in G must be an instance of an atom in A, because P∪{G} is A-closed; say $A_1 \in A$ and $A_1\gamma = A$. The steps in the refutation of P∪{G} in which A and its derived atoms are selected, constitute a refutation of P∪{←A}. Hence the completed version of the SLD-tree of P∪{←A_1} that was constructed during the partial deduction contains a successful branch B that uses the same steps (possibly in a different order). B gives a computed answer substitution τ such that $A_1\tau \leq A_1\gamma\sigma$. By the Switching Lemma (Lemma 4.6 in [LS]), the refutation steps of D can be reordered such that the new refutation D' begins with the steps proving A (more precisely: an instance of A more general than Aσ), in the order in which they occur in B.

Here two cases arise. If B is pruned by L_S, then the SLD-tree of P∪{←A_1} contains a branch B' that is not pruned by L_S and that gives a computed answer substitution τ' such that $A_1\tau' \leq A_1\tau$. This gives rise to yet another refutation (D'') of P∪{G}: the steps proving A according to refutation B can be replaced by steps proving A according to refutation B' (as $A_1\tau' \leq A\sigma$). If B is not pruned by

L_S then D'' = D', B' = B and $\tau' = \tau$. In both cases, the computed answer substitution σ'' of D'' satisfies $G\tilde{}\sigma'' \leq G\tilde{}\sigma$.

For some goal G_i on the branch B', the corresponding resultant R_i must be included in P'. Let H be the head of R_i. Then $H \leq A_1\tau' \leq A\sigma$, say $H\alpha = A\sigma$. As we may assume that D'' and H have no variables in common, it follows that $H\sigma\alpha = A\sigma\alpha$. Thus H and A unify, hence R_i can be used to resolve A, giving a resultant R'. By Lemma 4.12 of [LS], the SLD-derivation corresponding to B', starting from $\leftarrow A$ instead of $\leftarrow A_1$ yields R' in place of R_i. As, modulo a renaming and the presence of the rest of G, this derivation forms exactly the first i steps of D'', these steps can be replaced by the application of R_i, reaching the $(i+1)^{st}$ goal of D'' in one step; the resulting derivation still has σ'' as its computed answer substitution. If i = 1, then |B'| = 1 and R_i is a variant of the clause used in B'. Otherwise we can apply the induction hypothesis on this goal; the result is the refutation D^* of $P'\cup\{G\}$ with a computed answer substitution σ^* such that $G\tilde{}\sigma^* \leq G\tilde{}\sigma \leq G\tilde{}\theta$.

Thus θ is a correct answer substitution for $P'\cup\{G\}$. □

We now apply this part of the theory to the program given in Example 2.5. Especially the effect of the addition of a sound loop check is remarkable.

EXAMPLE 4.3.

Suppose that the SLD-tree of $P\cup\{\leftarrow$solve([tc(x,c)])$\}$ of Example 2.5 had not been finished at the resultant solve([tc(x,c)])$\leftarrow$solve([r(x,z)]),solve([tc(z,c)]), but continued as shown in Figure 4.1.

The resultant solve([tc(a,c)])$\leftarrow$solve([tc(a,c)]) could well be pruned by a sound loop check, e.g. EIR_L. The two other resultants could be pruned by some complete, but unsound loop check L_C (see Section 5).

Now the resulting partial deduction P_3 for P w.r.t. {solve([tc(x,c)])}, EIR_L and L_C contains the following clauses for 'solve':

```
solve([tc(b,c)]) ←,
solve([tc(a,c)]) ← solve([tc(b,c)]),
solve([tc(b,c)]) ← solve([tc(c,c)]).
```

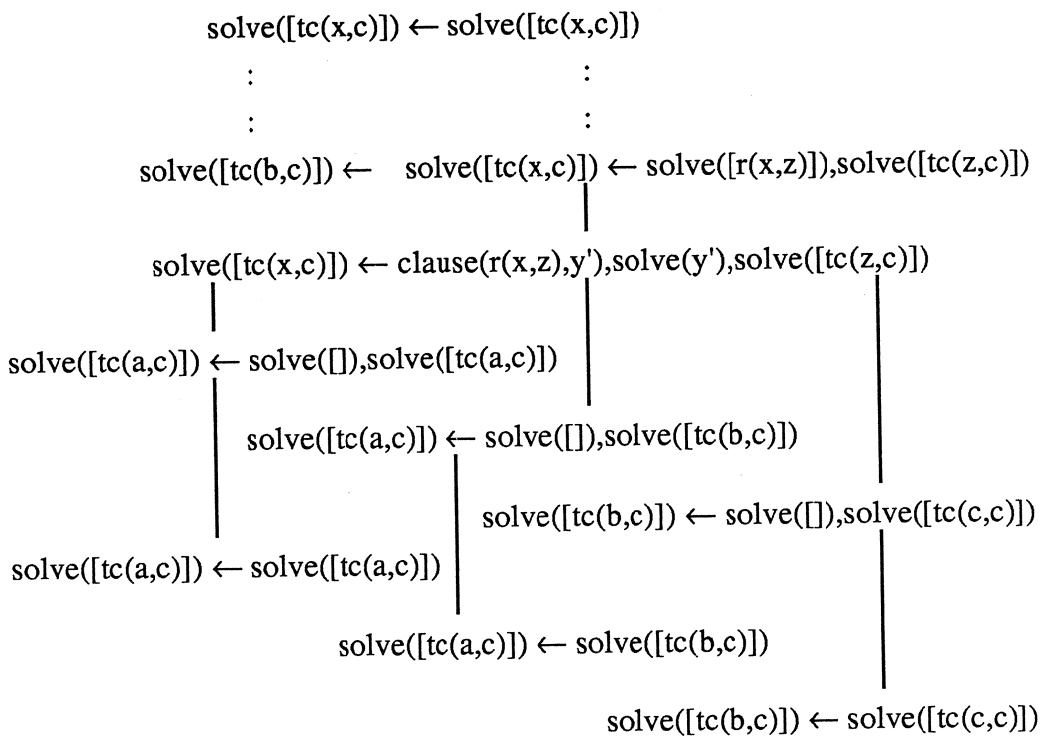

FIGURE 4.1

In contrast to Example 2.5, where **A** had to be extended, P_3 is already **A**-closed. So by Theorem 4.2, P_3 is complete for $P \cup \{\leftarrow solve([tc(x,c)])\}$. Moreover, whereas the SLD-trees of $P \cup \{\leftarrow solve([tc(x,c)])\}$ and $P_2 \cup \{\leftarrow solve([tc(x,c)])\}$ contain an infinite branch, the SLD-tree of $P_3 \cup \{\leftarrow solve([tc(x,c)])\}$ is finite. In this case the use of a sound loop check during partial deduction (making the clause solve([tc(a,c)])←solve([tc(a,c)]) disappear) can replace the use of a loop check at run-time. Obviously this will not always be the case. □

5. COMPLETE LOOP CHECKS

Most papers on loop checking ([Be, B1, BAK, BW, Co, vG, PG] consider the application of loop checks at run-time, on an SLD-tree generated by a PROLOG-like interpreter. Consequently, the soundness of a loop check is usually considered to be more important than its completeness. Only a few loop checks that are not weakly sound have been studied in some detail (e.g. in [BW]), and even those loop checks are mostly not complete.

So for the purpose of partial deduction, a sound loop check can be chosen from the literature. In this section we concentrate on the complete loop check needed. This loop check in general is not weakly sound. Our first observation concerns the relationship between complete loop checks and the selection rule.

5.1. Complete loop checks and the selection rule

Sound loop checks indicate that there is *certainly* a loop (or at least a redundant goal). If that is the case, then the derivation is best stopped immediately: the remainder of the derivation can succeed, giving a redundant answer, finitely fail or be infinite (depending on the selection rule), but in all cases there is no point in constructing it. This explains why such loop checks are normally independent of the atom selected in the current goal.

The complete, but generally unsound loop checks studied here indicate the *possibility* of a loop. Such a possibility is obviously related to the selection of the atom. Selecting another atom could be perfectly safe (i.e. not possibly loop). Moreover, this selection could remove the possibility of a loop, either by finitely failing or by instantiating the 'possibly dangerous' atom to a harmless instance.

Thus it is worthwhile to use a loop check that prunes only if it finds that the *selected* atom is 'dangerous', and to adopt a selection rule that avoids pruning (selecting a 'dangerous' atom) as long as possible. (In the same way, floundering is avoided in the presence of negation by the use of a *safe* selection rule.) In [BL] partial selection rules are used that do not select 'dangerous' atoms at all: by stating that 'the computation terminates in deadlock when no literal is available for selection', the loop check is described implicitly by the partiality of the selection rule.

Four of these selection rules are given; they are all of the same form: an atom A is 'dangerous' if it is produced by an atom A' higher up in the derivation such that

1) A and A' are variants ($A \leq A'$ and $A' \leq A$)
2) A is an instance of A' ($A' \leq A$)
3) A' is an instance of A ($A \leq A'$)
4) A and A' have a common instance (for some B: $B \leq A$ and $B \leq A'$).

Loop check 4) is obviously stronger than 2) and 3), which are in turn stronger than 1). Unfortunately, none of these loop checks is complete, a simple counterexample being the program $\{p(x) \leftarrow p(f(x))\}$ and the goal $\leftarrow p(a)$.

The simplest complete loop check is without doubt the use of a depth-bound on derivations ($L = \{D \mid |D| = d\}$ for some $d \geq 1$). But such a loop check is not very useful for partial deduction purposes. In order to obtain a partial deduction for P w.r.t. **A** that is **A**-closed, every atom occurring in a pruned goal must be an instance of an atom in **A**. Thus pruning goals regardless of their structure usually results in an 'explosion' of the set **A**.

5.2. The OverSizeCheck

More sophisticated loop checking mechanisms are discussed in [S1]. The following definition gives their general framework, leaving two parameters open: a *depth-bound* and a *size*-function on atoms. Roughly speaking, the loop check prohibits the selection of 'oversized' atoms. An atom is 'oversized' if it is 'produced' by at least *depth-bound* earlier selected atoms with the same predicate symbol that have a smaller or equal *size*. Let #S denote the number of elements of a set S and rel(A) the predicate symbol in an atom A.

DEFINITION 5.1 (OverSizeCheck).
Let $d \geq 1$ and let the function *size* be defined for all atoms (details on *size* follow later). The *OverSizeCheck* of d and size, *OSC*(d,size) =
Initials($\{G_0 \Rightarrow G_1 \Rightarrow \ldots \Rightarrow G_k \mid$ for $0 \leq i \leq k$, A_i is selected in G_i and
$\#\{i \mid 0 \leq i < k$, $rel(A_i) = rel(A_k)$, A_k is the result of resolving A_i and $size(A_i) \leq size(A_k)\} \geq d\}$). □

Notice that $d \geq 1$ ensures that OSC is a non-trivial loop check. The following remark follows immediately from the definitions.

REMARK 5.2.
For every function size, if $1 \leq d_1 \leq d_2$ then $OSC(d_1,size)$ is stronger than $OSC(d_2,size)$. For every $d \geq 1$, if for all atoms A and B: $size_1(A) \leq size_1(B)$ implies $size_2(A) \leq size_2(B)$, then $OSC(d,size_2)$ is stronger than $OSC(d,size_1)$. □

The size of an atom is usually just a natural number. This is the case for the versions 1 and 2 of OSC in [S1]. In version 1, the *size*-part of the condition is completely absent (equivalently, for all atoms A: size(A) = 0). Thus for every

predicate symbol only d atoms may be selected. By Remark 5.2, this is for a given value of d the strongest possible version of OSC.

In version 2, size(A) is the total number of variable, constant and function symbol occurrences in A. Example 5.7 shows an application of these versions. We now prove that OSC is complete if *size* returns natural numbers.

THEOREM 5.3. *Let $d \geq 1$ and let for every atom A, size(A) $\in \mathbb{N}$. Then OSC(d,size) is complete.*

PROOF. Suppose that $D = (G_0 \Rightarrow G_1 \Rightarrow \ldots)$ is an infinite SLD-derivation. Since D is infinite, at least one atom in G_0 has infinitely many selected descendants, hence the proof tree of this atom is infinite. Applying König's Lemma on this proof tree shows that it has an infinite branch, so there exists an infinite sequence of goals $G_{m_0}, G_{m_1}, \ldots$ $(0 \leq m_0 < m_1 < \ldots)$ containing atoms $A_0, A_1, \ldots$ such that for every $i \geq 0$:

- A_i is the selected atom in G_{m_i},
- A_{i+1} is (the further instantiated version of) an atom A_{i+1}', which is introduced in G_{m_i+1} as the result of resolving A_i.

The situation is depicted in Figure 5.1, selected atoms are underlined.

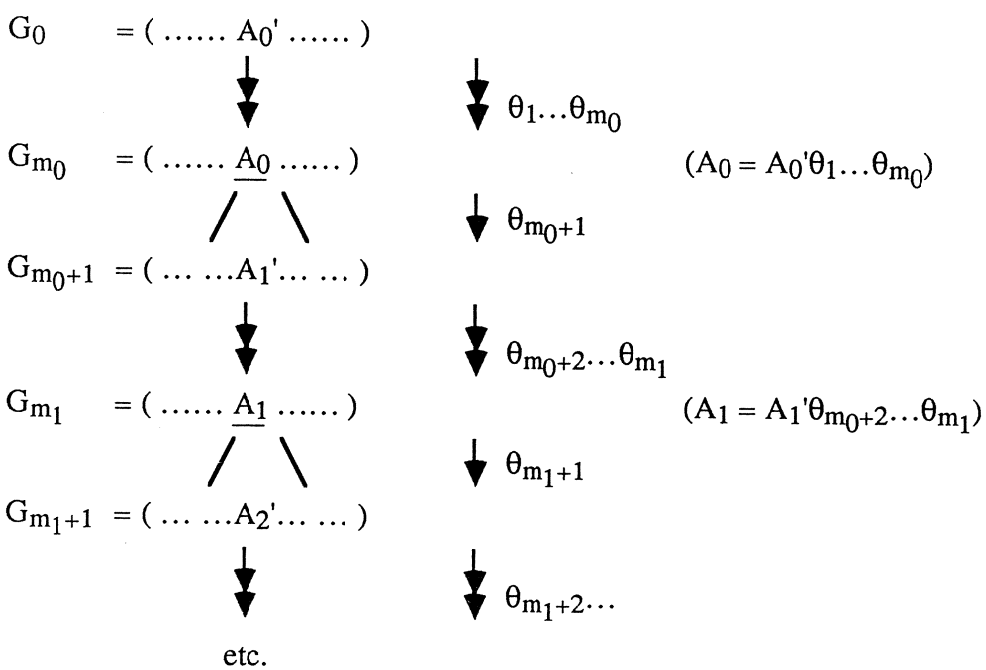

FIGURE 5.1

As we have only a finite number of predicate symbols, at least one predicate symbol p occurs in infinitely many atoms A_i. Let $I = \{i \mid rel(A_i) = p\}$ and let $i_1,\ldots,i_d$ be the smallest d members of I. Let $k = \max\{size(A_{i_j}) \mid 1 \leq j \leq d\}$. Two cases arise.

1. For some $n \in I$: $size(A_n) > k$. Then OSC(d,size) prunes D at G_{m_n} (or earlier).
2. For all $n \in I$: $size(A_n) \leq k$. Then in the worst case $(A_i)_{i\in I}$ consists of d atoms of size k, then d of size k–1, …, then d of size 1, then d of size 0. That makes (k+1)d atoms. So OSC(d,size) prunes D at the goal in which the $(k+1)d+1^{th}$ atom of $(A_i)_{i\in I}$ is selected (or earlier). □

In some cases a more complex size-function is convenient. We show that instead of the natural numbers, any well-quasi-ordered set can be used. (For a survey on well-quasi-ordered sets see [Kr]. They are frequently used in termination proofs for term rewriting systems, see e.g. [DJ].)

DEFINITION 5.4.
A set U is *well-quasi-ordered* under a quasi-ordering ≥ if every infinite sequence $u_1,u_2,\ldots$ of elements of U contains a pair u_j and u_k such that $j < k$ and $u_j \leq u_k$. □

The following lemma is a special case of a result well-known from the literature. For completeness sake we repeat the argument here, following [DJ].

LEMMA 5.5. *Let U be a well-quasi-ordered set under ≥ and let $n \geq 2$ be a natural number. Then every infinite sequence $u_1,u_2,\ldots$ of elements of U contains a sub-sequence $u_{i_1},\ldots,u_{i_n}$ such that $u_{i_1} \leq u_{i_2} \leq \ldots \leq u_{i_n}$.*

PROOF. By induction on n.
For n = 2, the claim corresponds to the definition of a well-quasi-ordered set. Assume that the claim holds for a certain value of n. Then we can define a function *row* such that for every infinite sequence $S = (u_i)_{i\in I}$ of elements of U, $row(S) = (u_{i_1},\ldots,u_{i_n})$ is a sub-sequence of S such that $u_{i_1}\leq\ldots\leq u_{i_n}$. Let end(row(S)) denote i_n.

Let $u_1,u_2,\ldots$ be an infinite sequence of elements of U. The required sub-sequence of length n+1 is constructed as follows.

Define inductively $j_0 = 0$ and for $k > 0$, $j_k = \text{end}(\text{row}((u_i)_{i>j_{k-1}}))$. Consider the infinite sequence $(u_{j_k})_{k>0}$. As U is well-quasi-ordered there exist p and q such that $p < q$ and $u_{j_p} \leq u_{j_q}$. The sequence $\text{row}((u_i)_{i>j_{p-1}})$ is an increasing sequence of length n that ends in u_{j_p}. Adding u_{j_q} to this sequence yields the required increasing sequence of length n+1. □

THEOREM 5.6. *Let $d \geq 1$ and let U be a well-quasi-ordered set. If for every atom A, size(A) $\in$ U, then OSC(d,size) is complete.*

PROOF. Suppose that $D = (G_0 \Rightarrow G_1 \Rightarrow \ldots)$ is an infinite SLD-derivation. Let I be defined as in Theorem 5.3. By Lemma 5.5 the sequence $(\text{size}(A_i))_{i \in I}$ contains an increasing sequence of length d+1. Let $A_{n_1},\ldots,A_{n_{d+1}}$ be the sequence of corresponding atoms. Then OSC(d,size) prunes D at the goal in which $A_{n_{d+1}}$ is selected. □

Version 3 of OSC in [S1] can serve as an example. There

$U = \{(p,\underline{n}) \mid p \text{ is a predicate symbol with arity k and } \underline{n} \in \mathbb{N}^k\}$ and

$(p,\underline{n}) \leq (q,\underline{m})$ if $p = q$ and $\underline{n} \leq \underline{m}$ lexicographically.

It is easy to see that for a language with finitely many predicate symbols, U is indeed well-quasi-ordered under $\geq$. Defining termsize as size was defined in version 2, the size of an atom $A = p(t_1,\ldots t_k)$ is defined as

$$\text{size}(A) = (p,<\text{termsize}(t_1),\ldots,\text{termsize}(t_k)>).$$

EXAMPLE 5.7.

This example shows the application of the three versions of OSC mentioned above. Throughout this example the depth-bound used is 1 (a poor choice in practice, but it serves to keep the example small). Consider the following variation of the reverse program that reverses a list of natural numbers (formed by the constant 0 and the successor-function s), but leaves out the 0's in the reversed list.

```
P = { reverse(          [ ], x, x)  ←.                          (C1),
      reverse(      [0 | x], y, z)  ← reverse(x, y, z).         (C2),
      reverse( [s(w) | x], y, z)  ← reverse(x, [s(w) | y], z).  (C3) }.
```

Figure 5.2 shows where the three versions of OSC prune the SLD-tree of P∪{←reverse([0,s(0),s(s(0)) | x], [ ], y).

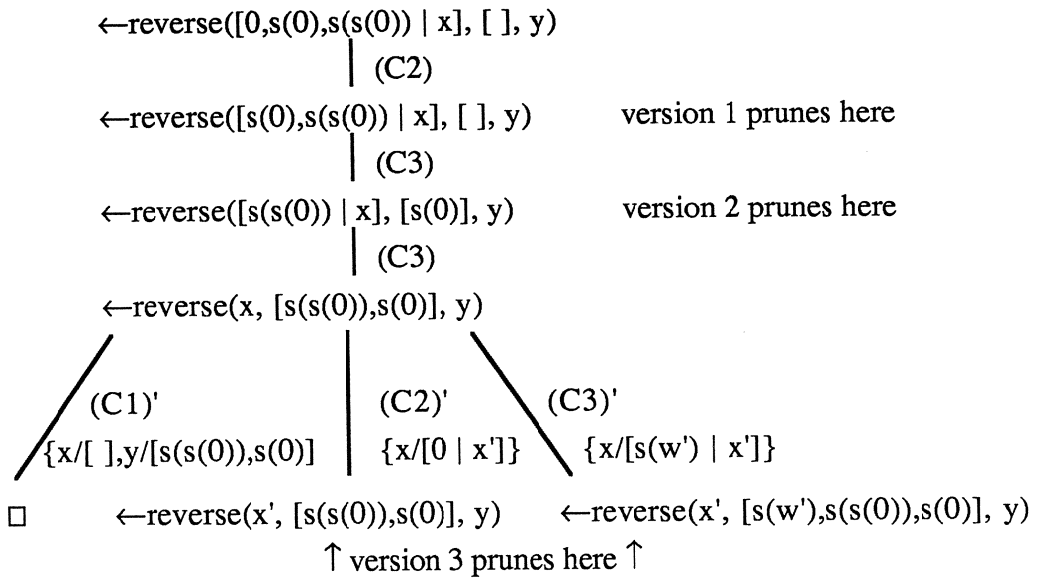

FIGURE 5.2

According to version 1, the predicate 'reverse' may be selected only once. Thus it prunes the second goal. Version 2 does not prune the second goal, because its size is strictly smaller than that of the initial goal. But the second and third goal have the same size, so version 2 prunes the third goal. Version 3 uses a different size-function. According to this function the third goal is smaller than the second, because its first argument is smaller. So version 3 does not prune until the given part of the list has been completely processed: after that the first argument cannot shrink any more and as the second argument stays the same or grows, version 3 prunes there. □

The formulation of version 3 of OSC shows that it is always possible to incorporate the predicate symbol of an atom A in size(A) and to make elements with different predicate symbols incomparable. In this sense the requirement $rel(A_i) = rel(A_k)$ in Definition 5.1 is superfluous. But normally it serves well to simplify the definitions of U, ≤ and size. Moreover it highlights that the OverSizeCheck takes the structure of the current goal into account, the feature that was missing in the simple depth-bound check.

The question which depth-bound and size-function are optimal shall remain unanswered here. It is not even clear how to compare different choices, let alone how to identify the optimal choice. The above framework for OSC allows for a wide range of complete loop checks, from very simple to very complex. But as is noted in both [BL] and [S1, S2], in practice a complex loop check is not necessarily better than a simpler one. An explanation for this phenomenon is that even if the partial deduction process is not in a loop, the result of stopping it at a certain point can be better than the result of stopping it later.

5.3. A related work

A closely related approach is pursued in [BdSM]. First they give the following characterization of finite (unfinished) SLD-trees, using well-founded sets.

DEFINITION 5.8.

Given a completed SLD-tree T, we associate to each node (goal) G of T a natural number (this number is needed to distinguish different occurrences of the same goal). The set of *goal-occurrences* in T is $G_T = \{(G,i) \mid G \text{ is a goal of T and i is its associated number}\}$. If the goal occurrence (G,i) is an ancestor of (G',j) in T then we write $(G,i) >_T (G',j)$. □

DEFINITION 5.9.

A strict partially ordered set $U,>_U$ is *well-founded* if there is no infinite sequence $u_1,u_2,\ldots$ of elements of U such that $u_j >_U u_{j+1}$ for all $j \geq 1$.

A *well-founded measure* on a strict partially ordered set $S,>_S$ is a monotonic function from $S,>_S$ to a well-founded strict partially ordered set $U,>_U$.

An SLD-tree T is *well-founded* if there exists a well-founded measure on $G_T,>_T$. □

THEOREM 5.10 ([BdSM]). *An SLD-tree T is finite iff T is well-founded.* □

This theorem can be used as follows. Given an SLD-tree T, we fix a well-founded set $U,>_U$ and a function f from G_T to U. We obtain a finite pruned version T' of T by pruning each node (G,i) in T unless $f(G',j) >_U f(G,i)$, where (G',j) is the parent of (G,i) in T. T' itself is not well-founded w.r.t. $U,>_U$, but

removing the leaves from T' yields a well-founded tree w.r.t. $U,>_U$. By Theorem 5.10 this tree is finite, and hence T' is finite.

The only-if part of Theorem 5.10 implies that for each finite initial subtree T' of T, we can find suitable $U,>_U$ and f. Thus this method cannot help us by allowing only 'good' nodes to be pruned.

We now compare this method with OSC(1,size). First of all, this method is *not* a loop check: it allows us to prune two derivations that are variants of each other and that occur both in the complete tree at different places. This is caused by an important difference between the functions f and size: where size takes only the selected atom as input, f takes the whole goal *and its associated number.*

A more technical difference is the use of well-quasi-ordered sets for OSC and well-founded sets here. Well-quasi-ordered sets seem to be more limited, as they allow only a finite number of incomparable elements. But they allow that distinct elements a and b are equivalent, i.e. $a \leq b$ and $b \leq a$. One must realize that a derivation step $G \Rightarrow H$ here requires a strict decrease: '$H < G$', whereas OSC prohibits increase: 'not $H \geq G$'. Thus when G and H are incomparable, they are pruned by this method, but not by OSC; the treatment of incomparable elements here is the same as the treatment of equivalent elements by OSC.

In order to make it easier for the user to specify which nodes are to be pruned, at the same time providing more guidance to the user as to where pruning could give 'good' results, a more complex characterization of finite SLD-trees is provided. It allows us to divide nodes in a finite number of classes, and to compare two nodes only if they are in the same class. In practice, the class of a node is often based on the predicate symbol of the selected atom in it. However, the theory does not require this. In OSC, this practice is 'built-in' through the requirement '$rel(A_i) = rel(A_k)$'. The measure associated to a class is usually some kind of term-size of the selected atom, like in OSC.

A special class (C_0) is added for those goals of which the user knows that they terminate or yield a goal in another class without pruning (typically goals of which the selected atom has a non-recursive predicate symbol, and the empty goal). They are not compared to any other goal.

DEFINITION 5.11.
An SLD-tree T is *subset-wise founded* if there exists a finite number of sets $C_0,\ldots,C_N$ such that
i) $G_T = \cup\{C_k \mid 0 \leq k \leq N\}$,
ii) for each $i = 1,\ldots,N$, $C_i,>_T$ has a well-founded measure f_i, and
iii) for each branch D of T and for each non-leaf $(G,i) \in C_0$ therein, there exists a node (G',j) in D such that $(G,i) >_T (G',j)$ and
- either $(G',j) \in C_k$ for some $k > 0$,
- or (G',j) is a leaf in T. □

Notice that $C_0,\ldots,C_N$ need not be a partition of G_T. Condition iii) ensures that goals in C_0 indeed terminate or lead to a goal in another class. This definition is still general enough to allow the following theorem.

THEOREM 5.12 ([BdSM]). *An SLD-tree T is finite iff T is subset-wise founded.* □

Thus any complete loop check can still be described as an instance of this method. A more interesting question is whether it can be done in a 'natural' way. For example, it is suggested in [BdSM] to formulate the use of a combination of a criterion C(G) (e.g. one of the criteria suggested in [BL]) and a simple depth-bound d by using a single class with the measure

$$f(G,i) = \begin{cases} d & \text{if } d_T(G) \geq d \text{ or } C(G) \\ d - d_T(G) & \text{otherwise} \end{cases}$$, where $d_T(G)$ is the depth of G in T.

One could argue that this measure is not 'natural', because it depends on the location of a goal in the tree.

Finally an even more complicated method is introduced in [BdSM], which we shall not discuss here in detail. The aim of this method is to facilitate the incorporation of a condition like 'A_k is the result of resolving A_i' in OSC. This condition is important: otherwise the partial deduction for a goal $\leftarrow q(\ldots)$ producing a goal $\leftarrow p(\ldots),p(\ldots)$ might be stopped when the second p-atom is selected, because it is 'similar' to the previously selected first p-atom. The definition is still general enough to define all pruned trees.

In my opinion this method is only of practical interest for 'natural' choices of $C_0,\ldots,C_N$ and $f_1,\ldots,f_N$. Although the choice of a depth-bound as used in

OSC will always remain arbitrary, it could be worthwhile to integrate the possibility of a depth-bound in this method as well. This could be done easily by allowing a derivation to 'disobey' the required monotonicity a (fixed) finite number of times, as is done in OSC.

In its full generality this method is too strong for practical purposes, but it might be of theoretical interest. A given loop check can always be seen as an instance of this method, but then the interesting question is how 'natural' this instance is. The answer to this question might be more informative than the answer to the question whether a given loop check can be seen as an instance of OSC, which is simply 'yes' or 'no'.

Finally, the method of [BdSM] can be automated. When this has been done, implementing an instance of this method requires only that $C_0,\ldots,C_N$ and $f_1,\ldots,f_N$ are typed in. For a 'natural' instance, this should take little effort.

6. CONCLUSIONS

Summarizing, we have the following results:

- Loop prevention methods for partial deduction can be formulated within the framework of loop checking presented in [BAK].

- However, loop prevention requires a complete, probably unsound loop check, whereas the use of a loop check at run-time requires a sound, probably incomplete loop check. This explains why loop checks proposed in the literature for use at run-time are not suitable for loop prevention.

- Nevertheless, sound loop checks can be added in a useful way to the partial deduction scheme, as outlined in Section 4. This can result in the removal of loops from the generated program.

- Further research on complete loop checks is required. In this respect, it is important that using the most selective (weakest) complete loop check not necessarily leads to the best possible generated program.

- The completeness of a loop check can be proved by showing that it is an instance of the framework presented in [BdSM]. Once the method based on this framework is automated, 'natural' instances of it can be implemented easily.

ACKNOWLEDGMENT

This paper benefitted from discussions with Krzysztof Apt, Kerima Benkerimi, Dan Sahlin and Jan Komorowski.

REFERENCES

[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. To appear in Theoretical Computer Science 85, 1991.

[B1] R.N. BOL, *Towards More Efficient Loop Checks*, in: Proceedings of the 1990 North American Conference on Logic Programming (S. Debray and M. Hermenegildo eds.), MIT Press, Cambridge Massachusetts, 1990, 465-479.

[B2] R.N. BOL, *Loop Checking and Negation*, Technical Report CS-R9075, Centre for Mathematics and Computer Science, Amsterdam, 1990. Extended abstract in: Logics in AI (J. van Eijck ed.), LNCS 478, Springer-Verlag, Berlin, 1991, 121-138.

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

[BEJ] D. BJØRNER, A.P. ERSHOV and N.D. JONES eds., *Workshop on Partial Evaluation and Mixed Computation*, Gammel Avernæs, Denmark, Oct. 1987.

[BL] K. BENKERIMI and J.W. LLOYD, *A Partial Evaluation Procedure for Logic Programs*, in: Proceedings of the 1990 North American Conference on Logic Programming (S. Debray and M. Hermenegildo eds.), MIT Press, Cambridge Massachusetts, 1990, 343-358.

[BdSM] M. BRUYNOOGHE, D. DE SCHREYE and B. MARTENS, *A General Criterion for Avoiding Infinite Unfolding during Partial Evaluation*, to appear in: Proceedings of the 1991 International Logic Programming Symposium, MIT Press, Cambridge Massachusetts, 1991.

[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.

[Cl] K.L. CLARK, *Negation as Failure*, in: Logic and Data Bases (H. Gallaire and J. Minker eds.), Plenum Press, New York, 1978, 293-322.

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

[DJ] N. DERSHOWITZ and J.-P. JOUANNAUD, *Rewrite Systems*, in: Handbook of Theoretical Computer Science (J. van Leeuwen ed.), vol. B, North Holland, 1990, 243-320.

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

[Ko] H.J. KOMOROWSKI, *A Specification of an Abstract Prolog Machine and its Application to Partial Evaluation*, Technical Report LSST 69, Linköping University, 1981.

[Kr] J.B. KRUSKAL, *Well-Quasi-Ordering, the Tree Theorem, and Vazsonyi's Conjecture*, Transactions of the AMS 95, 1960, 210-225.

[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. Revised 1989.

[P] T.C. PRZYMUSINSKI, *On the Declarative and Procedural Semantics of Logic Programs*, J. Automated Reasoning 5, 1989, 167-205.

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

[Re] R. REITER, *On Closed World Data Bases*, in: Logic and Data Bases (H. Gallaire and J. Minker eds.), Plenum Press, New York, 1978, 55-76.

[S1] D. SAHLIN,*The Mixtus Approach to Automatic Partial Evaluation of Full Prolog*, in: Proceedings of the 1990 North American Conference on Logic Programming (S. Debray and M. Hermenegildo eds.), MIT Press, Cambridge Massachusetts, 1990, 377-398.

[S2] D. SAHLIN, *An Automatic Partial Evaluator for Full Prolog*, Ph.D.thesis, Swedish Institute of Computer Science, 1991.

[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.

[TS] H. TAMAKI and T. SATO, *OLD Resolution with Tabulation*, in: Proceedings of the Third International Conference on Logic Programming (G. Goos and J. Hartmanis eds.), LNCS 225, Springer Verlag, Berlin, 1986, 84-98.

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