CWI Tracts

CWI
P.O. Box 94079, 1090 GB Amsterdam, The Netherlands
Telephone 31 - 20 592 9333, telex 12571 (mactr nl),
telefax 31 - 20 592 4199

CWI is the nationally funded Dutch institute for research in Mathematics and Computer Science.

Loop checking in logic programming

R.N. Bol

1991 Mathematics Subject Classification: 68N17.
ISBN 90 6196 456 3
NUGI-code: 811

Acknowledgements

This book has been written at the Centre for Mathematics and Computer Science, where I was employed from 1987 to 1991 in the Department of Software Technology led by Jaco de Bakker.

First of all I wish to express my gratitude to Krzysztof Apt, my supervisor, for the many things he has done for me. He suggested me to investigate the subject of this book, and was always ready to advise me on technical and organizational matters. He also gave me several opportunities to travel abroad.

I also thank Jan Willem Klop for teaching me the foundations of logic programming, for his continuous and friendly support and for four years of pleasant cooperation in giving a course at the Free University of Amsterdam. I thank Johan van Benthem, John Shepherdson, Jan Bergstra, Peter van Emde Boas, Paul Klint and Jan Willem Klop for their comments on drafts of this book.

Finally I thank my colleagues, friends and relatives for all they have done for me. But most of all I thank my parents and grandparents for supporting me in so many different ways.

Amsterdam, April 1991

In this edition, I corrected some spelling errors and I added a few references to related work. The most notable changes are the addition of a discussion on constructive negation in Chapter 5 and a more thorough analysis of nontermination of the partial deduction procedure (due to not reaching the closedness condition) in Chapter 6.

I thank the Eindhoven University of Technology and in particular Jos Baeten for allowing me to spend time on this book.

Eindhoven, January 1994

Contents

Introduction

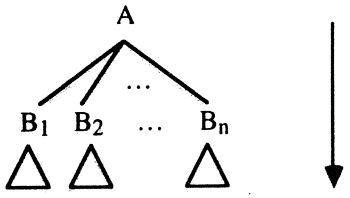
Logic programming

The first three generations of programming languages are imperative languages: a program consists of a list of instructions, telling the computer what to do. The exactness of the instructions can differ (for example storage allocation is nowadays performed by the operating system and not by the programmer), but the BASIC idea is the same. The fourth generation is already quite different: it is based on the composition of functions. This book is about the fifth generation of programming languages, which are based on logic: *logic programming*.

Programming languages are used to encode algorithms. An algorithm is a recipe for solving a problem. Kowalski's [Ko] famous equation separates the two aspects of an algorithm: 'Algorithm = Logic + Control'. The logical part of an algorithm states the problem and defines its solutions. The control part describes how these solutions are to be found. Imperative programs have very explicit control structures, but it is completely up to the programmer to provide them with logic. That is why they can be so unreadable: the programs states *what* the computer must do, but not *why*.

For logic programs the converse is true. They consist of a set of implications in first-order logic. Thus their logical, declarative meaning is explicit and well-understood. But pure logic programs say a priori nothing about control. Thus there must be a layer between the logic program and the computer that adds the control. From where does this layer get its information? First of all from the one who designs it. He decides how the logical rules of a program are to be interpreted operationally.

Given an implication or *clause* $A \leftarrow B_1,\ldots,B_n$, its logical meaning (declarative interpretation) is: if B_1 is true and ... and B_n is true then A is true. An obvious procedural interpretation can be derived from this: if the user requests a proof of A, try to prove B_1 and ... and B_n. This very popular control mechanism is known as *top-down* interpretation, because it corresponds to the top-down construction of a proof tree for A.



Top-down

The next question is in what order $B_1,\ldots,B_n$ should be proved. Although proving them in parallel has its advantages, it is usually done one-at-a-time (mainly because this is more efficient on sequential machines): first B_1 is proved, then B_2 and so on. This seems natural, but it has an important consequence: this interpretation assumes that some control information is implicitly present in the logic program, encoded by the order of $B_1,\ldots,B_n$. Thus the addition of control is still partly the responsibility of the programmer. More sophisticated methods for obtaining and using control information from the programmer are described in [BdSK].

A definition of logic programs and their logical meaning is given in Chapter 1. This chapter also formalizes the process called *SLD-resolution*: the construction of an *SLD-tree*, the search space of a top-down interpreter for logic programs. It recalls an important result: the soundness and strong completeness of SLD-resolution (Theorem 1.3.2). Soundness generally means that there are no 'undesirable results': in this case it means that every solution that is present in an SLD-tree is correct w.r.t. the logical meaning of the program. Completeness means that every 'desirable result' is achieved: every correct solution is present in every SLD-tree.

The most wide-spread logic programming language, PROLOG, allows the programmer to add much more and explicit control information to his programs. Unfortunately, programmers often over-use these *extra-logical features* of PROLOG, especially the cut. The result is 'imperative PROLOG': a program that consists mainly of control information and that has no declarative meaning. Additional control information is often provided to improve the efficiency of a logic program. The most extreme form of inefficient behaviour of a program is nontermination.

Nontermination

Although every solution is *present* in an SLD-tree, it is not guaranteed that it is also *found* by an interpreter. When a logic program is interpreted by a PROLOG-like interpreter, the result is often a nonterminating computation. This does not mean that the program is logically incorrect. It is caused by the fact that the interpreter employs a depth-first search through the SLD-tree. Consequently it can enter an infinite branch and miss a solution.

The problem of detecting such a possibility of nontermination is generally undecidable as logic programming has the full power of recursion theory. Programmers have developed a number of useful heuristics to enforce termination. Sometimes it suffices to give a more complex set of logical rules. However, the resulting program can be very different from the original one. More often than not, the programmer decides to add explicit control primitives to the program, thereby destroying its declarative meaning. In both cases the burden of avoiding nontermination rests with the programmer.

Another possible approach to this problem is based on modifying the interpreter that searches through the SLD-tree by adding a capability of pruning. Pruning an SLD-tree means that at some point the interpreter is forced to stop its search through a certain part of the tree, typically an infinite branch. Every method of pruning SLD-trees considered so far has been based on excluding some kind of repetition in the SLD-derivations, because such a repetition can make the interpreter enter an infinite loop. That is why pruning SLD-trees has been called *loop checking*.

Example

To better understand the relevance of the problems studied here, consider the following example. Let P be the following simple-minded logic program computing in the relation *tc* the transitive closure of the relation *r*:

P = { tc(x,y) ← r(x,y).
tc(x,y) ← r(x,z),tc(z,y). }.

Suppose we add to P the following facts about *r*: r(a,a)←, r(a,b)←, r(b,c)←, r(d,a)←. Then we can interpret P as a PROLOG program, but if we ask:

- tc(a,b) we get the answer 'yes';
- tc(a,c) the program gets into an infinite loop
  (whereas we should get the answer 'yes');
- tc(a,d) the program gets into an infinite loop
  (whereas we should get the answer 'no');
- tc(b,d) we get the answer 'no'.

Thus although *logically* P is the right program for computing the transitive closure of *r*, *operationally* it is not. One solution is to write a different program, which is not straightforward – see for example the program in [CM, Section 7.2]. In fact, Kunen [Ku2] proved that any such program must use either function symbols or negation. In our solution we retain the above program and we change the underlying interpreter by adding a loop checking mechanism to it.

Loop checking

A loop check is a mechanism that prunes SLD-trees. A formal definition is given in Section 2.1. Although this definition imposes some restrictions, it is still fairly general. Thus the question arises: 'What is a *good* loop check'. In Section 2.2 we introduce again notions of soundness and completeness, but now for loop checks.

An undesirable result of the application of a loop check would be the loss of solutions. Thus we call a loop check *sound* if it does not prune an SLD-tree to such an extent that solutions are lost. We call it *weakly sound* if its application results in the loss of *some* solutions, but not in the loss of *all* solutions. The purpose of a loop check is to reduce the search space. We would like to end up with a finite search space. If a loop checks achieves this result then it is *complete*.

Due to the undecidability of the Halting Problem, a loop check cannot be (weakly) sound and complete for all programs. In most cases we consider the soundness of a loop check to be more important than its completeness (except in Chapter 6). Therefore we shall usually investigate (weakly) sound loop checks, and identify classes of programs for which they are complete.

It is important to notice that not every derivation that is pruned by a sound loop check is actually in an infinite loop. Most of the loop checks we shall consider prune a derivation as soon as some kind of repetition occurs that makes

the resulting goal redundant (i.e., not producing any new answers). Whether or not this repetition gives rise to an infinite loop is irrelevant for them. For example, consider the program

P = { q ← p(x).
p(a) ← p(b).
p(b) ← p(c). }.

The SLD-derivation ←q ⇒ ←p(x) $\Rightarrow_{\{x/a\}}$ ←p(b) ⇒ ←p(c) fails finitely, but it is pruned at ←p(b) by many loop checks.

Applications

From these considerations we obtain an implementation of the *closed world assumption* of Reiter [Re] and of a query evaluation mechanism for various classes of definite deductive databases. The closed world assumption (CWA in short) is a way of inferring negative information in deductive databases. Reiter [Re] showed that in the case of definite deductive databases (DB in short) it does not introduce inconsistency. However, even though CWA is correctly defined for DB, there is still the problem of how it can be implemented, since it calls for the use of the following rule (or rather metarule):

$$\text{if DB} \nvdash \varphi \text{ then DB} \vdash \neg\varphi,$$

that is: deduce $\neg\varphi$ if φ cannot be proved from DB using first order logic.

The problem is how to determine for a particular ground atom A that there is no proof for it. A loop check that is weakly sound and complete for DB solves this problem as follows. A logic programming interpreter augmented with this loop check tries to prove A from DB. When this attempt fails, we can infer $\neg$A using Clark's [Cl2] *negation as (finite) failure* rule, the operational counterpart of CWA. The weak soundness of the loop check implies that if no proof is found for A, then there is indeed no proof for A. The completeness is needed to ensure the termination of the procedure.

A more general problem is that of query processing in DB: given an atom A, compute the set $[A]_{DB}$ of all its ground instances Aθ such that DB ⊢ Aθ. In other words: compute all answers to the query A. Indeed, when A is ground and DB ⊬ A, the query processing problem reduces to the problem of deducing $\neg$A by means of CWA. Here we need a loop check that is sound and complete for

DB to solve the problem: to compute $[A]_{DB}$ for an atom A, it suffices to collect all computed answer substitutions in the SLD-tree starting from A, pruned by this loop check. Again, the soundness of the loop check ensures that all solutions are found and the completeness ensures that the procedure terminates. These applications of loop checking are formalized in Section 2.2.

An alternative application of loop checking is outlined in Chapter 6, where loop checking is incorporated in the framework of *partial deduction* (following [LS], where it is called *partial evaluation*). We show two ways in which loop checking can be used in that framework. Firstly, the search space can be reduced safely by a sound loop check, as sketched previously. The second application is the use of a complete (but unsound) loop check to characterize and improve termination criteria for partial deduction (this is called 'loop prevention' in [S2]). Therefore Chapter 6 includes a description of a class of complete, unsound loop checks.

Specific loop checks

In Section 2.3 and Chapter 3 we discuss some specific loop checks. The loop checks of Section 2.3 depend on the program. Those in Chapter 3 don't: they are *simple* loop checks. It appears that for practical purposes simple loop checks are more interesting than nonsimple ones.

The simple loop checks defined in Chapter 3 have in common that they are based on making comparisons between goals and their ancestors in the SLD-tree. A goal is pruned if it is 'sufficiently similar' to one of its ancestors. Based on their notions of 'sufficiently similar', these loop checks are divided into three groups, called *equality checks*, *subsumption checks* and *context checks* respectively. Each group contains weakly sound and sound versions.

For all three groups of loop checks, we identify classes of programs for which they are complete (as they are at least weakly sound, they cannot be complete for all programs). The main restriction we make is that *when studying completeness* we rule out programs that compute over an infinite domain. In order to avoid unnecessary complications, we do so by restricting our attention to programs without function symbols (*function-free* programs).

This does *not* mean that these loop checks can be applied only when interpreting function-free programs, nor that these loop checks can only be complete for function-free programs. We do not study explicitly more

permissive conditions leading to a finite domain, but most of our results can be generalized easily in this direction.

We mention two possibilities. One is the use of typed functions. This solution requires an adapted form of SLD-resolution, particularly of the unification algorithm therein. Another solution is to consider only programs (and goals) that satisfy the *bounded term-size property* ([vG2], [P]). This property states that terms occurring in the computation do not grow beyond a certain limit. It is not necessary that this limit is known in advance.

As one would expect, equality checks are based on the *equality* between goals. They are complete for function-free *restricted* programs. Restricted programs allow a restricted form of recursion (hence the name): only one recursive call per clause is allowed. For example, the transitive closure program P mentioned before is restricted. Thus P becomes not only logically, but also operationally correct when the PROLOG-interpreter is augmented with an equality check. (In contrast, this solution cannot be applied to an alternative specification of the transitive closure of r obtained by replacing the second clause of P by $tc(x,y) \leftarrow tc(x,z),tc(z,y)$, as the resulting program is not any more restricted.)

Subsumption checks are based on the *inclusion* of goals. Consequently, they are stronger than equality checks. They are not only complete for function-free restricted programs, but also for function-free programs in which no new variables are introduced in clause bodies and for function-free programs in which each variable occurs at most once in every clause body.

Context checks compare atoms in goals, but they take the rest of the goal (the *context* of the atom) into account. The context checks are complete for the three classes of programs mentioned.

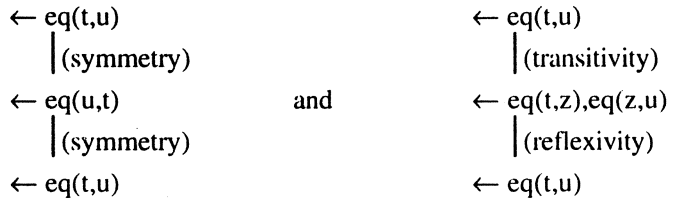
Another example

Consider the following program EAX, that summarizes the logical properties of *equality*. (The rules in EAX are called the *equality axioms*.)

$$
\begin{aligned}
EAX = \{\ & eq(x,x) \leftarrow. && \text{(reflexivity)},\\
& eq(x,y) \leftarrow eq(y,x). && \text{(symmetry)},\\
& eq(x,y) \leftarrow eq(x,z),eq(z,y). && \text{(transitivity)},\\
& eq(f(x_1,\ldots,x_n),f(y_1,\ldots,y_n)) \leftarrow eq(x_1,y_1),\ldots,eq(x_n,y_n). && \text{(substitutivity)}\ \}.
\end{aligned}
$$

EAX is usually added to an *equational program* EP. A substitutivity axiom is present for every n-ary function symbol in the language of EP.

Using EAX as a PROLOG program leads to a search space that contains many redundant derivations. One cannot expect a loop check to prune them all, as there can be infinitely many solutions to a query, but some of the most obvious redundancies are removed already by the equality checks, for example:



With or without loop checking, this is a rather naive way to deal with equality. In fact, the question how equality can be incorporated into logic programming has created a research area of its own. For a thorough survey we refer to [Hö].

Generalizations

One could say that both Chapter 4 and Chapter 5 contain generalizations of the elementary framework of loop checking outlined so far, but the nature of these generalizations is quite different. Chapter 4 focuses solely on the completeness of loop checks. Its central theorem, the Generalization Theorem, allows us to generalize certain completeness results. The theorem is applied on the results for the subsumption and context checks mentioned before; stronger completeness results for these loop checks are thus obtained.

Chapter 5 introduces loop checks for a broader class of programs, namely programs with negative literals in their clauses. The declarative and procedural semantics for logic programs with negation are considerably more complicated than for programs without negation ([ABo], [Cl2], [P1], [P2]). As a result the effect of applying a loop check is also more complex. Nevertheless we show that loop checks for programs without negation can easily be extended to loop checks for *locally stratified* programs, for which satisfactory semantics have been defined.

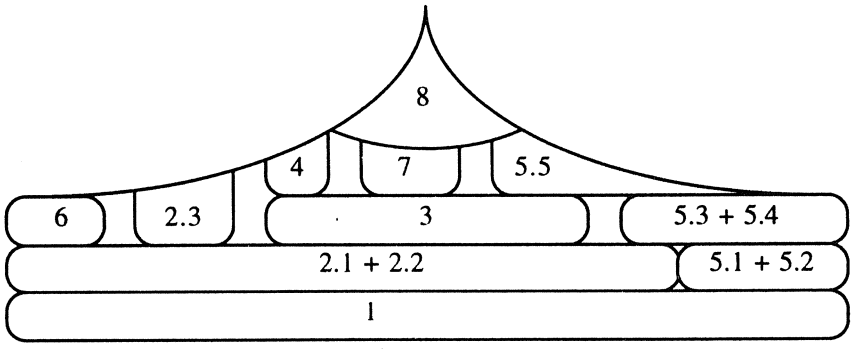
Towards an implementation of loop checking

Finally, we pay some attention to the implementation of loop checking. The loop checks we describe in Chapter 3 compare every goal with every ancestor of it. Thus the number of comparisons is quadratic in the number of goals generated. In practice this might turn out to be too expensive. Section 7.1 describes less expensive loop checks that compare only some *selected* goals. It is shown that this technique usually retains completeness results. Moreover, a proper selection renders the number of comparisons linear in the number of goals generated.

Section 7.2 reports a preliminary study on the practical implementation of several loop checks, in particular the equality and subsumption checks and their variants that compare only selected goals. Two implementations are described. The first one consists of a meta-interpreter, the second one transforms the input program such that the new program incorporates loop checking. Although these implementations are not very efficient, and some questions remain open, the measurements we performed on our implementations seem to suggest that loop checking can be done efficiently.

Interdependence of the chapters

The following figure shows how the various chapters depend on each other. It seems that Chapter 8 is the summit of this book. This is not the case: it discusses work by others related to several subjects discussed here.



1. Logic Programming

In this chapter we recall briefly the basic definitions of pure logic programming. More details and motivation can be found in [A] and [L].

1.1. Syntax

The language

A logic program is simply a set of formulas in a limited first-order language. The alphabet of such a language is determined by:

- a finite set of *constants*, denoted by a, b, c, d, ...,
- a finite set of *function symbols*, denoted by f, g, h, ...,
- a finite set of *predicate symbols*, denoted by p, q, r, s, ...

Each function and predicate symbol has a fixed *arity*, that is its number of arguments. Function symbols (or just *functions*) have a positive arity (constants are introduced instead of 0-ary functions), but 0-ary predicate symbols (or just *predicates*) are admitted.

We assume that an infinite set VAR of variables is fixed; typical elements of VAR are x, y, z, x', x_1, x_2. Every alphabet contains VAR and the set of symbols {'(', ')', ',', '$\neg$', '$\leftarrow$', '$\wedge$', '$\vee$'}. We now define by induction *terms* over a given alphabet:

- a variable is a term,
- a constant is a term,
- if f is an n-ary function symbol and $t_1,\ldots,t_n$ are terms, then $f(t_1,\ldots,t_n)$ is a term.

If p is an n-ary predicate symbol and $t_1,\ldots,t_n$ are terms, then $p(t_1,\ldots,t_n)$ is an *atom*. Terms are denoted by t, u, t_1, t_2, ... and atoms by A, B, A_1, A_2, ...

A *clause* is a formula of the form $A_1\vee\ldots\vee A_m\leftarrow B_1\wedge\ldots\wedge B_n$ (m,n $\geq$ 0), usually written as $A_1,\ldots,A_m\leftarrow B_1,\ldots,B_n$, where $A_1,\ldots,A_m,B_1,\ldots,B_n$ are atoms. We distinguish the following 'special' clauses by their values for m an n:

- m = 0: *goal* clauses or *goals*, with as a special case
- m = 0 and n = 0: the *empty* clause, denoted by $\square$;
- m = 1: *program* clauses or *definite* clauses, with as a special case
- m = 1 and n = 0: *unit* clauses or *facts*.

A *positive literal* is just an atom (A), a *negative literal* is the negation of an atom ($\neg$ A). Literals are denoted by L_1, L_2, ... In Chapter 5 we shall encounter *general clauses*: constructs of the form $A_1 \vee \ldots \vee A_m \leftarrow L_1 \wedge \ldots \wedge L_n$, where $A_1,\ldots,A_m$ are atoms but $L_1,\ldots,L_n$ are (not necessarily positive) literals. Again, such a general clause is called a *general goal* if m = 0, a *general program clause* if m = 1. (General) goals are denoted by G, H, G_1, G_2, ..., (general) program clauses by C_1, C_2, ... For a (general) program clause $A \leftarrow L_1,\ldots,L_n$, A is called the *head* of the clause and $L_1,\ldots,L_m$ the *body*. For a goal G, |G| denotes its *length*, i.e., the number of atoms in it.

A *logic program* (or just a *program*) is a finite nonempty set of program clauses. A *general logic program* (or just a *general program*) is a finite nonempty set of general program clauses. With each (general) program P we can uniquely associate a first-order language L_P whose constants, functions and predicates are those occurring in P. P is *function-free* if P contains no function symbols.

An *expression* is a term, literal, sequence of literals, clause or program, and is denoted by E. For an expression E, var(E) denotes the set of variables that occur in E. If var(E) = $\varnothing$ then E is called *ground*.

Substitutions

Consider now a fixed first-order language. A *substitution* is a finite mapping from variables to terms, and is written as

$$\theta = \{x_1/t_1,\ldots,x_n/t_n\}.$$

It is to be read: the variables $x_1,\ldots,x_n$ are mapped (*bound*) to $t_1,\ldots,t_n$ respectively. The notation implies that the variables $x_1,\ldots,x_n$ are different. We also assume that $x_i \neq t_i$ (i = 1,...,n). A pair x_i/t_i is called a *binding*. $\{x_1,\ldots,x_n\}$ is called the *domain* of θ (*dom*(θ)), $\{t_1,\ldots,t_n\}$ the *range* of θ (*ran*(θ)). If θ is a bijection, that is if dom(θ) = ran(θ), then θ is called a *renaming*. Thus a renaming is simply a permutation of a finite number of variables. The *empty substitution* or *identity substitution* is denoted by ε: ε = dom(ε) = ran(ε) = $\varnothing$.

Substitutions operate on expressions. For an expression E and a substitution θ, Eθ stands for the result of applying θ to E, which is obtained by *simultaneously* replacing each occurrence in E of a variable from dom(θ) by the corresponding term. A substitution θ is *ground* (in a given context) if Eθ is ground for all expressions E that occur in that context.

Substitutions can be composed. Given substitutions θ and η, their *composition* $\theta\eta$ is defined as $\eta\circ\theta$ (regarding substitutions as functions). Alternatively, if $\theta = \{x_1/t_1,\ldots,x_n/t_n\}$ and $\eta = \{y_1/u_1,\ldots,y_n/u_n\}$, then $\theta\eta$ is obtained by removing from the set $\{x_1/t_1\eta,\ldots,x_n/t_n\eta,y_1/u_1,\ldots,y_n/u_n\}$ the pairs $x_i/t_i\eta$ for which $x_i = t_i\eta$ as well as the pairs y_i/u_i for which $y_i \in \{x_1,\ldots,x_n\}$. Thus for an expression E and substitutions σ, θ and η, $(E\theta)\eta = E(\theta\eta)$ and can be written as $E\theta\eta$; $(\sigma\theta)\eta = \sigma(\theta\eta)$ and can be written as $\sigma\theta\eta$. A substitution θ is *idempotent* if $\theta\theta = \theta$. It is easy to see that a substitution θ is idempotent if and only if $dom(\theta) \cap var(ran(\theta)) = \varnothing$. So the only idempotent renaming is ε.

For two expressions E and F, E is an *instance* of F (F is *more general* than E, notation $F \leq E$) if for some substitution θ, $E = F\theta$. E and F are *variants* if $E = F\theta$ for some *renaming* θ. A substitution θ is *more general* than η if $\eta = \theta\gamma$ for some substitution γ. θ and η are *variants* if $\eta = \theta\gamma$ for some *renaming* γ. For a program P, *ground(P)* denotes the set of all ground instance of clauses in P in the language L_P. Notice that ground(P) can be infinite.

Unification

Consider two atoms A and B. If for some substitution θ we have $A\theta = B\theta$, then θ is called a *unifier* of A and B and we say then that A and B are *unifiable*. A unifier θ of A and B is called their *most general unifier* (or *mgu* in short) if it is more general than any other unifier of A and B. A unifier θ of A and B is called *relevant* if $dom(\theta) \subseteq var(A) \cup var(B)$. It is easy to prove that every idempotent mgu of A and B is relevant. The following theorem is due to Robinson [Ro].

THEOREM 1.1.1 (Unification Theorem). *There exists a* unification algorithm *which for any two atoms produces an idempotent most general unifier if they are unifiable and reports nonexistence of a unifier otherwise.*

PROOF. The unification algorithm we give here was first presented by Martelli & Montanari ([MM]). Two atoms can only be unified if they have the same predicate symbol. When $p(s_1,\ldots,s_n)$ and $p(t_1,\ldots,t_n)$ are to be unified, first the set of equations $\{s_1 = t_1, \ldots, s_n = t_n\}$ is constructed. This set is then transformed according to the following six rules:

(a) $E \dot\cup \{t = x\} \Rightarrow E \cup \{x = t\}$ if $t \notin VAR$,
(b) $E \dot\cup \{x = x\} \Rightarrow E$,
(c1) $E \dot\cup \{f(s_1,\ldots,s_n) = f(t_1,\ldots,t_n)\} \Rightarrow E \cup \{s_1 = t_1,\ldots,s_n = t_n\}$ $(n \geq 0)$,
(c2) $E \dot\cup \{f(s_1,\ldots,s_n) = g(t_1,\ldots,t_m)\} \Rightarrow$ failure if $f \neq g$,
(d1) $E \dot\cup \{x = t\} \Rightarrow E\{x/t\} \cup \{x = t\}$ if $x \notin var(t)$ and $x \in var(E)$,
(d2) $E \dot\cup \{x = t\} \Rightarrow$ failure if $x \neq t$ and $x \in var(t)$,

until none of these rules is applicable. (Here $\dot\cup$ denotes the disjoint union.) If the algorithm ends in failure, then the two atoms are not unifiable. Otherwise we take $\theta = \{x/t \mid (x = t) \in E\}$, where E denotes the final set of equations. We omit the proof that this algorithm is correct and that it always terminates. □

1.2. SLD-resolution

SLD-derivations

Let $G = \leftarrow A_1,\ldots,A_n$ be a goal and $C = A \leftarrow B_1,\ldots,B_m$ be a program clause. If for some i, $1 \leq i \leq n$, A_i and A unify with an idempotent mgu θ, then we call

$$G' = \leftarrow (A_1,\ldots,A_{i-1},B_1,\ldots,B_m,A_{i+1},\ldots,A_n)\theta$$

a *resolvent* of G and C. The transformation from G to G' is called a *resolution step* and denoted by $G \Rightarrow_{C,\theta} G'$. The atom A_i is called the *selected* atom in G.

Now let P be a program and G_0 a goal. An *SLD-derivation* of $P \cup \{G_0\}$ is a (finite or infinite) sequence $G_0 \Rightarrow_{C_1,\theta_1} G_1 \Rightarrow \ldots \Rightarrow G_{k-1} \Rightarrow_{C_k,\theta_k} G_k \Rightarrow \ldots$ of resolution steps such that for all i = 1,2,..., C_i is a variant of a clause in P and $var(C_i) \cap (var(G_0) \cup var(C_1) \cup \ldots \cup var(C_{i-1})) = \varnothing$ (*standardizing apart*). It is important to note that, in contrast with [A], we do *not* require SLD-derivations to be a *maximal* sequence.

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 $G_i = \square$, $G_0{}^\sim\theta_1\theta_2\ldots\theta_i \leftarrow G_i{}^\sim$ otherwise.

In every nonempty goal in an SLD-derivation, one atom must be selected. Which one it is may depend on every aspect of the preceding derivation. This dependency is formalized by the notion of *selection rule*. Let HIS (for history) be the set of finite SLD-derivations of which the last goal is nonempty. A *selection rule* **R** is a function which when applied to an element D of HIS selects

an atom in the last element of D. For example, the *leftmost selection rule* selects always the leftmost atom of a goal, the *rightmost selection rule* selects always the rightmost atom.

Given a selection rule **R**, an SLD-derivation $D = (G_0 \Rightarrow_{C_1,\theta_1} G_1 \Rightarrow \ldots \Rightarrow G_{k-1} \Rightarrow_{C_k,\theta_k} G_k \Rightarrow \ldots)$ is an SLD-derivation *via* ***R*** if for all $i \geq 0$ (and $i \leq |D|$ if D is finite), if $G_i \neq \square$ then the selected atom in G_i is $\mathbf{R}(G_0 \Rightarrow_{C_1,\theta_1} G_1 \Rightarrow \ldots \Rightarrow G_i)$. Obviously for every SLD-derivation D there is a selection rule **R** such that D is an SLD-derivation via **R** (this is not the case in [L]). This explains the abbreviation 'SLD': SLD-resolution is Selection rule driven Linear resolution for Definite clauses.

For a program P and a goal G, we distinguish SLD-derivations of $P \cup \{G\}$ with four different outcomes:

- *infinite* derivations,
- *successful* derivations: finite derivations of which the last goal is empty,
- *failed* derivations: finite derivations of which the selected atom in the last goal does not unify with the head of any variant of a clause in P,
- *unfinished* derivations: all other finite derivations.

A *proper initial subderivation* of an SLD-derivation $D = (G_0 \Rightarrow_{C_1,\theta_1} G_1 \Rightarrow \ldots \Rightarrow G_{k-1} \Rightarrow_{C_k,\theta_k} G_k \Rightarrow \ldots)$ is an unfinished SLD-derivation $D' = (G_0 \Rightarrow_{C_1,\theta_1} G_1 \Rightarrow \ldots \Rightarrow G_{k-1} \Rightarrow_{C_k,\theta_k} G_k)$ such that if D is finite, $|D'| < |D|$.

Successful SLD-derivations are also called *SLD-refutations*. The *computed answer substitution* of an SLD-refutation $G_0 \Rightarrow_{C_1,\theta_1} G_1 \Rightarrow \ldots \Rightarrow G_{k-1} \Rightarrow_{C_k,\theta_k} G_k = \square$ is the substitution $\theta_1\theta_2\ldots\theta_k$ (in contrast with [A] and [L] *not* restricted to $\mathrm{var}(G_0)$). The *computed answer* is the resultant associated to $\square$, which is $G_0{\sim}\theta_1\theta_2\ldots\theta_k$.

Two SLD-derivations are *variants* if the following conditions hold:

- their initial goals are variants,
- in corresponding goals atoms in the same position are selected,
- in corresponding derivation steps the clauses used are variants.

It has been proved in [LS] that these conditions imply that all corresponding resultants are variants. In particular, if the derivations are successful then their computed answers are variants (but not necessarily their computed answer substitutions).

Properties of SLD-derivations

The following lemma shows that our conditions on SLD-derivations, notably standardization apart and the use of idempotent mgu's, guarantee that a variable in an SLD-derivation has only one 'meaning': it appears only in one consecutive sequence of derivation steps and it is not renamed within this sequence.

LEMMA 1.2.1. *Let* $D = (G_0 \Rightarrow_{C_1,\theta_1} G_1 \Rightarrow \ldots \Rightarrow G_i \Rightarrow_{C_{i+1},\theta_{i+1}} G_{i+1} \Rightarrow \ldots)$ *be an SLD-derivation and let* $0 \leq i < k$ $(\leq |D|)$. *If* $x \in var(C_{i+1}) \cup var(G_i)$ *and* $x \in var(G_k)$, *then for all* j, $i < j \leq k$, $x \in var(G_j)$ *and* $x\theta_j = x$.

PROOF. We use induction on j from k down to i. $x \in var(G_k)$ is given. Now assume that $i \leq j < k$ and $x \in var(G_{j+1})$. We prove that $x\theta_{j+1} = x$ and that if $j > i$, $x \in var(G_j)$. Let $G_j = \leftarrow(S_1,A,S_2)$, where A is the selected atom in G_j. Let $C_{j+1} = H\leftarrow S_3$. ($S_1$, S_2 and S_3 are possibly empty sequences of atoms.) Then θ_{j+1} is an idempotent mgu of A and H and $G_{j+1} = \leftarrow(S_1,S_3,S_2)\theta_{j+1}$. So $x \in var(S_1,S_3,S_2)\theta_{j+1}$, hence for some $y \in var(S_1,S_3,S_2)$, $x \in var(y\theta_{j+1})$. Two cases arise.

- $x = y$. Thus $x\theta_{j+1} = x$. Also, if $j > i$, $x \notin var(S_3)$ since $x \in var(C_{i+1}) \cup var(G_i)$ and S_3 is standardized apart. So $x \in var(S_1,S_2) \subseteq var(G_j)$.
- $x \neq y$. Then $x \in var(ran(\theta_{j+1}))$, and since θ_{j+1} is idempotent, $x \notin dom(\theta_{j+1})$, so $x\theta_{j+1} = x$. Also, since θ_{j+1} is relevant, $x \in var(A,H)$. If $j > i$, $x \notin var(H)$ since $x \in var(C_{i+1}) \cup var(G_i)$ and H is standardized apart. So $x \in var(A) \subseteq var(G_j)$.

So in both cases we have $x\theta_{j+1} = x$ and if $j > i$ also $x \in var(G_j)$. □

The following definition captures the notion that two variables in a goal are related, i.e., that they might be unified in an attempt to refute the goal. (Compare this notion with *connected (sets of) predicate instances* in [Na].) We then prove that when two variables occur unrelated in a certain goal, they cannot be related in any goal later in the derivation.

DEFINITION 1.2.2.

Let S be a set of atoms. We define the relation $\sim_S$ on variables as:

$x \sim_S y$ if there is an atom A in S such that $x,y \in var(A)$.

Obviously, $\sim_S$ is a symmetrical relation. Now we define the relation $\approx_S$ to be the transitive and reflexive closure of $\sim_S$. Then $\approx_S$ is an equivalence relation.

An equivalence class of $\approx_S$ is called a *chain (in S)*. For $x \in var(S)$, the chain of x is denoted by $C_S(x)$, or $C(x)$ whenever S is clear from the context. □

LEMMA 1.2.3. *Let* $D = G_0 \Rightarrow_{C_1,\theta_1} G_1 \Rightarrow \ldots \Rightarrow G_{i-1} \Rightarrow_{C_i,\theta_i} G_i \Rightarrow \ldots$ *be an SLD-derivation and let* $0 < i (\leq |D|)$.

If $x \approx_{G_i} y$ *and* $x,y \in var(G_{i-1})$, *then* $x \approx_{G_{i-1}} y$.

PROOF. Let $G_{i-1} = \leftarrow(A,R)$, where A is the selected atom in G_{i-1}. Let $C_i = H \leftarrow S$ and let θ_i be an mgu of A and H. Then $G_i = \leftarrow(S,R)\theta_i$. Assume $x \neq y$ (for $x = y$ the claim is trivial). Since $x \approx_{G_i} y$, there is a sequence of variables $x = w_1, w_2, \ldots, w_{2n} = y$ in G_i such that $w_{2j-1} \approx_{S\theta_i} w_{2j}$ for $1 \leq j \leq n$ and $w_{2j} \sim_{R\theta_i} w_{2j+1}$ for $1 \leq j < n$.

For $1 < j < 2n$, every variable $w_j \in var(R\theta_i)$, so we can choose for it a corresponding variable $z_j \in var(R) \subseteq var(G_{i-1})$ such that $w_j \in var(z_j\theta_i)$. Since θ_i is idempotent, and $x,y \in var(G_{i-1}) \cap var(G_i)$, we can choose $z_1 = w_1 = x = x\theta_i$ and $z_{2n} = w_{2n} = y = y\theta_i$. Now let $1 \leq j < 2n$.

We prove that $z_j \approx_{G_{i-1}} z_{j+1}$. Two cases arise.

- j is even, so $w_j \sim_{R\theta_i} w_{j+1}$.

Then there is an atom B in R such that $w_j, w_{j+1} \in var(B\theta_i)$. So we have variables $v_j, v_{j+1} \in var(B)$ such that $w_j \in var(v_j\theta_i)$ and $w_{j+1} \in var(v_{j+1}\theta_i)$. So $v_j \sim_B v_{j+1}$, and hence $v_j \sim_R v_{j+1}$. For v_j (and analogously for v_{j+1}) two subcases arise.

- $v_j = z_j$. Then $v_j \approx_A z_j$.
- $v_j \neq z_j$. Then, since $w_j \in var(v_j\theta_i) \cap var(z_j\theta_i)$ and θ_i is relevant, we have $v_j, z_j \in var(A)$. Hence $v_j \approx_A z_j$.

Therefore $z_j \approx_A v_j \sim_R v_{j+1} \approx_A z_{j+1}$, so $z_j \approx_{G_{i-1}} z_{j+1}$.

- j is odd, so $w_j \approx_{S\theta_i} w_{j+1}$.

If $w_j = w_{j+1}$, then $z_j = z_{j+1}$, so $z_j \approx_{G_{i-1}} z_{j+1}$. Otherwise, we can prove that $z_j \in var(A)$ (and analogously $z_{j+1} \in var(A)$). Again two subcases arise.

- $z_j\theta_i \neq z_j$. Then $z_j \in var(A)$: θ_i is relevant and $z_j \in var(G_{i-1})$, so $z_j \notin var(H)$.
- $z_j\theta_i = z_j$. Then $w_j = z_j \in var(S\theta_i)$, say $v_j \in var(S)$ such that $z_j \in var(v_j\theta_i)$. Then $v_j\theta_i \neq v_j$, since $v_j \in var(S)$, $z_j \in var(G_{i-1})$ and S is standardized apart. Therefore $v_j \in var(H)$, and hence $z_j \in var(A)$.

Now $z_j \sim_A z_{j+1}$, so $z_j \approx_{G_{i-1}} z_{j+1}$.

Therefore we have $x = z_1 \approx_{G_{i-1}} z_2 \approx_{G_{i-1}} z_3 \approx_{G_{i-1}} \cdots \approx_{G_{i-1}} z_{2n} = y$. □

Normal SLD-derivations

In some cases it appears to be convenient to restrict the choice of the mgu by disallowing the 'needless renaming of variables in a derivation'. We explain this now. When we have a variable x in the selected atom of the goal which is to be unified with a variable y in the input clause, then two idempotent mgu's are available: {x/y} and {y/x}.

When {x/y} is chosen, it is likely that the variable y occurs further on in the derivation as a substitute for x, whereas x itself does not occur any more. On the other hand, if {y/x} is chosen, the variable x is retained and the variable y will not occur in any goal of the derivation. Therefore the renaming from x to y is considered to be a needless renaming. So we choose {y/x}, thereby retaining the 'older' variable x and adjusting the 'newer' variable y.

A more indirect instance of the same principle is shown in the derivation

$$\leftarrow A(x) \Rightarrow_{A(x')\leftarrow B(x',y),\ \{x'/x\}} \leftarrow B(x,y) \Rightarrow_{B(z,z)\leftarrow,\ \{y/x,z/x\}} \square.$$

In the first step {x'/x} is chosen for the reason described above. In the second step, the choice of {x/z,y/z} is out of the question for the same reason. However, this still leaves the choice between {x/y,z/y} and {y/x,z/x}. Although x and y occur both in B(x,y), x appears earlier in the derivation than y. Therefore we choose {y/x,z/x}, thereby again retaining the older variable x and adjusting the newer variable y.

It is important to note two things. Firstly, Lemma 1.2.1 says that a variable cannot be introduced, disappear and later on in the derivation reappear, which would complicate the decision criterion given above. Secondly, the choice of the mgu is still nondeterministic, as is shown in the derivation

$$\leftarrow A \Rightarrow_{A\leftarrow B(x,y),\ \varepsilon} \leftarrow B(x,y) \Rightarrow_{B(z,z)\leftarrow,\ \{y/x,z/x\}} \square.$$

Here the choice between {y/x,z/x} and {x/y,z/y} is arbitrary.

We now formalize these intuitions.

DEFINITION 1.2.4 (Normal SLD-derivation).

Let $D = (G_0 \Rightarrow_{C_1,\theta_1} G_1 \Rightarrow \ldots \Rightarrow G_{i-1} \Rightarrow_{C_i,\theta_i} G_i \Rightarrow \ldots)$ be an SLD-derivation. For every variable x occurring in D, we define

$$tag(x) = \begin{cases} 0 & \text{if } x \in var(G_0), \\ i & \text{if } x \in var(C_i). \end{cases}$$

D is a *normal* SLD-derivation if for every $i > 0$ (and $i \leq |D|$ when D is finite), for every variable $x \in var(G_{i-1})$: if $x\theta_i$ is a variable, then $tag(x) \geq tag(x\theta_i)$. □

Intuitively, the lower the tag of a variable is, the 'older' it is. The following lemma shows that we may restrict our attention to normal SLD-derivations.

LEMMA 1.2.5. *Every SLD-derivation has a normal variant.*

PROOF. Consider the unification algorithm of Theorem 1.1.1. We change rule (a) of this algorithm to:

(a') $E \cup \{t = x\} \Rightarrow E \cup \{x = t\}$ if $t \notin VAR$ or $tag(t) < tag(x)$,

thus taking tags into account.

Recall that we take $\theta = \{x/t \mid (x = t) \in E\}$, where E denotes the final set of equations to which none of the rules is applicable. Thus whenever $x\theta = y \neq x$, we have that $(x = y) \in E$ and rule (a') is not applicable on E, hence $tag(x) \geq tag(y)$. Showing that this algorithm also terminates and yields an idempotent mgu of $p(s_1,\ldots,s_n)$ and $p(t_1,\ldots,t_n)$ is straightforward. □

Properties of normal SLD-derivations

In this subsection we prove some properties of normal SLD-derivations that appear to be needed in Chapter 3 and 4. The reader who is not interested in such technical details is encouraged to skip the rest of this section.

LEMMA 1.2.6. *Let* $D = (G_0 \Rightarrow_{C_1,\theta_1} G_1 \Rightarrow \ldots \Rightarrow G_{i-1} \Rightarrow_{C_i,\theta_i} G_i \Rightarrow \ldots)$ *be a normal SLD-derivation and let* $0 \leq j < k$ $(\leq |D|)$. *Let C be a chain in* G_j. *Then* $C\theta_k \cap VAR \subseteq C$.

PROOF. Let $x \in C$ and assume that $x\theta_k$ is a variable. We prove that $x\theta_k \in C$.

If $x\theta_k = x$ then clearly $x\theta_k \in C$.

Otherwise, $x \in var(G_{k-1})$, since θ_k is relevant and by standardizing apart, $x \notin var(C_k)$. D is normal, $x \in var(G_{k-1})$ and $x\theta_k$ is a variable, so $tag(x) \geq tag(x\theta_k)$. Hence $x\theta_k \notin var(C_k)$. $x\theta_k \neq x$ and θ_k is relevant, so $x\theta_k \in var(G_{k-1})$. Thus x and $x\theta_k$ occur both in the selected atom of G_{k-1}. Therefore $x \approx_{G_{k-1}} x\theta_k$.

Also, $tag(x\theta_k) \leq tag(x) \leq j$, thus by Lemma 1.2.1, for every i such that $j \leq i < k$, $x \in var(G_i)$ and $x\theta_k \in var(G_i)$. Applying Lemma 1.2.3 k−1−j times yields that $x \approx_{G_j} x\theta_k$. Hence $x\theta_k \in C$. □

COROLLARY 1.2.7. *Let* $D = (G_0 \Rightarrow_{C_1,\theta_1} G_1 \Rightarrow \ldots \Rightarrow G_{i-1} \Rightarrow_{C_i,\theta_i} G_i \Rightarrow \ldots)$ *be a normal SLD-derivation of a function-free program P and* G_0 *and let* $0 \leq j < k\ (\leq |D|)$. *Then* $var(G_j\theta_k) \subseteq var(G_j)$ *and* $var(G_j\theta_{j+1}\ldots\theta_k) \subseteq var(G_j)$.

PROOF. Let $x \in var(G_j\theta_k)$. P is function-free, so for some $y \in var(G_j)$, $x = y\theta_k$. Now by Lemma 1.2.6, $x = y\theta_k \in C_{G_j}(y)\theta_k \cap VAR \subseteq C_{G_j}(y) \subseteq var(G_j)$.

Now $var((G_j\theta_{j+1})\theta_{j+2}\ldots\theta_k) \subseteq var(G_j\theta_{j+2}\ldots\theta_k) \subseteq \ldots \subseteq var(G_j\theta_k) \subseteq var(G_j)$. □

COROLLARY 1.2.8. *Let* $D = (G_0 \Rightarrow_{C_1,\theta_1} G_1 \Rightarrow \ldots \Rightarrow G_{i-1} \Rightarrow_{C_i,\theta_i} G_i \Rightarrow \ldots)$ *be a normal SLD-derivation and let* $0 \leq j < k\ (\leq |D|)$. *Let C be a chain in* G_j. *Then* $C\theta_{j+1}\theta_k \cap VAR \subseteq C\theta_{j+1}$ *and* $C\theta_{j+1}\ldots\theta_k \cap VAR \subseteq C\theta_{j+1}$.

PROOF. If $j+1 = k$, then the claim is trivial. So assume $j+1 < k$.

Let $x \in C\theta_{j+1}$ and assume that $x\theta_k$ is a variable. We prove that $x\theta_k \in C\theta_{j+1}$. By Lemma 1.2.6, $x \in C\theta_{j+1} \cap VAR$ implies $x \in C$. Therefore, again by Lemma 1.2.6, $x\theta_k \in C\theta_k \cap VAR \subseteq C$. Two cases arise.

- $x\theta_k\theta_{j+1} = x\theta_k$. Then $x\theta_k \in C$ implies $x\theta_k = x\theta_k\theta_{j+1} \in C\theta_{j+1}$.
- $x\theta_k\theta_{j+1} \neq x\theta_k$. Then $x\theta_k \notin var(G_{j+1})$, since θ_{j+1} is idempotent. As we have $x\theta_k \in C \subseteq var(G_j)$, $x\theta_k \notin var(G_{k-1})$ by Lemma 1.2.1 and $x\theta_k \notin var(C_k)$ by standardizing apart. Thus $x\theta_k = x \in C\theta_{j+1}$.

Now $((C\theta_{j+1})\theta_{j+2})\ldots\theta_k \cap VAR \subseteq (C\theta_{j+1})\theta_{j+3}\ldots\theta_k) \cap VAR \subseteq \ldots \subseteq (C\theta_{j+1})\theta_k \cap VAR \subseteq C\theta_{j+1}$. □

In order to formulate the final property of normal derivations we prove in this section, we need the following definition.

DEFINITION 1.2.9 (Local selection rule).
(This definition is equivalent to the definition of local selection functions in [V].) A selection rule **R** is *local* if every SLD-derivation $D = (G_0 \Rightarrow_{C_1,\theta_1} G_1 \Rightarrow \ldots)$ via **R** satisfies the following property. If in a goal G_i, an atom A is selected and in a goal G_j $(j > i)$ the further instantiated version $B\theta_{i+1}\ldots\theta_j$ of the atom B $(\neq A)$ in G_i is selected, then A is resolved completely between G_i and G_j. □

It is easy to see that the leftmost selection rule and the rightmost selection rule are examples of local selection rules.

COROLLARY 1.2.10. *Let $D = (G_0 \Rightarrow_{C_1,\theta_1} G_1 \Rightarrow \ldots \Rightarrow G_{i-1} \Rightarrow_{C_i,\theta_i} G_i \Rightarrow \ldots)$ be a normal SLD-derivation of a function-free program P and G_0 and let $0 \leq j < k$ ($\leq |D|$). Let A be the selected atom in G_j. Suppose a local selection rule is used between G_j and G_k and A is not completely resolved before G_k. Then $var(A\theta_k) \subseteq var(A)$ and $var(A\theta_{j+1}\ldots\theta_k) \subseteq var(A)$.*

PROOF. Let $x \in var(A)$ and assume that $x\theta_k$ is a variable. We prove that $x\theta_k \in var(A)$. Let $G_j = (A,R)$ and regard the derivation $\leftarrow A = H_j \Rightarrow_{C_{j+1},\theta_{j+1}} H_{j+1} \Rightarrow \ldots \Rightarrow_{C_k,\theta_k} H_k$ (hence for $j \leq i \leq k$, $G_i = (H_i, R\theta_{j+1}\ldots\theta_i)$). Note that this derivation exists, since a local selection rule is used and A is not completely resolved before G_k, and note that the derivation is normal. Now $x \in var(A) = var(H_j)$ implies $x\theta_k \in var(H_j) = var(A)$ by Corollary 1.2.7.

Now $var((A\theta_{j+1})\theta_{j+2}\ldots\theta_k) \subseteq var(A\theta_{j+2}\ldots\theta_k) \subseteq \ldots \subseteq var(A\theta_k) \subseteq var(A)$.□

1.3. Soundness and Completeness of SLD-resolution

SLD-trees

We are now ready to define the search space of top-down interpreters for logic programming, the *SLD-tree*.

DEFINITION 1.3.1 (SLD-tree).

Let P be a program, G a goal and **R** a selection rule.

An *SLD-tree of $P \cup \{G\}$ via **R*** is a tree satisfying the following:

- Each node of the tree is a goal; the root node is G.
- If a goal G' has a child G", then the edge between G' and G" is $\Rightarrow_{C,\theta}$ such that $G' \Rightarrow_{C,\theta} G"$ is a resolution step.
- Each branch of the tree is an SLD-derivation of $P \cup \{G\}$ via **R**.
- Nodes which are the empty goal are leaves.
- Let G' be a nonempty goal and let A be selected by **R** in G'. For every clause C in P such that a variant of the head of C unifies with A, G' has a child G" such that the clause in the edge from G' to G" is a variant of C. G' has no other children. □

Notice that the last condition implies that no branch of an SLD-tree is an unfinished SLD-derivation. An SLD-tree is *successful* if it contains the empty goal (equivalently: if at least one of its branches is a successful SLD-derivation),

otherwise it is *failed*. An SLD-tree is finite if it has no infinite branches (because programs are finite sets of clauses, SLD-trees are finitely branching).

Pruning a node (goal) in an SLD-tree means removing all its descendants. An *initial subtree* of a tree T is obtained by pruning zero or more nodes of T. Sometimes we shall call an initial subtree of an SLD-tree an *unfinished SLD-tree*: all its branches are (possibly unfinished) SLD-derivations.

Soundness and completeness

We assume that the reader is familiar with the notions *interpretation*, *model* and *semantical implication* (see [F] for an introduction). The latter is denoted by $\models$. Let P be a program and G a goal. A substitution θ is a *correct answer substitution* for $P \cup \{G\}$ and $G\theta$ is a *correct answer* for $P \cup \{G\}$ if $P \models G\theta$. We can now formulate the soundness and strong completeness of SLD-resolution, which is due to Clark [Cl1]. See also [AvE].

THEOREM 1.3.2 (Soundness and strong completeness of SLD-resolution).

Let P be a program, G a goal and **R** *a selection rule. Let T be an SLD-tree of* $P \cup \{G\}$ *via* **R**.

(i) Each computed answer (substitution) of an SLD-refutation in T is a correct answer (substitution) for $P \cup \{G\}$.

(ii) For each correct answer for $P \cup \{G\}$, *there is an SLD-refutation in T that gives a more general computed answer.* □

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, $CWA(P) = \{\neg A \mid A \text{ is a ground atom and } P \not\models A\}$. The soundness and completeness of SLD-resolution imply that $CWA(P) = \{\neg A \mid A$ is a ground atom and every SLD-tree of $P \cup \{\leftarrow A\}$ is failed$\}$.

Searching SLD-trees

When a program P and a goal G are presented to an interpreter for logic programming, it will perform a search through an SLD-tree for $P \cup \{G\}$ via some selection rule **R** (we can also say that it constructs such a tree). The order

in which the nodes of this SLD-tree are visited (constructed) is determined by the *search rule* (which is not to be confused with the selection rule).

From now on we assume that the clauses of a program P are ordered. Then the SLD-tree becomes an ordered tree: the outgoing edges of a node are ordered by the clauses in P of which they use a variant. (For simplicity we assume that P itself does not contain two clauses that are variants.) In practical systems the clauses of a program form a text, which is naturally ordered.

Informally, we call an interpreter sound if all the answers it gives are correct, and we call it complete if its answers 'cover' all correct answers. The soundness of SLD-resolution (Theorem 1.3.2.i) implies that interpreters based on searching the SLD-tree are sound, regardless of their search rule. The completeness of SLD-resolution (Theorem 1.3.2.ii) implies that these interpreters are also sound w.r.t. CWA: given a program P and a ground atom A, if the interpreter terminates on $P \cup \{\leftarrow A\}$ reporting failure, then the SLD-tree of $P \cup \{\leftarrow A\}$ is (finite and) failed, thus $\neg A \in CWA(P)$.

We would prefer to have a complete interpreter, which requires a search rule that eventually finds each successful branch on the SLD-tree. A breadth-first search rule satisfies this property, but is not very compatible with an efficient implementation. Therefore PROLOG uses a depth-first left-to-right search rule. If a solution in the SLD-tree occurs to the right of an infinite branch, this solution is not found by a standard PROLOG interpreter (sometimes reordering the clauses of the program helps, but not always; for an example see [L], page 59). Consequently the standard PROLOG interpreter is *not* complete.

Completeness w.r.t. CWA would require that for every program P and ground atom A such that $\neg A \in CWA(P)$, the interpreter terminates on $P \cup \{\leftarrow A\}$ reporting failure. But the SLD-tree of $P \cup \{\leftarrow A\}$ can be infinite, in which case an interpreter that tries to search it completely does not terminate. So in this case both a breadth-first interpreter and the standard PROLOG interpreter are incomplete w.r.t. CWA.

In Section 2.2 we show how the introduction of loop checking can improve the situation regarding the incompleteness of interpreters with a depth-first search rule as well as the general incompleteness of interpreters w.r.t. CWA.

2. Foundations of Loop Checking

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

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

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

2.1. What is a Loop Check?

Definitions

One might define a loop check as a function from SLD-trees to unfinished SLD-trees. However, this would be a very general definition, allowing practically everything. The purpose of a loop check is to prune an SLD-tree to an initial subtree of it. Moreover, we shall use here a more restricted definition: given a program P and a goal G, the decision to prune a node is based only upon its ancestors in the SLD-tree of $P \cup \{G\}$, 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 by Vieille [V] and Tamaki & Sato [TS] (see Chapter 8).

Due to this restriction we could define a loop check as a function which, given a program and an SLD-derivation, returns it unchanged if it is not pruned, and otherwise returns the proper initial subderivation of it that ends in the pruned

node. Of course, if a derivation D is pruned at the goal G, then every derivation D' that is the same as D until and including G must also be pruned at G: the ancestors of G are the same in D and D'.

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

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

DEFINITION 2.1.1.
Let L be a set of SLD-derivations. *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 2.1.2 (Simple loop check).
A *simple loop check* is a computable set L of finite SLD-derivations such that L is closed under variants and subderivation free. □

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

DEFINITION 2.1.3 (Loop check).
A *loop check* is a computable function L from programs to sets of SLD-derivations such that for every program P, L(P) is a simple loop check. □

Of course, we can treat a simple loop check L as a loop check, namely as the constant function $\lambda P.L$.

DEFINITION 2.1.4.
Let L be a loop check. An SLD-derivation D of $P \cup \{G\}$ is *pruned by* L if L(P) contains D or a proper initial subderivation of D. □

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

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

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

DEFINITION 2.1.5 (Variant of Atom check).
The *Variant of Atom check* is the set of SLD-derivations
$VA = Initials(\{D \mid D = (G_0 \Rightarrow_{C_1,\theta_1} G_1 \Rightarrow \ldots \Rightarrow G_{k-1} \Rightarrow_{C_k,\theta_k} G_k)$ such that for some i and j, $0 \le i \le j < k$, G_k contains an atom A that is
- a variant of an atom A' in G_i and
- the result of an attempt to resolve $A'\theta_{i+1}\ldots\theta_j$, the further instantiated version of A', that is selected in $G_j\}$). □

We now illustrate the use of this loop check.

EXAMPLE 2.1.6.

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

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

let G = ← r.

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

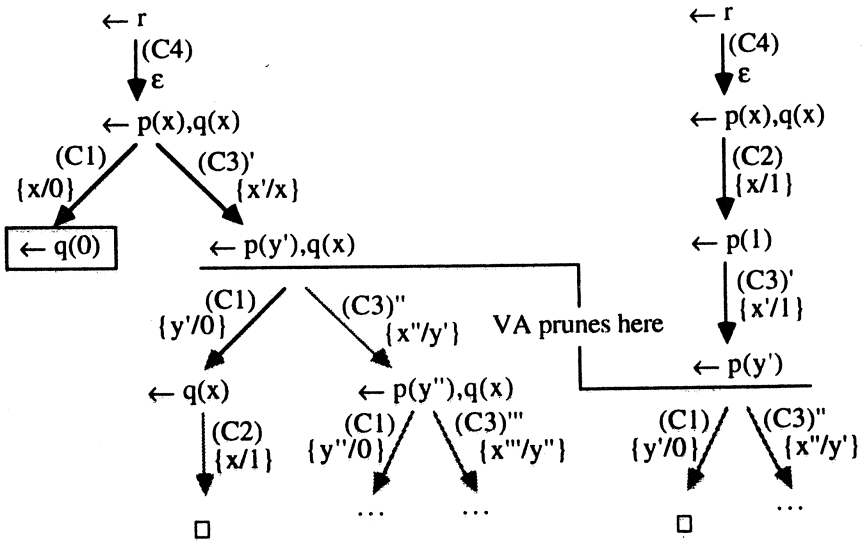


FIGURE 2.1.1

A detailed analysis shows why the goal G_3 = ←p(y') in the rightmost tree is pruned by the VA check. Clearly, a variant of p(y') occurs in an earlier goal: p(x) in G_1. So we take i = 1. In G_1, p(x) is not yet selected, so j > i. In fact

j = 2, for in G_2 the atom p(1), which is a further instantiated version of p(x), is selected. Indeed, p(y') is the result of resolving p(1). Therefore the derivation is pruned at G_3 by the VA check. (In this case, p(y') is the direct result of resolving p(1), but in general there may be any number of derivation steps between G_j and G_k.) □

Indeed, this loop check has not worked properly here: all successful derivations have been pruned. Clearly, this is an undesirable property for loop checks. On the other hand, all infinite derivations are pruned, as intended. In the next section, we shall give formal definitions of these and related properties of loop checks.

2.2. Properties of Loop Checks

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

Soundness and completeness

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

Even stronger, often we do not want to lose any individual solution. That is, if the original tree contains a successful branch, giving some computed answer 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 complete-ness of SLD-resolution with loop checking.

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

DEFINITION 2.2.1 (Soundness).

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

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

The following lemma is an immediate consequence of these definitions.

LEMMA 2.2.2. *Let L be a loop check.*

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 2.2.3 (Completeness).

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

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

Interpreters and loop checks

When a top-down interpreter is augmented with a loop check, we obtain a new interpreter. The soundness and completeness of this new interpreter depends on the soundness and completeness of the old one, as well as on the soundness and completeness of the loop check. However, these relations are not trivial. For

example, it is not true that adding a complete loop check to a complete interpreter yields a complete interpreter (recall that the notion of *soundness* of a loop check was introduced to ensure the *completeness* of the interpreter equipped with it).

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

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

PROOF. We know by the Soundness and strong completeness Theorem 1.3.2 that $\neg A \in CWA(P) \Leftrightarrow$ T contains no successful branches.

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

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

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

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

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

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

$\Leftarrow$ $f_L(T)$ contains a successful branch with a computed answer $G\sim\tau \leq G\sim\theta$, so T contains this branch as well. □

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

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

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

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

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

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

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

Comparing and combining loop checks

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

DEFINITION 2.2.8.

Let L_1 and L_2 be loop checks. L_1 is *stronger than* L_2 if for every program P and goal G, every SLD-derivation $D_2 \in L_2(P)$ of P∪{G} that is not itself contained in $L_1(P)$ has a proper initial subderivation $D_1 \in L_1(P)$. □

In other words, L_1 is stronger than L_2 if every SLD-derivation that is pruned by L_2 is also pruned by L_1. Note that the definition implies that every loop check is stronger than itself.

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

DEFINITION 2.2.9 (Sum of loop checks).
Let L_1 and L_2 be loop checks. For every program P, The *union* of L_1 and L_2 (denoted by L_1+L_2) is defined as: $(L_1+L_2)(P) = \text{Initials}(L_1(P) \cup L_2(P))$. □

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

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

i) L_1+L_2 is a loop check.

ii) $L_1+L_1 = L_1$.

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

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

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

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

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

PROOF. i)-vi). Straightforward.

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

REMARK 2.2.11. Even if L_1 and L_2 are sound, L_1+L_2 can be unsound. The following example shows that this is still true if L_1 and L_2 are both simple.

Let P = { p(x,1) ← p(x,0). (C1),
p(1,x) ← p(0,x). (C2),
p(0,0) ←. (C3) }

Consider the SLD-tree of P∪{←p(1,1)} in Figure 2.2.1.

Let L_1 be the set of variants of D_1 = (←p(1,1) ⇒(C1) ←p(1,0) ⇒(C2) ←p(0,0)) and let L_2 be the set of variants of D_2 = (←p(1,1) ⇒(C2) ←p(0,1) ⇒(C1) ←p(0,0)). Both L_1 and L_2 are sound: every SLD-tree that contains (a variant of) D_1 must contain (a variant of) D_2 and vice versa. Clearly L_1+L_2 is unsound. □

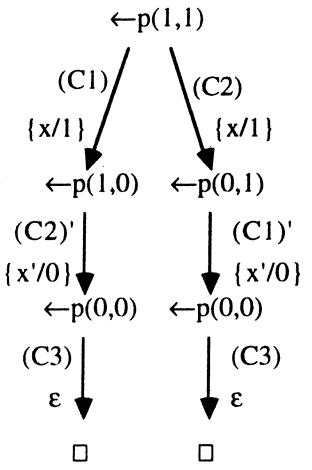


FIGURE 2.2.1.

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

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

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

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

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

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

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

iv) Every infinite SLD-derivation is pruned by L_2, so it is also pruned by L_1. □

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 existence of sound and complete loop checks

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

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

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

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

$$\leftarrow p(x_0),q(x_0)$$
$$\Downarrow$$
$$\leftarrow p(x_1),s(x_1,x_0),q(x_0)$$
$$\Downarrow$$
$$\leftarrow p(x_2),s(x_2,x_1),s(x_1,x_0),q(x_0)$$
$$\Downarrow$$
$$\leftarrow p(x_3),s(x_3,x_2),s(x_2,x_1),s(x_1,x_0),q(x_0)$$
$$\Downarrow$$
$$\ldots$$

FIGURE 2.2.2

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

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

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

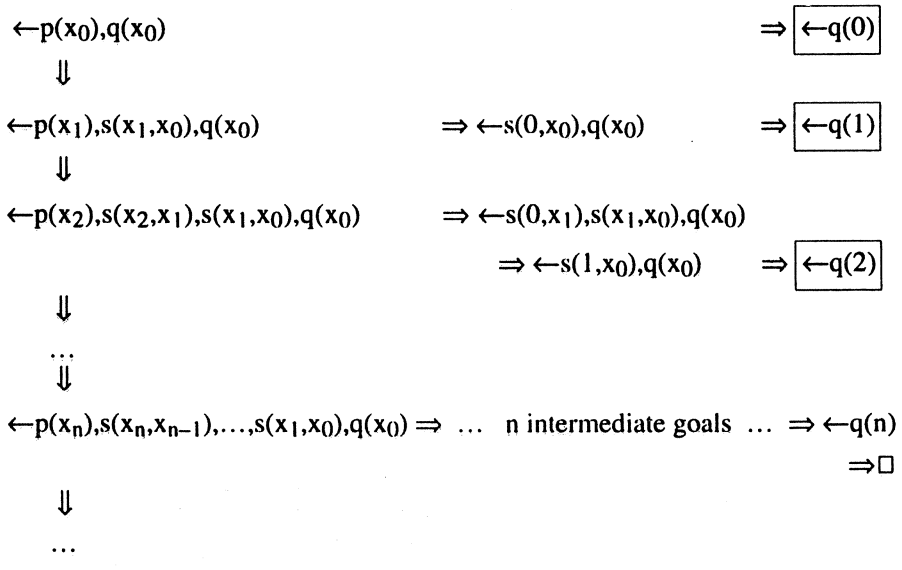


FIGURE 2.2.3

2.3. Nonsimple loop checks

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

Proof Tree Redundancy

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

DEFINITION 2.3.1 (Proof tree).

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

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

Figure 2.3.1 shows an example of this conversion of an SLD-derivation D via $D\theta_1 \ldots \theta_k$ (where $\theta_1 \ldots \theta_k = \{x/x', y/1, y'/0, z'/z\}$) into its proof tree.

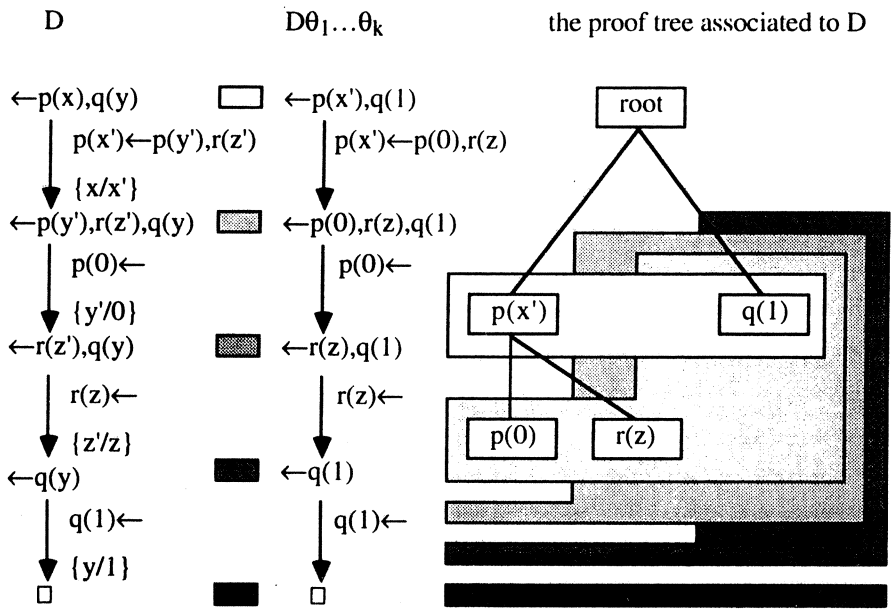


FIGURE 2.3.1

A goal in D corresponds to a 'horizontal layer' through the proof tree. A derivation step corresponds to the replacement of a node (representing the selected atom) in such a layer by its children. A simple induction argument shows that the length of an SLD-refutation equals the number of nodes in its proof tree, not counting the root.

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

DEFINITION 2.3.2 (Proof Tree Redundancy check).
For a function-free program P, the *Proof Tree Redundancy* check is defined as *PTR(P)* = Initials({D | for some G, D is an SLD-derivation of P∪{G} and for some predicate symbol p, a branch of the proof tree associated to D contains more p-atoms than there are ground p-atoms in $L_{P,G}$}). □

EXAMPLE 2.3.3.
In Theorem 2.2.13 we considered the program $P = \{s(i,i+1)\leftarrow. \mid 0 \leq i < n\} \cup \{p(0)\leftarrow.\ p(x)\leftarrow p(y),s(y,x).\ q(n)\leftarrow.\}$ and the resulting infinite SLD-derivation D of $P\cup\{\leftarrow p(x_0),q(x_0)\}$ shown in Figure 2.2.2. The proof tree associated to the first three steps of D is depicted in Figure 2.3.2.

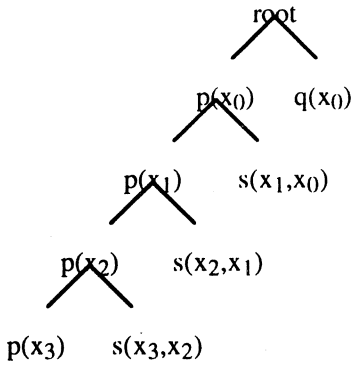


FIGURE 2.3.2

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

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

THEOREM 2.3.4. *PTR is shortening (so a fortiori sound).*
PROOF. Let P be a program, G a goal in L_P, σ a substitution and D a shortest derivation in $Succ(P,G,\sigma)$. We must show that D is not pruned by PTR. To this end we prove that for every predicate symbol p in L_P, no branch in the proof

tree T associated to D contains more p-atoms than there are ground p-atoms in $L_{P,G}$.

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

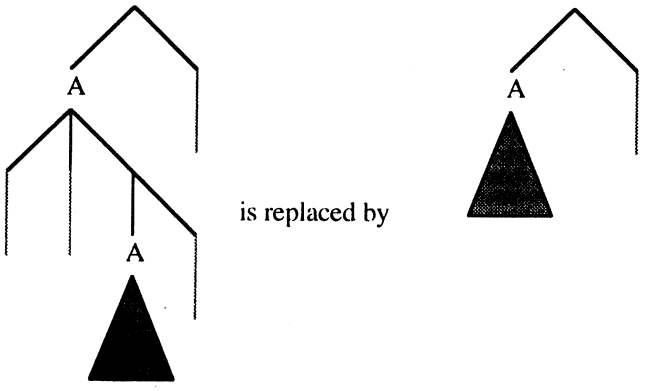


FIGURE 2.3.3

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

THEOREM 2.3.5. *PTR is complete.*

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

[1] Strictly speaking we have not defined the proof tree associated to an infinite derivation (in order to avoid an infinite composition of substitutions). Here it is sufficient to consider only the predicate symbols of the atoms, forgetting the arguments and substitutions altogether.

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

EXAMPLE 2.3.6.

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

$\{ s(i,i+1)\leftarrow. \mid 0 \le i < n \} \cup$

$\{ p(x_1,x_2,x_3) \leftarrow p(x_1,x_2,y),p(x_1,x_2,y),s(y,x_3).$

$p(x_1,x_2,0) \leftarrow p(x_1,y,n),p(x_1,y,n),s(y,x_2).$

$p(x_1,0,0) \leftarrow p(y,n,n),p(y,n,n),s(y,x_1).$

$p(0,0,0) \leftarrow. \}$

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

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

The strongest loop check

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

The aim of generating an SLD-tree is to find all solutions to a problem. For a function-free program, this set of solutions is finite. Once this set is known, a

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

DEFINITION 2.3.7 (STRONG check).
STRONG(P) = Initials({D = G ⇒ ... | for no σ, D is an initial fragment of a shortest derivation in Succ(P,G,σ)}). □

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

EXAMPLE 2.3.8.
Let P = { p(1) ←. (C1),
p(y) ← q(y,z),p(z). (C2),
q(w,0) ←. (C3),
q(0,1) ←. (C4) },
and let G = ←p(x).

Consider an SLD-tree of P∪{G} displayed in Figure 2.3.4. In Succ(P,G,{x/1}) a minimal length derivation has 2 goals, in Succ(P,G,{x/0}) a minimal length derivation has 4 goals and in Succ(P,G,ε) a minimal length derivation has 6 goals. These derivations are retained by STRONG in the considered SLD-tree, the others are pruned (at the horizontal lines in the figure). Among these are successful ones, but not minimal length successful ones. (The tree in Figure 2.3.4 is extended beyond the sixth level to show this effect.) □

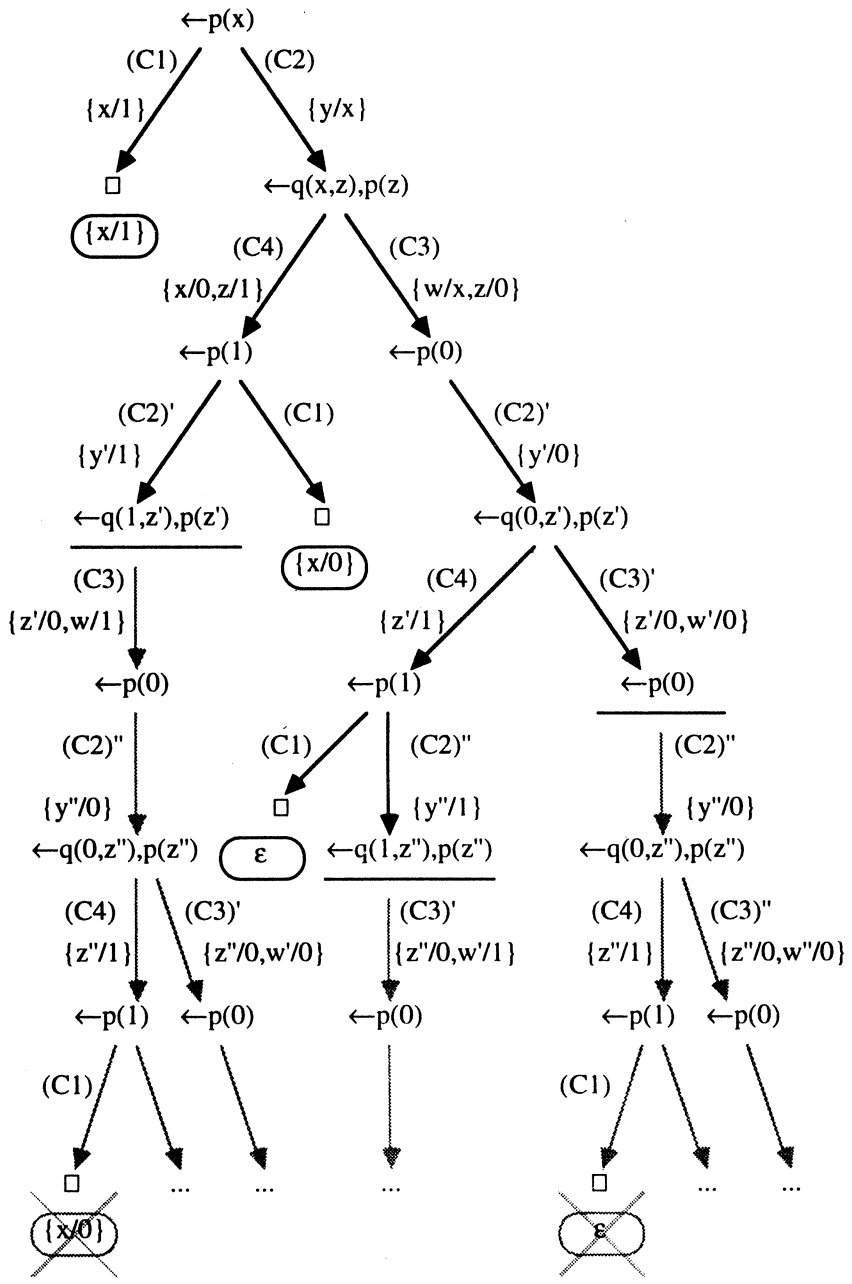


FIGURE 2.3.4

We can now prove the claims we made in the beginning of this section.

THEOREM 2.3.9. *For function-free programs:*

i) STRONG is a loop check.

ii) STRONG is shortening.

iii) STRONG is stronger than any shortening loop check.

iv) STRONG is complete.

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

1. Compute the set of correct answers for $P \cup \{G\}$ modulo renamings (e.g. bottom-up). Since P has no function symbols, this set is finite. Construct (breadth-first) an initial subtree of an SLD-tree of $P \cup \{G\}$ that contains (a proper subderivation of) D and for each correct answer a successful branch with a more general computed answer.
2. For each correct answer $G\sim\sigma$, mark the nodes of the shortest refutations in $Succ(P,G,\sigma)$.
3. Prune D at the first node in the tree that is not marked. If such a node does not exist, then D is a subderivation of a minimal length refutation.

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

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

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

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

3. Simple Loop Checks

3.1. Overview

In this chapter we study a number of intuitive simple loop checks. We can divide them into three groups, which are studied in Section 3.2, 3.3 and 3.4 respectively. These sections are all organized in the same way. First the loop checks of the group are defined and their effect is shown in an example. Also the relative strength of the loop checks within the group and in relation to the other groups is investigated.

Then we prove the appropriate soundness results. It appears that every loop check comes in two versions: a weakly sound one and a shortening one. Furthermore, the shortening version is always obtained from the weakly sound version in the same way: by adding a condition involving the computed answers generated so far. Because adding such a computed answer to a goal yields the corresponding resultant, we say that the weakly sound loop checks are *based on goals*, whereas the shortening ones are *based on resultants*. An immediate consequence of this construction is that the weakly sound versions are stronger than their shortening counterparts.

Finally we identify one or more natural classes of (function-free) programs for which the loop checks in the group are complete. The loop checks in all three groups appear to be complete for *restricted* programs without function symbols. Restricted programs allow a restricted form of recursion (hence the name).

All loop checks in this chapter are based on the same idea: a goal is pruned if it is 'sufficiently similar' to one of its ancestors. It is only in the notion of 'sufficiently similar' that the groups, and the loop checks within each group, differ. In the first group, the notion of 'sufficiently similar' is based on the *equality* of goals, respectively resultants. We call these loop checks *equality checks*.

The second group consists of loop checks based on the *inclusion* (or *subsumption*, see e.g. [CL]) of goals, respectively resultants. We call these loop checks *subsumption checks*. Subsumption checks are stronger than the corresponding equality checks. This makes it more difficult to establish their

soundness but opens a possibility for completeness for more classes of programs than just restricted ones.

We show that subsumption checks are complete for function-free programs in which no variables are introduced in the clause bodies (so called *nvi programs*), and for function-free programs in which each variable occurs at most once in every clause body (so called *svo programs*). These completeness theorems make use of a simple version of Kruskal's Tree Theorem, called Higman's Lemma [H]. While the use of this theorem to establish termination of term rewriting systems is well-known (see e.g. [DJ] or [Kr]), we have not encountered any applications of this theorem in the area of logic programming.

The third group of loop checks we study is based on a simple loop check introduced by Besnard [B]. These checks are directly inspired by the Variant of Atom check (Definition 2.1.5), but when comparing two atoms they take into account a certain context (a goal or a resultant) of those atoms. Therefore we call them *context checks*. We prove that for local selection rules (see Definition 1.2.9), the subsumption checks are stronger than the context checks.

As mentioned above, we prove that context checks are complete for function-free restricted programs. We also prove that they are complete for function-free nvi programs (a result that has been claimed in [B] without much proof) and for function-free svo programs.

The differences between the loop checks within a group are rather small. The most important one has been mentioned: the distinction between loop checks based on goals and those based on resultants. Another (independent) distinction is made between loop checks testing for *variants* and those testing for *instances*. For example, if the word 'variant' in Definition 2.1.5 is replaced by the word 'instance', we get the *Instance of Atom* (*IA*) check. As every variant of an expression is also an instance of it, an 'instance' check is stronger than the corresponding 'variant' check. (Thus, like VA, IA is not weakly sound.)

Finally, for the equality checks and subsumption checks, which deal with complete goals, one more (again independent) distinction is made. Whereas equality between atoms is unambiguous, equality between goals is much less clear. In SLD-derivations, we regard goals as lists, so both the number and the order of occurrences of atoms is important. However, we may also regard goals as multisets, where the order of the occurrences is unimportant. We might even consider regarding them as sets, which is customary in mathematical logic.

However that proves to be impractical: the goals ←A,A and ←A become indistinguishable, making the derivation step ←A,A $\Rightarrow_{A\leftarrow}$ ←A seem useless (which it is not). Regarding goals as sets in our loop checks would require regarding goals as sets in SLD-derivations, which would result in too many undesirable effects.

3.2. Equality Checks

In this section we study the equality checks in detail. First we give a definition of the weakly sound versions. Then we formally introduce the 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.

Definitions

In fact, we should give a definition for each equality check. This would yield eight 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 $=_L$; similarly $=_M$ for multisets. We begin with the weakly sound versions.

DEFINITION 3.2.1 (Equality checks based on Goals).

For Type ∈ {L,M}, the *Equals Variant/Instance of Goal*$_{Type}$ check is the set of SLD-derivations

EVG/EIG_{Type} = 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, $0 \le i < k$, there is a renaming/substitution τ such that $G_k =_{Type} G_i\tau$}). □

For example, EIG_M = 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, $0 \le i < k$, there is a substitution τ such that $G_k =_M G_i\tau$}).

The informal justification for these loop checks is similar to the one given for the VA check. 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 Gτ, then two cases arise. If there is no solution for Gτ, then pruning Gτ 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.

We shall prove later in this section that 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 3.2.3, 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.) It will appear that this condition works not only for EVG and EIG, but for all other loop checks studied in this chapter, as well.

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

DEFINITION 3.2.2 (Equality checks based on Resultants).
For Type $\in$ {L,M}, the *Equals Variant/Instance of Resultant*$_{Type}$ check is the set of SLD-derivations

EVR/EIR_{Type} = 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, $0 \le i < k$, there is a renaming/substitution τ such that $G_k =_{Type} G_i\tau$ and $G_0\theta_1\ldots\theta_k = G_0\theta_1\ldots\theta_i\tau$}). □

For example, EVR_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, $0 \le i < k$, there is a renaming 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 other variations (variants or instances, goals regarded as lists or as multisets) do not play a role.

EXAMPLE 3.2.3.

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

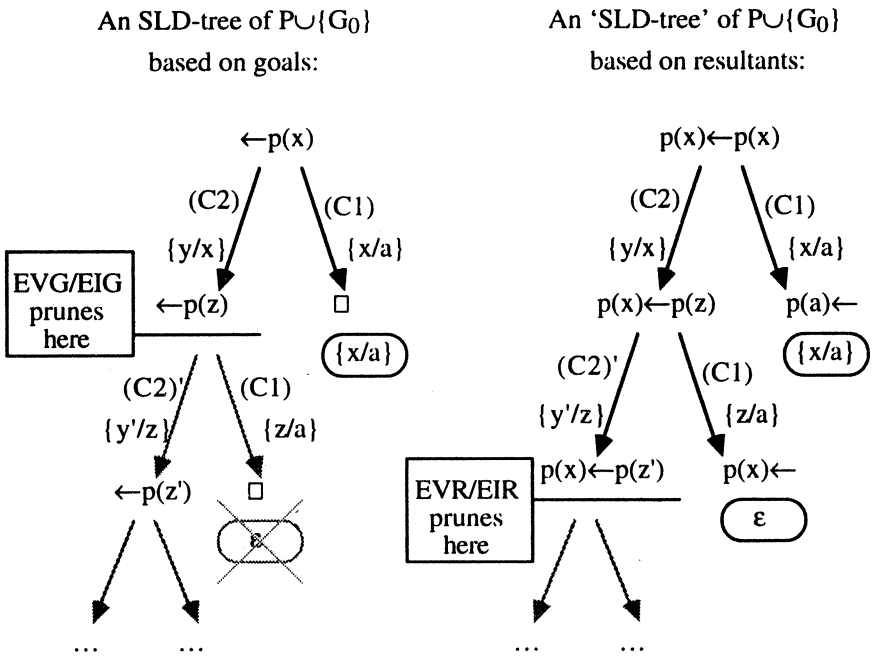


FIGURE 3.2.1

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 3.2.1, ←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. □

LEMMA 3.2.4. *All equality checks are simple loop checks.*
PROOF. Straightforward. □

Figure 3.2.2. shows the 'stronger than' relationships between the equality checks (and the VA and IA checks) and summarizes their properties. In this figure, an arrow $L_1 \rightarrow L_2$ means that L_2 is stronger than L_1. Proving these 'stronger than' relations is straightforward.

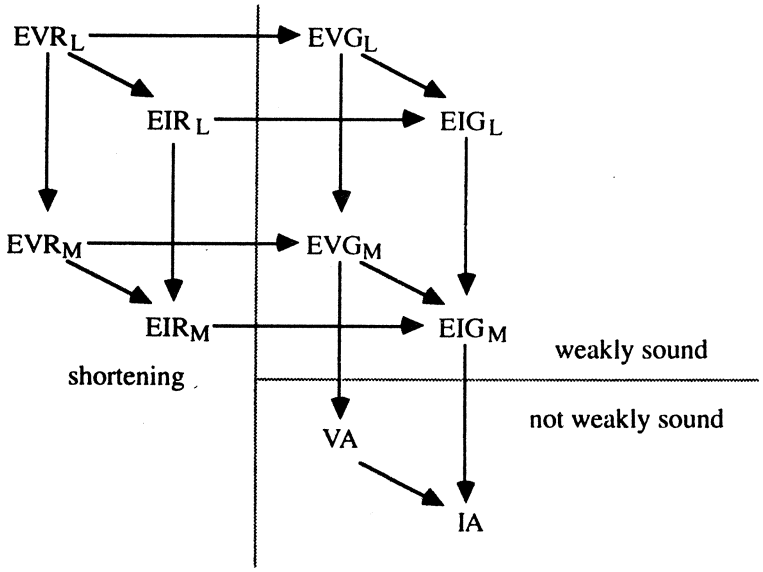


FIGURE 3.2.2

Soundness

We now prove that the equality checks based on resultants are shortening and that the equality checks based on goals are weakly sound. According to the Relative Strength Theorem 2.2.12 it is sufficient to focus on the strongest checks in both classes: the EIR_M and the EIG_M checks. The proof consists of two stages. The first stage, established in the following lemma, does not depend on the loop checking criterion and can therefore also be used to prove the soundness of the simple loop checks presented in the following sections.

LEMMA 3.2.5 (Shortening Condition). *Let L be a loop check.*
If, for every program P, goal G_0 and SLD-refutation $D = (G_0 \Rightarrow_{C_1,\theta_1} G_1 \Rightarrow \ldots \Rightarrow G_{k-1} \Rightarrow_{C_k,\theta_k} G_k \Rightarrow \ldots \Rightarrow_{C_m,\theta_m} \square)$ of $P \cup \{G_0\}$ $(0 < k \leq m)$:

[G_k is pruned by L] implies

[for some goal G_i $(0 \leq i < k)$ in D there exists an SLD-refutation $G_i \Rightarrow_{\sigma_1} \ldots \Rightarrow_{\sigma_n} \square$ of $P \cup \{G_i\}$ such that $n < m-i$],

then L is weakly sound.

Moreover, if also $G_0\theta_1 \ldots \theta_i\sigma_1 \ldots \sigma_n \leq G_0\theta_1 \ldots \theta_k\theta_{k+1} \ldots \theta_m$ is implied,

then L is shortening.

PROOF. First we focus on the weakly sound case. Let P be a program, G_0 a goal and T an SLD-tree of $P \cup \{G_0\}$. Suppose T contains a successful branch D $= (G_0 \Rightarrow_{C_1,\theta_1} G_1 \Rightarrow \ldots \Rightarrow G_{i-1} \Rightarrow_{C_i,\theta_i} G_i \Rightarrow \ldots \Rightarrow G_{k-1} \Rightarrow_{C_k,\theta_k} G_k \Rightarrow \ldots \Rightarrow_{C_m,\theta_m} \square)$ and suppose that D is pruned at G_k. We use here induction on m, i.e., we assume that for every successful branch B in T shorter than D, $f_L(T)$ contains either B or a successful branch shorter than B.

We prove that $f_L(T)$ contains a successful branch D' that is shorter than D. By assumption an SLD-derivation $D_1 = (G_i \Rightarrow_{\sigma_1} \ldots \Rightarrow_{\sigma_n} \square)$ of $P \cup \{G_i\}$ exists. Adding (a properly renamed variant of) D_1 to the initial part of D gives the derivation $D_2 = (G_0 \Rightarrow_{C_1,\theta_1} G_1 \Rightarrow \ldots \Rightarrow G_{i-1} \Rightarrow_{C_i,\theta_i} G_i \Rightarrow_{\tau_1} \ldots \Rightarrow_{\tau_n} \square)$. By the independence of the selection rule, T contains a branch D_3 such that $|D_3| = |D_2|$ and the computed answers of D_3 and D_2 are variants ([KM]). Since D_3 is shorter than D ($|D_3| = i+n < i+(m-i) = m = |D|$), by the induction hypothesis $f_L(T)$ contains either $D' = D_3$ or a successful branch D' shorter than D_3, which proves the claim.

For the shortening case, it remains to prove that $G_0\sigma' \leq G_0\theta_1 \ldots \theta_m$, where σ' is the computed answer substitution of D'. First we strengthen the induction hypothesis: for every successful branch B in T shorter than D giving a computed answer $G\sigma$, $f_L(T)$ contains either B or a successful branch shorter than B, giving a computed answer $G_0\sigma' \leq G_0\sigma$.

Then either since $D' = D_3$ or by the new induction hypothesis, and since the computed answers of D_3 and D_2 are variants, $G_0\sigma' \leq G_0\theta_1 \ldots \theta_i\tau_1 \ldots \tau_n \leq G_0\theta_1 \ldots \theta_i\sigma_1 \ldots \sigma_n \leq G_0\theta_1 \ldots \theta_m$. □

We now use this lemma to prove the desired result.

THEOREM 3.2.6. *i) The loop check EIR_M is shortening.*

ii) The loop check EIG_M is weakly sound.

PROOF. Let P be a program, G_0 a goal and $D = (G_0 \Rightarrow_{C_1,\theta_1} G_1 \Rightarrow \ldots \Rightarrow G_{i-1} \Rightarrow_{C_i,\theta_i} G_i \Rightarrow \ldots \Rightarrow G_{k-1} \Rightarrow_{C_k,\theta_k} G_k \Rightarrow \ldots \Rightarrow_{C_m,\theta_m} \square)$ an SLD-refutation of $P \cup \{G_0\}$ (where $0 \leq i < k \leq m$).

i) Assume that for some substitution τ: $G_k =_M G_i\tau$ and $G_0\theta_1 \ldots \theta_k = G_0\theta_1 \ldots \theta_i\tau$. So the SLD-derivation $G_i\tau \Rightarrow_{C_{k+1},\theta_{k+1}} \ldots \Rightarrow_{C_m,\theta_m} \square$ exists (the order of the atoms in $G_i\tau$ may differ from the order in G_k, so a different selection rule may be necessary). By the Lifting Lemma of [L] a derivation $G_i \Rightarrow_{\sigma_1} \ldots \Rightarrow_{\sigma_n} \square$ of $P \cup \{G_i\}$ exists, with $\sigma_1 \ldots \sigma_n \leq \tau\theta_{k+1} \ldots \theta_m$ ($n = m-k < m-i$). Now $G_0\theta_1 \ldots \theta_i\sigma_1 \ldots \sigma_n \leq G_0\theta_1 \ldots \theta_i\tau\theta_{k+1} \ldots \theta_m = G_0\theta_1 \ldots \theta_k\theta_{k+1} \ldots \theta_m$, hence the full condition of Lemma 3.2.5 is satisfied, so EIR_M is shortening.

ii) The additional condition $G_0\theta_1 \ldots \theta_k = G_0\theta_1 \ldots \theta_i\tau$ was only used to prove the additional shortening condition of Lemma 3.2.5. □

COROLLARY 3.2.7 (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.

PROOF. By Theorem 3.2.6 and the Relative Strength Theorem 2.2.12. □

Completeness

For completeness issues, it is sufficient to consider the weakest of the equality checks: the EVR_L check. We know that EVR_L is not complete—Theorem 2.2.13 presents a counterexample that holds for every simple loop check. For the EVR_L check this counterexample can even be simplified. What we need in fact is only the propositional structure of the recursive clause that is the 'core' of Theorem 2.2.13, i.e., we may remove its arguments.

EXAMPLE 3.2.8.

Let P = { p ← p,s. }.

Then for 'the' SLD-tree T of $P \cup \{\leftarrow p\}$ via the leftmost selection rule, $f_{EVR_L}(T)$ is infinite. Indeed, every descendant of the initial goal has one occurrence of s more than its parent goal, so it cannot be a variant of any of its ancestors. □

Obviously, the problem is that the atom p in the goal is allowed to generate infinitely many s-atoms, which are never selected, thereby making the goal wider and wider. We now introduce a class of programs for which this phenomenon cannot occur and we prove that EVR_L is complete for these programs. 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 3.2.9.
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: $clp(p) = \{q \mid (p,q) \in D_P^* \text{ and } (q,p) \in D_P^*\}$. □

DEFINITION 3.2.10 (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 *nonrecursive*.
A clause $H \leftarrow A_1,\ldots,A_n$ is *restricted w.r.t. P* if $A_1,\ldots,A_{n-1}$ are nonrecursive.
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 'transitive closure' program given in the introduction is restricted. Note also that programs of which all clauses have a body with at most one atom are restricted. 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.

We now prove that EVR_L is complete w.r.t. the leftmost selection rule for restricted programs. First we demonstrate an interesting feature of restricted programs, namely that in each SLD-derivation via the leftmost selection rule, goals have a number of atoms which is bounded by a value depending only on the program and the initial goal. Then we show that this implies that modulo the 'being a variant of' relation, the number of possible goals in such an SLD-derivation is finite.

In the rest of this section, P is a function-free restricted program and G is a goal in L_P. The maximum length of the goals in a derivation of $P \cup \{G\}$ can be computed by means of the following *weight*-function, which is defined on goals and predicate symbols (by mutual induction).

DEFINITION 3.2.11.

The function *weight* is defined as follows:

i) for a goal $G = \leftarrow A_1,\ldots,A_n$ $(n \geq 1)$,

$\text{weight}(G) = \max\{\text{weight}(\text{rel}(A_i)) + n - i \mid i = 1,\ldots,n\}$

($n - i$ is the number of atoms to the right of A_i in G);

ii) for a predicate symbol p, weight(p) =

$\max(\ \{\ \text{weight}(\leftarrow A_1,\ldots,A_n) \mid$

$A \leftarrow A_1,\ldots,A_n \in P,\ n > 0,\ \text{rel}(A) \in \text{cl}_P(p),\ \text{rel}(A_n) \notin \text{cl}_P(p)\ \} \cup$

$\{\ 1 + \text{weight}(\leftarrow A_1,\ldots,A_{n-1}) \mid$

$A \leftarrow A_1,\ldots,A_n \in P,\ n > 1,\ \text{rel}(A) \in \text{cl}_P(p),\ \text{rel}(A_n) \in \text{cl}_P(p)\ \} \cup$

$\{\ 1\ \}\ ).$ □

Note that in the definition of weight(p), clauses of the form $A \leftarrow B$, with $\text{cl}(\text{rel}(A)) = \text{cl}(\text{rel}(B))$ are not considered—they do not affect the length of goals appearing in a derivation. Moreover, if the predicate symbols p and q are mutually dependent, then weight(p) = weight(q).

The fact that P is restricted ensures that the weight-function is well-defined: if weight(p) is defined in terms of weight(q), then $(q,p) \notin D_P^*$, hence weight(q) is not defined in terms of weight(p). Intuitively, the weight of a goal G majorizes the length of all goals which appear in an SLD-derivation of $P \cup \{G\}$ using leftmost selection rule. More precisely, we have the following lemmas (recall that $|G|$ denotes the length of G).

LEMMA 3.2.12. *Let $G = \leftarrow A_1,\ldots,A_n$ $(n \geq 1)$. Then $|G| \leq weight(G)$.*

PROOF. $weight(G) \geq weight(rel(A_1)) + n - 1 \geq n = |G|$. □

LEMMA 3.2.13. *Let $G \Rightarrow_C H$ be a derivation step w.r.t. P where the leftmost atom of G is selected. Then $weight(G) \geq weight(H)$.*

PROOF. Since the weight of a goal depends only on the predicates appearing in it, and not on the arguments of these predicates, we prove this fact for the case of programs written in propositional logic. Let $G = \leftarrow A_1,\ldots,A_n$; then $weight(G) = \max\{weight(A_i) + n - i \mid i = 1,\ldots,n\}$, and let $C = A_1 \leftarrow B_1,\ldots,B_m$.

Then the goal $H = \leftarrow B_1,\ldots,B_m,A_2,\ldots,A_n$ and therefore

$$\begin{aligned} weight(H) &= \max(\{weight(B_i) + m + n - 1 - i \mid i = 1,\ldots,m\} \\ &\quad \cup \{weight(A_{i-m+1}) + m + n - 1 - i \mid i = m+1,\ldots,m+n-1\}) \\ &= \max(\{weight(B_i) + m + n - 1 - i \mid i = 1,\ldots,m\} \\ &\quad \cup \{weight(A_i) + n - i \mid i = 2,\ldots,n\}). \end{aligned}$$

Two cases arise.

i) $weight(H) = \max\{weight(A_i) + n - i \mid i = 2,\ldots,n\}$.

Then clearly $weight(H) \leq weight(G)$.

ii) $weight(H) = \max\{weight(B_i) + m + n - 1 - i \mid i = 1,\ldots,m\}$ (hence $m > 0$).

We show that in this case $weight(H) \leq weight(A_1) + n - 1 \leq weight(G)$. Subtracting $n - 1$, it suffices to show that

$weight(A_1) \geq \max\{weight(B_i) + m - i \mid i = 1,\ldots,m\}$. Again two cases arise.

iia) $(B_m,A_1) \notin Dp^*$. Then because of the existence of C, $weight(A_1) \geq weight(\leftarrow B_1,\ldots,B_m) = \max\{weight(B_i) + m - i \mid i = 1,\ldots,m\}$.

iib) $(B_m,A_1) \in Dp^*$. Then

$$\begin{aligned} weight(A_1) &\geq 1 + weight(\leftarrow B_1,\ldots,B_{m-1}) = \\ &1 + \max\{weight(B_i) + m - 1 - i \mid i = 1,\ldots,m-1\} = \\ &\max\{weight(B_i) + m - i \mid i = 1,\ldots,m-1\}. \end{aligned}$$

Also $weight(B_m) + m - m = weight(A_1)$, since $B_m \in clp(A_1)$. This proves the claim that $\max\{weight(B_i) + m - i \mid i = 1,\ldots,m\} \leq weight(A_1)$. □

COROLLARY 3.2.14. *Let $D = (G_0 \Rightarrow G_1 \Rightarrow G_2 \Rightarrow \ldots \Rightarrow G_i \Rightarrow \ldots)$ be an SLD-derivation via the leftmost selection rule. Then for every goal G_i in D: $|G_i| \leq weight(G_0)$.*

PROOF. By induction on i. The induction basis is provided by Lemma 3.2.12, the induction step by Lemma 3.2.13. □

So weight(G_0) is indeed the desired maximum length of goals occurring in any SLD-derivation of $P \cup \{G_0\}$.

We now present a formalization of the 'being a variant of' relation on resultants. Our presentation here is more general than needed to prove the completeness of the equality checks. However, we need these results in full generality to prove the completeness of the subsumption checks and the context checks.

DEFINITION 3.2.15.
Let X be a set of variables. We define the relation $\sim_X$ on resultants as $R_1 \sim_X R_2$ if for some renaming ρ, $R_1\rho = R_2$ and for every $x \in X$, $x\rho = x$. Now let G be a goal and let $k \geq 1$. Then the relation $\sim_{X,G,k}$ stands for the restriction of the relation $\sim_X$ to resultants $S_1 \leftarrow S_2$ such that $\leftarrow S_1$ is an instance of G and $|\leftarrow S_2| \leq k$. □

LEMMA 3.2.16. *For every set of variables X, goal G and $k \geq 1$, $\sim_{X,G,k}$ is an equivalence relation.*
PROOF. Straightforward. □

For a resultant R, the equivalence class of R w.r.t. the relation $\sim_{X,G,k}$ will be denoted as $[R]_{X,G,k}$, or just [R] whenever X, G and k are clear from the context. The following lemma is crucial for our considerations.

LEMMA 3.2.17. *Suppose that the language L has no function symbols and finitely many predicate symbols and constants. Then for every finite set of variables X, goal G and $k \geq 1$, the relation $\sim_{X,G,k}$ has only finitely many equivalence classes.*
PROOF. Let #c be the number of constants in L, #p the number of predicate symbols, #x the number of variables in X and let #a be the maximum arity of the predicate symbols in L. Let G be a goal of the form $\leftarrow p_1(\ldots),p_2(\ldots),\ldots,p_m(\ldots)$ with $m \geq 1$ and let #v be the number of distinct variables in G.

A resultant in an equivalence class of $\sim_{X,G,k}$ is then of the form $p_1(\ldots),p_2(\ldots),\ldots,p_m(\ldots) \leftarrow q_1(\ldots),q_2(\ldots),\ldots,q_n(\ldots)$ with $0 \leq n \leq k$. An equivalence class of $\sim_{X,G,k}$ is completely described by the predicate symbols

$q_1,\ldots,q_n$, the arguments of $p_1,\ldots,p_m$ (in accordance with G) and the arguments of $q_1,\ldots,q_n$.

The number of arguments that must be specified in this resultant is #v for $p_1,\ldots,p_m$, plus at most n·#a for $q_1,\ldots,q_n$. For every argument we may choose either a constant, a variable from X or another (fresh) variable. However, we need at most #v + n·#a different fresh variables. Therefore the choice of the arguments is limited to $(\#c + \#x + \#v + n\cdot\#a)^{\#v+n\cdot\#a}$ possibilities.

Since for a fixed n, the choice of the predicate symbols $q_1,\ldots,q_n$ is limited to $\#p^n$ possibilities, we have at most $\sum_{n=0}^{k} \#p^n\cdot(\#c + \#x + \#v + n\cdot\#a)^{\#v + n\cdot\#a}$ equivalence classes of $\sim_{X,G,k}$. □

We can now prove the desired theorem.

THEOREM 3.2.18. *The loop check EVR_L is complete w.r.t. the leftmost selection rule for function-free restricted programs.*

PROOF. Let P be a function-free restricted program and let G_0 be a goal in L_P. Let k = weight(G_0). Consider an infinite SLD-derivation $D = (G_0 \Rightarrow_{C_1,\theta_1} G_1 \Rightarrow\ldots\Rightarrow G_{i-1} \Rightarrow_{C_i,\theta_i} G_i \Rightarrow\ldots)$ of $P\cup\{G_0\}$. By Corollary 3.2.14 for every $i \geq 0$: $|G_i| \leq k$. Every goal G_i is a goal in L_P and hence every resultant $G_0\theta_1\ldots\theta_i\leftarrow G_i$ belongs to an equivalence class of $\sim_{\emptyset,G_0,k}$. Since L_P satisfies the conditions of Lemma 3.2.17, $\sim_{\emptyset,G_0,k}$ has only finitely many equivalence classes, so for some $i \geq 0$ and $j > i$, $G_0\theta_1\ldots\theta_i\leftarrow G_i$ and $G_0\theta_1\ldots\theta_j\leftarrow G_j$ are variants. This implies that D is pruned by EVR_L. □

COROLLARY 3.2.19 (Equality Completeness). *All equality checks are complete w.r.t. the leftmost selection rule for function-free restricted programs.*

PROOF. By Theorem 3.2.18 and the Relative Strength Theorem 2.2.12. □

Now combining Corollary 2.2.6 and Corollary 2.2.7 with the Equality Soundness Corollary 3.2.7 and the Equality Completeness Corollary 3.2.19, we conclude that all equality checks lead to an implementation of CWA for function-free restricted programs. Moreover, a depth-first interpreter augmented with any of the equality checks based on resultants yields an implementation of query processing for these programs.

3.3. Subsumption Checks

Definitions

Similar to the equality checks, there are eight subsumption checks. We shall define them by means of two parametrized definitions, again trusting that the reader is willing to understand our notation. The inclusion relation between goals regarded as lists is denoted by $\subseteq_L$; similarly $\subseteq_M$ for multisets. Note: $G_1 \subseteq_L G_2$ if all elements of G_1 occur in the same order in G_2; they need not occur on adjacent positions. For example, (p,r) $\subseteq_L$ (p,q,r).

DEFINITION 3.3.1 (Subsumption checks based on Goals).
For Type $\in$ {L,M}, the *Subsumes Variant/Instance of Goal*$_{Type}$ check is the set of SLD-derivations

SVG/SIG_{Type} = 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 with $G_k \supseteq_{Type} G_i\tau$}). □

DEFINITION 3.3.2 (Subsumption checks based on Resultants).
For Type $\in$ {L,M}, the *Subsumes Variant/Instance of Resultant*$_{Type}$ check is the set of SLD-derivations

SVR/SIR_{Type} = 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 with $G_k \supseteq_{Type} G_i\tau$ and $G_0\theta_1 \ldots \theta_k = G_0\theta_1 \ldots \theta_i\tau$}). □

LEMMA 3.3.3. *All subsumption checks are simple loop checks.*
PROOF. Straightforward. □

The following example shows the differences between the behaviour of various subsumption checks and the equality checks.

EXAMPLE 3.3.4.
Let P = { p(y) ← p(0),r(y). (C1),
p(0) ←. (C2),
q(1) ←. (C3),
r(z) ← q(z),p(w). (C4) },

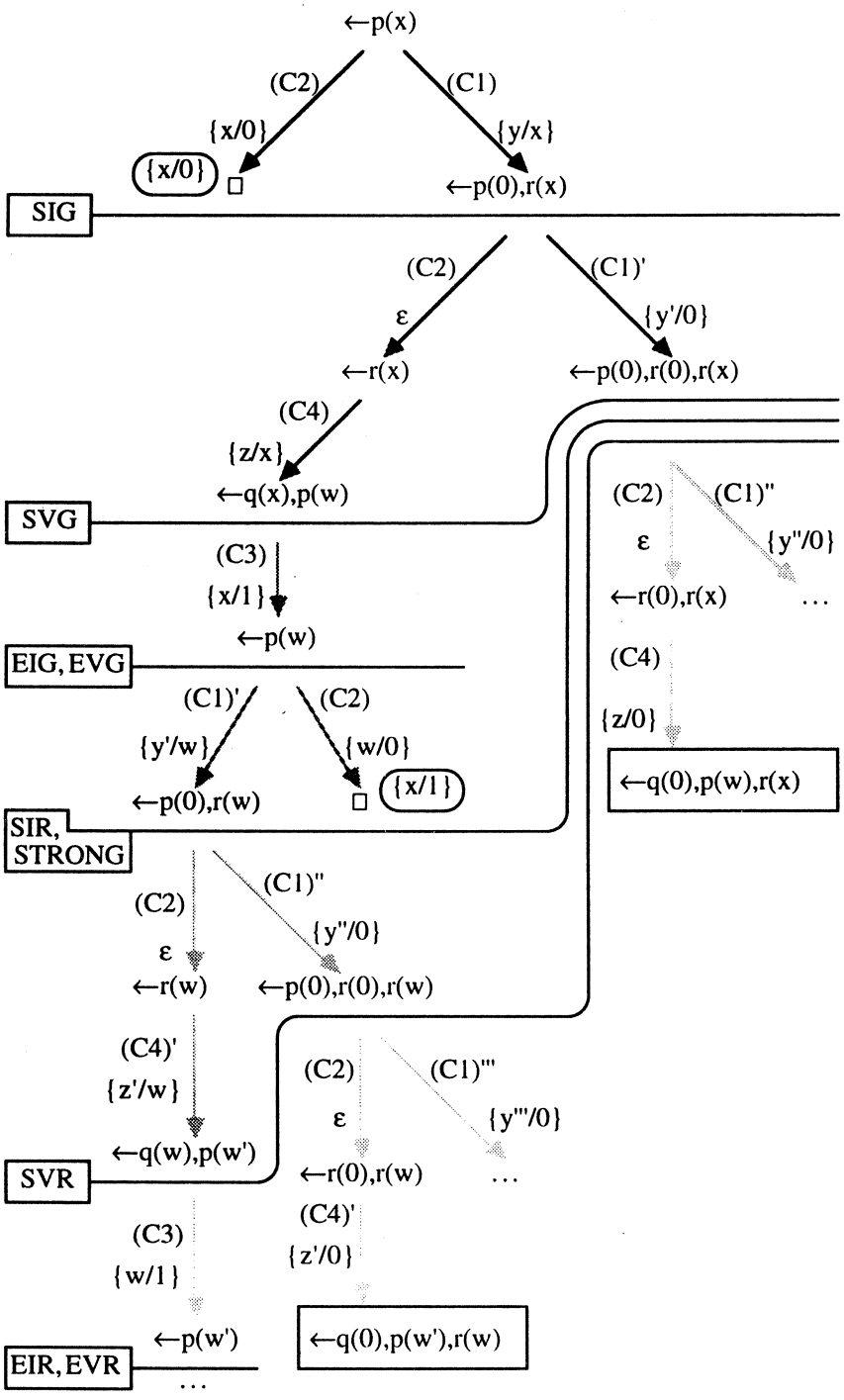

←p(x)
(C2)
{x/0}
{x/0}
(C1)
{y/x}
←p(0),r(x)
SIG
(C2)
ε
←r(x)
(C1)'
{y'/0}
←p(0),r(0),r(x)
(C4)
{z/x}
←q(x),p(w)
SVG
(C3)
{x/1}
←p(w)
EIG, EVG
(C1)'
{y'/w}
←p(0),r(w)
(C2)
{w/0}
{x/1}
SIR, STRONG
(C2)
ε
←r(w)
(C1)"
{y"/0}
←p(0),r(0),r(w)
(C4)'
{z'/w}
←q(w),p(w')
SVR
(C3)
{w/1}
←p(w')
EIR, EVR
(C2)
ε
←r(0),r(w)
(C1)'''
{y'''/0}
(C4)'
{z'/0}
←q(0),p(w'),r(w)
(C2)
ε
←r(0),r(x)
(C1)"
{y"/0}
(C4)
{z/0}
←q(0),p(w),r(x)

FIGURE 3.3.1

and let G = ←p(x).

Figure 3.3.1 shows an SLD-tree of P∪{G} using the leftmost selection rule. It also shows how this tree is pruned by different loop checks. First we explain the behaviour of the loop checks with respect to this tree. Then we shall make some generalizing comments on this behaviour. In this example, the distinction between list versus multiset based loop checks does not play a role.

Starting at the root, the first loop check that prunes the tree is the SIG check. It prunes the goal ←p(0),r(x), because it contains p(0), an instance of p(x). Following the leftmost infinite branch two steps down, the SVG check prunes the goal ←q(x),p(w), because it contains p(w), a variant of p(x). One step later, the atom q(x) is resolved, so the EIG and EVG checks prune the goal ←p(w) for the same reason.

However, the loop checks based on resultants do not yet prune the tree. The computed answer substitution built up so far maps x to x after the first three steps and to 1 later on. This is clearly different from the substitutions {x/0} and {x/v}, which are used to show that p(0) respectively p(w) are an instance respectively a variant of p(x).

Now the derivation repeats itself, but with x replaced by w. Therefore the loop checks based on resultants prune the tree during this second phase, exactly where the corresponding loop checks based on goals pruned during the first phase.

The side branch that is obtained by repeatedly applying the first clause (and corresponding side branches later on) is pruned by the subsumption checks at the goal ←p(0),r(0),r(x). This goal contains the previous goal ←p(0),r(x). Therefore both the resultant based and the goal based loop checks prune this goal. In contrast, the equality checks do not prune this infinite branch because the goals in it become longer in every derivation step (analogously to Example 3.2.8).

The loop checks based on goals all prune the solution {x/1}, so they are not sound. Among these loop checks, the SIG check prunes as soon as possible for a weakly sound loop check. Conversely, the SIR check prunes this tree as soon as possible for a shortening loop check. So on this tree, it behaves exactly like STRONG, which exhibits such a behaviour by definition. □

Another example shows that there can be a nontrivial difference between the behaviour of subsumption checks based on list subsumption and those based on multiset subsumption.

EXAMPLE 3.3.5.

Let P = { $p(x) \leftarrow p(y),s(x),r(y)$. }. (Note the similarity between this clause and the clause $p(x) \leftarrow p(y),s(y,x)$ in Theorem 2.2.13.)

Let G = $\leftarrow p(x_0),q(x_0)$.

An SLD-derivation (and SLD-tree) of $P \cup \{G\}$ via the leftmost selection rule is depicted in Figure 3.3.2. This infinite SLD-derivation is pruned by the SVR_M check at the goal $\leftarrow p(x_2),s(x_1),r(x_2),s(x_0),r(x_1),q(x_0)$, since a variant of an earlier goal, namely $(\leftarrow p(x_1),s(x_0),r(x_1),q(x_0))\{x_1/x_2,x_2/x_1\}$, is 'multiset-contained' in it.

$$\leftarrow p(x_0),q(x_0)$$
$$\Downarrow$$
$$\leftarrow p(x_1),s(x_0),r(x_1),q(x_0)$$
$$\Downarrow$$
$$\leftarrow p(x_2),s(x_1),r(x_2),s(x_0),r(x_1),q(x_0)$$
$$\Downarrow$$
$$\leftarrow p(x_3),s(x_2),r(x_3),s(x_1),r(x_2),s(x_0),r(x_1),q(x_0)$$
$$\Downarrow$$
$$\ldots$$

FIGURE 3.3.2

However, this derivation is *not* pruned by the SVR_L check, nor by the stronger SIG_L check. For, assume that the SIG_L check prunes this derivation at the goal $G_k = \leftarrow p(x_k),s(x_{k-1}),r(x_k),s(x_{k-2}),r(x_{k-1}),\ldots,s(x_0),r(x_1),q(x_0)$, because $G_i\tau = (\leftarrow p(x_i),s(x_{i-1}),r(x_i),s(x_{i-2}),r(x_{i-1}),\ldots,s(x_0),r(x_1),q(x_0))\tau$, an instance of an earlier goal G_i, is list-contained in it.

Clearly, the presence of the q-atoms in $G_i\tau$ and G_k requires $x_0\tau = x_0$. So the atom $s(x_0)\tau$ in $G_i\tau$ corresponds to the atom $s(x_0)$ in G_k. Then, because $G_i\tau$ is *list*-contained in G_k, $r(x_1)\tau$ can only correspond to $r(x_1)$, the only atom between $s(x_0)$ and $q(x_0)$. Therefore $x_1\tau = x_1$. Using induction, we can derive $x_2\tau = x_2$,

..., $x_i\tau = x_i$. However, the presence of the p-atoms in $G_i\tau$ and G_k requires $x_i\tau = x_k$. Since $i < k$, this is a contradiction. So the assumption that the SIG_L check prunes the derivation is refuted. □

The above examples *suggest* some 'stronger than' relationships (although an example can only prove the absence of such a relationship). Figure 3.3.3 shows such relationships between the subsumption checks, the equality checks, VA and IA, extending Figure 3.2.2.

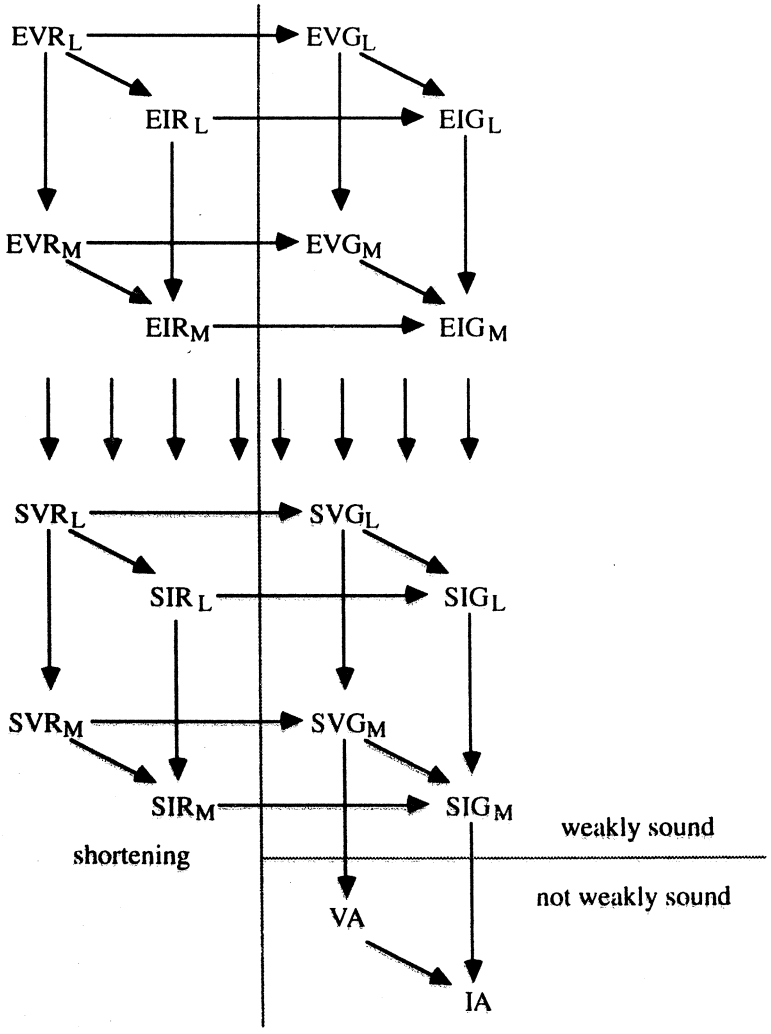


FIGURE 3.3.3

The arrows between the 'cubes' mean that every subsumption check is stronger than the corresponding equality check in the other 'cube'. So the structure of 'stronger than' relations between equality checks and subsumption checks is a four-dimensional hypercube. Again, proving these 'stronger than' relations is straightforward.

Soundness

To prove the desired soundness results, we prove that the SIR_M check is shortening and that the SIG_M check is weakly sound, since these are the strongest loop checks based on resultants, respectively goals, in our scheme. First we need the following lemma.

LEMMA 3.3.6. *Let P be a program and τ a substitution. Let G_1 and G_2 be goals such that $G_2\tau \subseteq_M G_1$. Suppose D_1 is an SLD-refutation of $P \cup \{G_1\}$ with computed answer substitution σ_1. Then there exists an SLD-refutation D_2 of $P \cup \{G_2\}$ with a computed answer substitution σ_2 such that $|D_2| \leq |D_1|$ and $\sigma_2 \leq \tau\sigma_1$.*

PROOF. Let $D = (G_1 \Rightarrow_{C_1,\theta_1} \dots \Rightarrow_{C_n,\theta_n} \square)$ and let $C_{n_1},\dots,C_{n_m}$ be those clauses from $C_1,\dots,C_n$ that are used (directly or indirectly) to resolve atoms belonging to $G_2\tau$, with $1 \leq n_1 < \dots < n_m \leq n$. Then there exists an unrestricted (in the sense of [L]) SLD-derivation $G_2\tau\theta_1\dots\theta_{n_1-1} \Rightarrow_{C_{n_1},\theta_{n_1}\dots\theta_{n_2-1}} \cdots \Rightarrow_{C_{n_m},\theta_{n_m}\dots\theta_n} \square$. Now apply the Mgu Lemma and the Lifting Lemma of [L]. □

We can now prove the desired theorem.

THEOREM 3.3.7. *i) The SIR_M check is shortening.*

ii) The SIG_M check is weakly sound.

PROOF. Let P be a program, G_0 a goal and $D = (G_0 \Rightarrow_{C_1,\theta_1} G_1 \Rightarrow \dots \Rightarrow G_{i-1} \Rightarrow_{C_i,\theta_i} G_i \Rightarrow \dots \Rightarrow G_{k-1} \Rightarrow_{C_k,\theta_k} G_k \Rightarrow \dots \Rightarrow_{C_m,\theta_m} \square)$ an SLD-refutation of $P \cup \{G_0\}$ (where $0 \leq i < k \leq m$).

i) Assume that for some substitution τ: $G_k \supseteq_M G_i\tau$ and $G_0\theta_1\dots\theta_k = G_0\theta_1\dots\theta_i\tau$. Since $G_k \Rightarrow_{C_{k+1},\theta_{k+1}} \cdots \Rightarrow_{C_m,\theta_m} \square$, by Lemma 3.3.6 an SLD-derivation $G_i \Rightarrow_{\sigma_1} \dots \Rightarrow_{\sigma_n} \square$ of $P \cup \{G_i\}$ exists, with $\sigma_1\dots\sigma_n \leq \tau\theta_{k+1}\dots\theta_m$ ($n \leq m-k < m-i$). $G_0\theta_1\dots\theta_i\sigma_1\dots\sigma_n \leq G_0\theta_1\dots\theta_i\tau\theta_{k+1}\dots\theta_m = G_0\theta_1\dots\theta_k\theta_{k+1}\dots\theta_m$, hence the full condition of Lemma 3.2.5 is satisfied, so SIR_M is shortening.

ii) The additional condition $G_0\theta_1\ldots\theta_k = G_0\theta_1\ldots\theta_i\tau$ was only used to prove the additional shortening condition of Lemma 3.2.5. □

COROLLARY 3.3.8 (Subsumption Soundness).

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

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

PROOF. By Theorem 3.3.7 and the Relative Strength Theorem 2.2.12. □

Completeness

We now shift our attention to completeness issues. From the results of the previous section we can immediately deduce the following result.

COROLLARY 3.3.9 (Subsumption Completeness 1). *All subsumption checks are complete w.r.t. the leftmost selection rule for function-free restricted programs.*

PROOF. By the Equality Completeness Corollary 3.2.19 and the Relative Strength Theorem 2.2.12. □

However, the subsumption checks are stronger than the corresponding equality checks. So we can try to find other classes of programs for which the subsumption checks are complete. We know that the subsumption checks are not complete for all programs, not even for all function-free programs. For P = {p(x)←p(y),s(y,x)}, a derivation of P∪{←p(x),q(x)} is not pruned by any of the subsumption checks, as was shown in Theorem 2.2.13.

A close analysis of the proof of this theorem shows that the problem is caused by three characteristics of this clause occurring simultaneously, namely:

1. A new variable, y, is introduced by a recursive atom, p(y).
2. There is a relation between this new variable, y, and an old variable, x, namely via the atom s(y,x).
3. The recursive atom p(y) is selected before the 'relating' atom s(y,x).

It appears that, in order to obtain the completeness of the subsumption checks, it is enough to prevent any of these events. Clearly, the use of restricted programs and the leftmost selection rule prevents the third event. We now

introduce two new classes of programs, preventing the first and the second event, respectively.

DEFINITION 3.3.10 (Nvi program).
A clause C is *non-variable introducing* (in short *nvi*) if every variable that appears in the body of C also appears in the head of C.
An *nvi program* is a program of which every clause is nvi. □

DEFINITION 3.3.11 (Svo program).
A clause C has the *single variable occurrence* property (in short *is svo*) if in the body of C, no variable occurs more than once.
An *svo program* is a program of which every clause is svo. □

Clearly, in nvi programs the first event cannot occur, whereas in svo programs the second event is prevented. We would rather have used the terminology *right-linear* instead of svo, which is common in the area of term rewriting systems. However, in the area of deductive databases this term is already in use for a completely different notion.

EXAMPLE 3.3.12.
The following program is an nvi program and an svo program, but not a restricted program. It computes in the relation 'add' the sum of two two-digit binary numbers (the first four arguments of 'add'); this sum is a three-digit binary number, stored in the last three arguments of 'add'.

ADD = { add(0,0, x,y, 0,x,y) ←.
add(x,y, 0,0, 0,x,y) ←.
add(x,y, x,y, x,y,0) ←.
add(x_1,y_1, x_2,y_2, z,x_3,y_3) ← add(0,y_1, 0,y_2, 0,0,y_3),
add(0,x_1, 0,x_2, 0,z,x_3).
add(x_1,1, x_2,1, 1,0,0) ← add(0,x_1, 0,x_2, 0,0,1). }.

The first three clauses are evidently correct; every addition of the form 0X + 0Y is taken care of by them. The fourth clause deals with the case where adding the last digits of both numbers does not give a carry (ensured by the first atom in the body). The fifth clause deals with the case where there is such a carry. Only the

case $x_1 \neq x_2$ (or equivalently, $x_1 + x_2 = 1$) has to be considered there: if $x_1 = x_2$ then the third clause applies.

Note that this program yields infinite derivations that are not pruned by any of the equality checks. Indeed, starting with the goal $\leftarrow$ add(0,y_1,0,y_2,0,0,y_3), the first recursive clause applies, giving the goal $\leftarrow$ add(0,y_1,0,y_2,0,0,y_3), add(0,0,0,0,0,0,0). Repeatedly selecting add(0,y_1,0,y_2,0,0,y_3) and applying the first recursive clause yields an infinite derivation containing goals of increasing length, which is not pruned by any of the equality checks. □

We now prove that the weakest of the subsumption checks, the SVR_L check, is complete for function-free nvi programs. To this end we use the following (weakened) version of Kruskal's Tree Theorem, called Higman's Lemma. (See [H]; for a formulation of the full version of Kruskal's Tree Theorem, see [DJ] or [Kr].)

LEMMA 3.3.13 (Higman's Lemma). *Let w_0, w_1, w_2, ... be an infinite sequence of (finite) words over a finite alphabet Σ.*
Then for some i and $k > i$, $w_i \subseteq_L w_k$. □

In order to prove that the SVR_L check is complete for function-free nvi programs, we prove that that every *normal* SLD-derivation (see Definition 1.2.4) of a function-free nvi program (and an arbitrary goal) does not introduce new variables. Then we prove that in the absence of function symbols, infinite derivations in which no new variables are introduced are pruned by the SVR_L check.

DEFINITION 3.3.14.
An SLD-derivation $D = (G_0 \Rightarrow_{C_1,\theta_1} G_1 \Rightarrow \ldots)$ is *non-variable introducing* (in short *nvi*) if $var(G_0) \supseteq var(G_1) \supseteq var(G_2) \supseteq \ldots$. □

LEMMA 3.3.15. *Let P be a function-free nvi program and let G_0 be a goal in L_P. Let D be a normal infinite SLD-derivation of $P \cup \{G_0\}$. Then D is an infinite nvi derivation.*

PROOF. Suppose that $D = (G_0 \Rightarrow_{C_1,\theta_1} G_1 \Rightarrow_{C_2,\theta_2} G_2 \Rightarrow \ldots)$. We prove that for every $i > 0$: $var(G_i) \subseteq var(G_{i-1})$. By Corollary 1.2.7 $var(G_{i-1}\theta_i) \subseteq var(G_{i-1})$,

so it is sufficient to prove that $var(G_i) \subseteq var(G_{i-1}\theta_i)$. Let A be the selected atom in G_{i-1} and let S_1 be the rest of G_{i-1}. Let $C_i = H \leftarrow S_2$, then $G_i = \leftarrow (S_1,S_2)\theta_i$. Let $x \in var(G_i)$. Two cases arise.

- x is introduced by G_{i-1}, that is $x \in var(S_1\theta_i)$. Then $x \in var(G_{i-1}\theta_i)$.
- x is introduced by C_i, that is $x \in var(S_2\theta_i)$. Then, since P is an nvi program, $x \in var(H\theta_i)$. θ_i is a unifier of H and A, so $x \in var(A\theta_i) \subseteq var(G_{i-1}\theta_i)$. □

LEMMA 3.3.16. *In the absence of function symbols, every infinite nvi SLD-derivation is pruned by SVR_L.*

PROOF. Let $D = (G_0 \Rightarrow_{C_1,\theta_1} G_1 \Rightarrow \ldots)$ be an infinite nvi SLD-derivation. We take for Σ the set of equivalence classes of $\sim_{var(G_0),G_0,1}$ as defined in Definition 3.2.15. By Lemma 3.2.17, Σ is finite. To apply Higman's Lemma 3.3.13 we represent for $j \geq 0$ a goal $G_j = \leftarrow A_{1j},\ldots,A_{n_jj}$ (or rather the corresponding resultant $G_0\theta_1\ldots\theta_j \leftarrow G_j$) as the word $[G_0\theta_1\ldots\theta_j \leftarrow A_{1j}],\ldots,[G_0\theta_1\ldots\theta_j \leftarrow A_{n_jj}]$ over Σ. (Recall that for a resultant R, [R] denotes its equivalence class.) The sequence of representations of G_0, G_1, G_2, ... yields an infinite sequence of words w_0, w_1, w_2, ... over Σ.

Now by Higman's Lemma 3.3.13, for some j and $k > j$: $[G_0\theta_1\ldots\theta_j \leftarrow A_{1j}]$, $\ldots,[G_0\theta_1\ldots\theta_j \leftarrow A_{n_jj}] \subseteq_L [G_0\theta_1\ldots\theta_k \leftarrow A_{1k}],\ldots,[G_0\theta_1\ldots\theta_k \leftarrow A_{n_kk}]$. So by the definition of $\sim_{var(G_0),G_0,1}$, there exist renamings $\rho_1,\ldots,\rho_{n_j}$ which do not act on the variables of G_0 such that $(G_0\theta_1\ldots\theta_j \leftarrow A_{1j})\rho_1,\ldots,(G_0\theta_1\ldots\theta_j \leftarrow A_{n_jj})\rho_{n_j} \subseteq_L (G_0\theta_1\ldots\theta_k \leftarrow A_{1k}),\ldots,(G_0\theta_1\ldots\theta_k \leftarrow A_{n_kk})$.

As D is nvi, $var(G_j) \subseteq var(G_0)$ and therefore the renamings ρ_h do not act on the atoms A_{ij} of G_j ($1 \leq h,i \leq n_j$). Thus $G_j = G_j\rho_1 \subseteq_L G_k$ and $G_0\theta_1\ldots\theta_j\rho_1 = G_0\theta_1\ldots\theta_k$. So D is pruned by SVR_L. □

THEOREM 3.3.17. *The SVR_L loop check is complete for function-free nvi programs.*

PROOF. By Lemma 1.2.5 (every SLD-derivation has a normal variant), Lemma 3.3.3 (SVR_L is closed under variants), and Lemma 3.3.15 and 3.3.16. □

COROLLARY 3.3.18 (Subsumption Completeness 2). *All subsumption checks are complete for function-free nvi programs.*

PROOF. By Theorem 3.3.17 and the Relative Strength Theorem 2.2.12. □

We now prove that the SVR_L check (and hence all subsumption checks) are complete for function-free svo programs.

LEMMA 3.3.19. *Let P be a function-free svo program and let G_0 be a goal in L_P. Let $D = (G_0 \Rightarrow_{C_1,\theta_1} G_1 \Rightarrow_{C_2,\theta_2} G_2 \Rightarrow \ldots)$ be a normal SLD-derivation of $P \cup \{G_0\}$. Then for every goal G_i $(i \geq 0)$: if x occurs more than once in G_i, then $x \in var(G_0)$.*

PROOF. By induction. For i = 0, the claim is trivial.

Now suppose that x occurs more than once in G_i (i > 0).

Let A be the selected atom in G_{i-1} and let S_1 be the rest of G_{i-1}. Let $C_i = H \leftarrow S_2$, then $G_i = \leftarrow (S_1,S_2)\theta_i$. There are two ways in which we can obtain a variable x occurring more than once in G_i.

1. A variable y occurs more than once in (S_1,S_2) and $y\theta_i = x$.

By standardizing apart, $var(S_1) \cap var(S_2) = \emptyset$, so y occurs either only in S_1 or only in S_2. Since C_i is svo, y does not occur more than once in S_2. Therefore y occurs more than once in S_1. Then by the induction hypothesis, $y \in var(G_0)$. Consequently $x = y\theta_i \in var(G_0\theta_i) \subseteq var(G_0)$ (by Corollary 1.2.7).

2. There are variables y_1, y_2 in (S_1,S_2) such that $y_1\theta_i = y_2\theta_i = x$ and $y_1 \neq y_2$.

In this case $y_1, y_2 \in var(A,H)$, since θ_i is relevant. If $y_1 \in var(S_1)$, then by standardizing apart $y_1 \notin var(H)$, so $y_1 \in var(A)$. Therefore y_1 occurs more than once in G_i (in A and in S_1), and we can apply the induction hypothesis and Corollary 1.2.7 again. Since the same argument holds if $y_2 \in var(S_1)$, only the case $y_1, y_2 \in var(S_2)$ is left. In this case, since $y_1, y_2 \in var(A,H)$, and by standardizing apart $y_1, y_2 \in var(H)$. Since $y_1\theta_i = y_2\theta_i = x$, the sets

$Z_1 = \{z \in var(A) \mid z$ occurs in A at the position of an occurrence of y_1 in H$\}$ and

$Z_2 = \{z \in var(A) \mid z$ occurs in A at the position of an occurrence of y_2 in H$\}$ are not disjoint. (Otherwise, a more general unifier of A and H than θ_i would exist, mapping y_1 to an element of Z_1 and y_2 to an element of Z_2.) Let $z \in Z_1 \cap Z_2$. z occurs at least twice in A, so $z \in var(G_0)$. Thus $x = z\theta_i \in var(G_0\theta_i) \subseteq var(G_0)$.□

We can now prove the desired theorem.

THEOREM 3.3.20. *The SVR_L loop check is complete for function-free svo programs.*

PROOF. Let P be a function-free svo program and let G_0 be a goal in L_P. Let $D = (G_0 \Rightarrow_{C_1,\theta_1} G_1 \Rightarrow_{C_2,\theta_2} G_2 \Rightarrow \ldots)$ be an infinite SLD-derivation of $P \cup \{G_0\}$. By Lemma 1.2.5 and 3.3.3 we may assume that D is normal.

Again, we take for Σ the set of equivalence classes of $\sim_{var(G_0),G_0,1}$ as defined in Definition 3.2.15. By Lemma 3.2.17, Σ is finite. To apply Higman's Lemma 3.3.13 we represent a goal $G_j = A_{1j},\ldots,A_{n_jj}$ in D as the word $w_j = [G_0\theta_1\ldots\theta_j \leftarrow A_{1j}],\ldots,[G_0\theta_1\ldots\theta_j \leftarrow A_{n_jj}]$ over Σ. The sequence of representations of $G_0, G_1, G_2, \ldots$ yields an infinite sequence of words $w_0, w_1, w_2, \ldots$ over Σ.

Now by Higman's Lemma 3.3.13, for some j and $k > j$: $[G_0\theta_1\ldots\theta_j \leftarrow A_{1j}], \ldots,[G_0\theta_1\ldots\theta_j \leftarrow A_{n_jj}] \subseteq_L [G_0\theta_1\ldots\theta_k \leftarrow A_{1k}],\ldots,[G_0\theta_1\ldots\theta_k \leftarrow A_{n_kk}]$. So there are renamings $\rho_1,\ldots,\rho_{n_j}$ such that $(G_0\theta_1\ldots\theta_j \leftarrow A_{1j})\rho_1,\ldots,(G_0\theta_1\ldots\theta_j \leftarrow A_{n_jj})\rho_{n_j} \subseteq_L (G_0\theta_1\ldots\theta_k \leftarrow A_{1k}),\ldots,(G_0\theta_1\ldots\theta_k \leftarrow A_{n_kk})$.

We now construct a renaming ρ. Consider the set $X = var(G_j) - var(G_0)$. By Lemma 3.3.19 a variable $x \in X$ occurs at most once in G_j; if x occurs in A_{ij}, then we define $x\rho = x\rho_i$. In order to make ρ a renaming ρ maps (one-to-one) the variables of $X\rho - X$ to the variables of $X - X\rho$; ρ is the identity mapping on variables outside $X \cup X\rho$. Since, by the definition of $\sim_{var(G_0),G_0,1}$, the renamings ρ_i do not act on variables in $var(G_0)$, $x \in X \cup X\rho$ implies $x \notin var(G_0)$. Hence ρ does not act on the variables in $var(G_0)$, so $G_j\rho \subseteq_L G_k$. By Corollary 1.2.7 $var(G_0\theta_1\ldots\theta_j) \subseteq var(G_0)$, thus $G_0\theta_1\ldots\theta_j\rho = G_0\theta_1\ldots\theta_j$. So $G_j\rho \subseteq_L G_k$ and $G_0\theta_1\ldots\theta_j\rho = G_0\theta_1\ldots\theta_k$, hence D is pruned by SVR_L. □

COROLLARY 3.3.21 (Subsumption Completeness 3). *All subsumption checks are complete for function-free svo programs.*

PROOF. By Theorem 3.3.20 and the Relative Strength Theorem 2.2.12. □

Now combining Corollary 2.2.6 and Corollary 2.2.7 with the Subsumption Soundness Corollary 3.3.8 and the Subsumption Completeness Corollaries 3.3.9, 3.3.18 and 3.3.21, we conclude that all subsumption checks lead to an implementation of CWA for restricted programs, nvi programs and svo programs without function symbols. Moreover, the subsumption checks based on resultants also lead to an implementation of query processing for these programs.

3.4. Context Checks

Definitions

The explanation for the fact that the Variant (Instance) of Atom check is not (weakly) sound is that it does not take into account the context of an atom. However, whereas ←p(x) and ←p(y) are variants, the existence of a refutation of ←p(y),q(x) does not imply the existence of a refutation of ←p(x),q(x). To remedy this problem we should keep track of the links between the variables in the atom and those in the rest of the goal.

Roughly speaking, the IA check prunes a derivation as soon as a goal G_k occurs that contains an instance $A\tau$ 'produced' by an atom A that occurred in an earlier goal G_i. But when a variable occurs both inside and outside of A in G_i, we should not prune the derivation if this link has been altered. Such a variable x in G_i is substituted by $x\theta_{i+1}\ldots\theta_k$ when G_k is reached. Therefore τ and $\theta_{i+1}\ldots\theta_k$ should agree on x. This leads us to a loop check introduced by Besnard [B].

DEFINITION 3.4.1 (Context checks based on Goals).

The *Variant/Instance Context check based on Goals* is the set of SLD-derivations

$CVG/CIG = Initials(\{D \mid D = (G_0 \Rightarrow_{C_1,\theta_1} G_1 \Rightarrow \ldots \Rightarrow G_{k-1} \Rightarrow_{C_k,\theta_k} G_k)$ such that for some i and j, $0 \le i \le j < k$, there is a renaming/substitution τ such that for some atom A in G_i: $A\tau$ appears in G_k as the result of an attempt to resolve $A\theta_{i+1}\ldots\theta_j$, the further instantiated version of A in G_j and for every variable x that occurs both inside and outside of A in G_i, $x\theta_{i+1}\ldots\theta_k = x\tau\}$). □

Besnard describes the condition on the substitutions as follows: 'When $A\tau$ is substituted for $A\theta_{i+1}\ldots\theta_k$ in $G_i\theta_{i+1}\ldots\theta_k$, this should give an instance of G_i.' We show that this formulation is equivalent to ours. Let $G_i = (A,S)$, that is A occurs in G_i and S is the list of other atoms in G_i. Then $(A\tau,S\theta_{i+1}\ldots\theta_k)$ should be an instance of (A,S), say $(A\sigma,S\sigma)$.

$$\text{Clearly, } x\sigma = \begin{cases} x\tau & \text{for } x \in \text{var}(A), \\ x\theta_{i+1}\ldots\theta_k & \text{for } x \in \text{var}(S), \end{cases}$$

so for $x \in \text{var}(A) \cap \text{var}(S)$, $x\tau = x\theta_{i+1}\ldots\theta_k$.

The following example clarifies the use of the context checks.

EXAMPLE 3.4.2.

We use the program P and the goal G of Example 2.1.6 and apply the CIG check on two SLD-trees of P∪{G}, via the leftmost and rightmost selection rule, respectively. This yields the trees in Figure 3.4.1.

The goal G_3 = ←p(y') in the rightmost tree that was incorrectly pruned by the VA check, is not pruned by the CIG check. Certainly, p(y') is the result of resolving p(1) in G_2, the further instantiated version of p(x) in G_1. But replacing $p(x)\theta_2\theta_3$ by p(y') in $G_1\theta_2\theta_3$ yields ←p(y'),q(1), which is *not* an instance of ←p(x),q(x). □

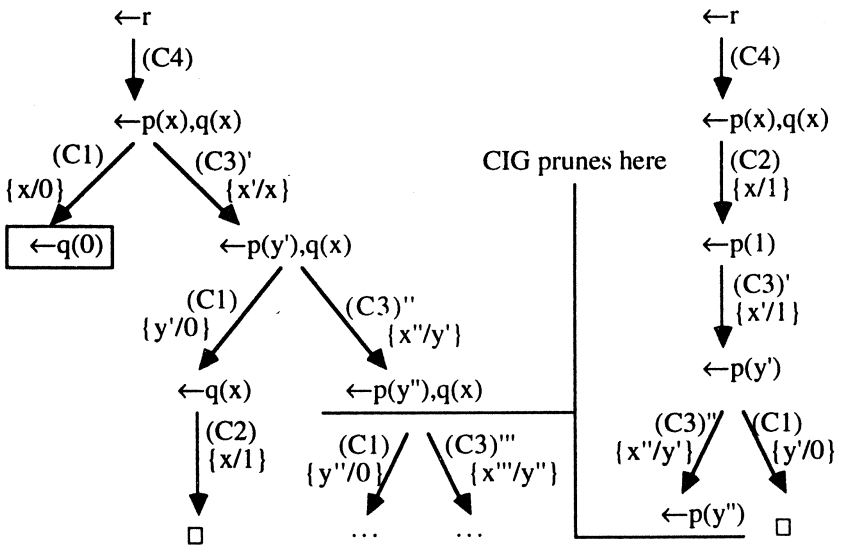


FIGURE 3.4.1

CLAIM 3.4.3. *CVG and CIG are weakly sound simple loop checks.*

PROOF. Proving that CVG and CIG are simple loop checks is straightforward. Besnard proves in [B] that CIG is weakly sound. From this it follows that the weaker CVG check is also weakly sound. See also Theorem 3.4.6. □

In Example 3.2.3, the context checks act exactly in the same way as the corresponding equality checks. This shows that CVG and CIG are not sound. Again we can obtain sound, even shortening, versions by using resultants instead of goals.

DEFINITION 3.4.4 (Context checks based on Resultants).
The *Variant/Instance Context check based on Resultants* is the set of SLD-derivations

CVR/CIR = 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 and j, $0 \leq i \leq j < k$, there is a renaming/substitution τ such that $G_0\theta_1 \ldots \theta_k = G_0\theta_1 \ldots \theta_i\tau$ and for some atom A in G_i: $A\tau$ appears in G_k as the result of an attempt to resolve $A\theta_{i+1} \ldots \theta_j$, the further instantiated version of A in G_j and for every variable x that occurs both inside and outside of A in G_i, $x\theta_{i+1} \ldots \theta_k = x\tau$}). □

Using Besnard's phrasing, the conditions on the substitutions can be summarized as: 'When $A\tau$ is substituted for $A\theta_{i+1} \ldots \theta_k$ in the resultant $R_i\theta_{i+1} \ldots \theta_k$, this should give an instance of R_i.'

LEMMA 3.4.5. *CVR and CIR are simple loop checks.*
PROOF. Straightforward. □

Soundness
Now we prove that the CIR check is shortening. From this it follows that the weaker loop check CVR is also shortening.

THEOREM 3.4.6. *The CIR check is shortening.*
PROOF. Let P be a program, G_0 a goal and D = ($G_0 \Rightarrow_{C_1,\theta_1} G_1 \Rightarrow \ldots \Rightarrow G_{i-1} \Rightarrow_{C_i,\theta_i} G_i \Rightarrow \ldots \Rightarrow G_{k-1} \Rightarrow_{C_k,\theta_k} G_k \Rightarrow \ldots \Rightarrow_{C_m,\theta_m} \square$) an SLD-refutation of $P \cup \{G_0\}$ (where $0 \leq i < k \leq m$).

Assume that D is pruned by CIR, that is for some substitution τ: $G_i = \leftarrow(A,S_i)$, $G_k = \leftarrow(A\tau,S_k)$, $A\tau$ descends from A, $S_i\theta_{i+1} \ldots \theta_k = S_i\tau$ and $G_0\theta_1 \ldots \theta_k = G_0\theta_1 \ldots \theta_i\tau$. (Here $G = \leftarrow(A,S)$ means: A occurs in G and $\leftarrow S$ is obtained by removing A from G.)

Then $\leftarrow S_i \subseteq_M G_i$ and $\leftarrow A\tau \subseteq_M G_k$. Since $G_i \Rightarrow_{C_{i+1},\theta_{i+1}} \cdots \Rightarrow_{C_k,\theta_k} G_k \Rightarrow_{C_{k+1},\theta_{k+1}} \cdots \Rightarrow_{C_m,\theta_m} \square$, by Lemma 3.3.6 we have SLD-refutations D_1 of $P \cup \{\leftarrow S_i\}$ and D_2 of $P \cup \{\leftarrow A\}$, where the computed answer substitution of D_1, $\tau_1 \leq \theta_{i+1} \ldots \theta_m$ and the computed answer substitution of D_2, $\tau_2 \leq \tau\theta_{k+1} \ldots \theta_m$. Say $\tau_2\gamma = \tau\theta_{k+1} \ldots \theta_m$. Now we combine D_1 and D_2 into an unrestricted SLD-

refutation of $P \cup \{\leftarrow(A,S_i)\}$: first resolve A as in D_2; the goal $S_i\tau_2$ remains. Replacing the last mgu μ of this derivation by $\mu\gamma$, this remaining goal becomes $S_i\tau_2\gamma = S_i\tau\theta_{k+1}\ldots\theta_m = S_i\theta_{i+1}\ldots\theta_k\theta_{k+1}\ldots\theta_m$. From Lemma 8.5 of [L] and the existence of D_1 it follows that $P \cup \{\leftarrow S_i\theta_{i+1}\ldots\theta_m\}$ can be refuted indeed, giving a computed answer $S_i\theta_{i+1}\ldots\theta_m$. The Mgu Lemma of [L] shows that the combined unrestricted refutation can be turned into a real SLD-refutation D_3 of $P \cup \{\leftarrow(A,S_i)\}$ giving a computed answer $G_0\theta_1\ldots\theta_i\tau_3 \leq G_0\theta_1\ldots\theta_i\tau\theta_{k+1}\ldots\theta_m = G_0\theta_1\ldots\theta_k\theta_{k+1}\ldots\theta_m$.

Since $A\tau$ descends from A, an inspection of the proof of Lemma 3.3.6 shows that every derivation step in D_1 and D_2 has a corresponding derivation step in the tail ($G_i \Rightarrow\ldots\Rightarrow \square$) of D. This tail consists of m–i derivation steps. On the other hand, at least one step in this tail has no corresponding step in D_1 or D_2: the step in which $A\theta_{i+1}\ldots\theta_j$ is selected. Hence $|D_3| = |D_1| + |D_2| < m–i$. Now apply Lemma 3.2.5. □

COROLLARY 3.4.7 (Context Soundness).

i) The context checks based on resultants are shortening. A fortiori they are sound.

ii) The context checks based on goals are weakly sound.

PROOF. By Theorem 3.4.6 and the Relative Strength Theorem 2.2.12. Note that omitting the considerations about computed answer substitutions from the proof of Theorem 3.4.6 yields a proof for ii), i.e., for Claim 3.4.3. □

For derivations via local selection rules (see Definition 1.2.9), e.g. the leftmost and rightmost selection rule, a different soundness proof exists, based on the relative strength of the context checks.

LEMMA 3.4.8. *The SIG_L check is stronger than the CIG check and the SIR_L check is stronger than the CIR check w.r.t. local selection rules.*

PROOF. Suppose $D = (G_0 \Rightarrow_{C_1,\theta_1} G_1 \Rightarrow\ldots\Rightarrow_{C_k,\theta_k} G_k)$ is pruned at G_k by the CIG check. We show that D is pruned by the SIG_L check at G_k (or earlier).

We have an atom A in G_i, $A\theta_{i+1}\ldots\theta_j$ in G_j as the selected atom and $A\tau$ as the result of resolving $A\theta_{i+1}\ldots\theta_j$. Let $G_i = (S,A,T)$, where S consists of those atoms in G_i that are completely resolved between G_i and G_j. The use of a local selection rule yields $G_j = (A\theta_{i+1}\ldots\theta_j, T\theta_{i+1}\ldots\theta_j)$ and $G_k = (A\tau,U,T\theta_{i+1}\ldots\theta_k)$

(U consists of the atoms in G_k other than $A\tau$ that are the result of resolving $A\theta_{i+1}...\theta_j$). Finally, if $x \in var(A) \cap var(S,T)$ then $x\theta_{i+1}...\theta_k = x\tau$.

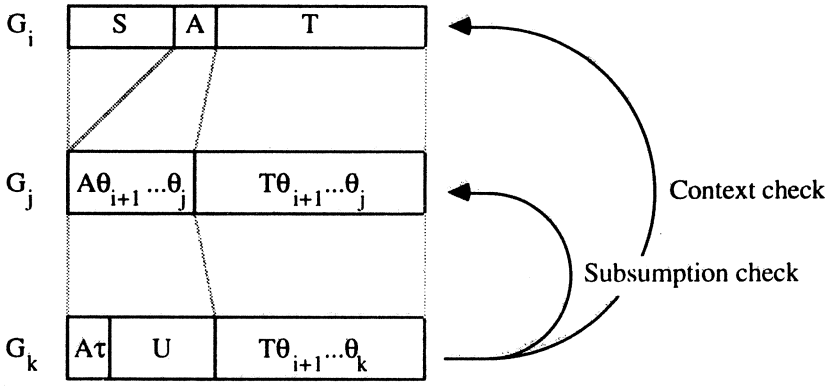


FIGURE 3.4.2

We show that for some substitution $\bar{\sigma}$, $G_j\sigma \subseteq_L G_k$. We define σ as follows:

$$x\sigma = \begin{cases} x & \text{if } x \notin var(G_j), \\ x\theta_{j+1}...\theta_k & \text{if } x \in var(G_j) - var(A), \\ x\tau & \text{if } x \in var(G_j) \cap var(A). \end{cases}$$

We show that (i) $A\theta_{i+1}...\theta_j\bar{\sigma} = A\tau$ and that (ii) $T\theta_{i+1}...\theta_j\bar{\sigma} = T\theta_{i+1}...\theta_k$.

(i) Let $x \in var(A)$, then $x \in var(G_i)$. We prove that $x\theta_{i+1}...\theta_j\bar{\sigma} = x\tau$.

If $x \in var(G_j)$, then (by Lemma 1.2.1) $x\theta_{i+1}...\theta_j = x$, hence $x\theta_{i+1}...\theta_j\bar{\sigma} = x\bar{\sigma} = x\tau$.

If $x \notin var(G_j)$ then $x\theta_{i+1}...\theta_j \neq x$, hence $x \in var(S)$. So $x\theta_{i+1}...\theta_k = x\tau$. Moreover, for every $y \in var(x\theta_{i+1}...\theta_j)$, either $y \in var(S)$ or y is introduced by $C_{i+1},...,C_j$, i.e., $y \notin var(G_i)$, in particular $y \notin var(A)$. In both cases $y\bar{\sigma} = y\theta_{j+1}...\theta_k$ (notice that $y \in var(x\theta_{i+1}...\theta_j) \subseteq var(A\theta_{i+1}...\theta_j) \subseteq var(G_j)$). So $x\theta_{i+1}...\theta_j\bar{\sigma} = x\theta_{i+1}...\theta_k = x\tau$.

(ii) Now let $y \in var(T\theta_{i+1}...\theta_j)$. We prove that $y\bar{\sigma} = y\theta_{j+1}...\theta_k$.

First note that for some $x \in var(T)$: $y \in var(x\theta_{i+1}...\theta_j)$.

If $x \notin var(S)$, then $x = x\theta_{i+1}...\theta_j = y$, so $y \in var(T)$, hence $y\sigma = y\theta_{j+1}...\theta_k$.

If $x \in var(S)$, then again either $y \in var(S)$ or $y \notin var(A)$, and in both cases $y\bar{\sigma} = y\theta_{j+1}...\theta_k$.

If D is pruned by the CIR check, then we also have that $G_0\theta_1 \ldots \theta_k = G_0\theta_1 \ldots \theta_i\tau$. We show that this implies $G_0\theta_1 \ldots \theta_k = G_0\theta_1 \ldots \theta_j\sigma$, i.e., that D is pruned by the SIR_L check. Let $x \in var(G_0\theta_1 \ldots \theta_i)$, hence $x\theta_{i+1} \ldots \theta_k = x\tau$. We show that $x\theta_{i+1} \ldots \theta_j\sigma = x\theta_{i+1} \ldots \theta_k$.

If $x \notin var(S)$, then $x\theta_{i+1} \ldots \theta_j = x$, hence $x\theta_{i+1} \ldots \theta_j\sigma = x\sigma =$

$$= \begin{cases} x = x\theta_{i+1} \ldots \theta_j = x\theta_{i+1} \ldots \theta_k & \text{if } x \notin var(G_j), \\ x\theta_{j+1} \ldots \theta_k = x\theta_{i+1} \ldots \theta_k & \text{if } x \in var(G_j) - var(A), \\ x\tau = x\theta_{i+1} \ldots \theta_k & \text{if } x \in var(G_j) \cap var(A). \end{cases}$$

If $x \in var(S)$, then again for every $y \in var(x\theta_{i+1} \ldots \theta_j)$, either $y \in var(S)$ or $y \notin var(A)$, and in both cases $y\sigma = y\theta_{j+1} \ldots \theta_k$ (if $y \notin var(G_j)$ then $y\sigma = y = y\theta_{j+1} \ldots \theta_k$). So $x\theta_{i+1} \ldots \theta_j\sigma = x\theta_{i+1} \ldots \theta_k$. □

Although we conjecture that the subsumption checks using variants are also stronger than the corresponding context checks using variants, it appeared not easy to prove this. (One must show that σ is a renaming, or more precisely that an alternative definition for σ makes it a renaming and preserves the properties proved above.) We feel that the conjecture is not interesting enough to justify such an effort.

The following example shows that the previous result does not hold for selection rules that are not local.

EXAMPLE 3.4.9 (based on Example 10 in [B]).

Let P = { p ← q. (C1),
q ← p. (C2),
r ← s. (C3) },

and let G = ←p,r.

Then the derivation ←p,r $\Rightarrow_{(C1)}$ ←q,r $\Rightarrow_{(C3)}$ ←q,s $\Rightarrow_{(C2)}$ ←p,s (in which the selected atoms are underlined) is pruned by the context checks (the p in the fourth goal is the result of resolving the p in the first goal), but not by the subsumption checks. □

Now we can add the context checks to our 'stronger than' scheme (see Figure 3.4.3; the dotted arrows are only valid for local selection rules). Proving the relations between the context checks is again straightforward.

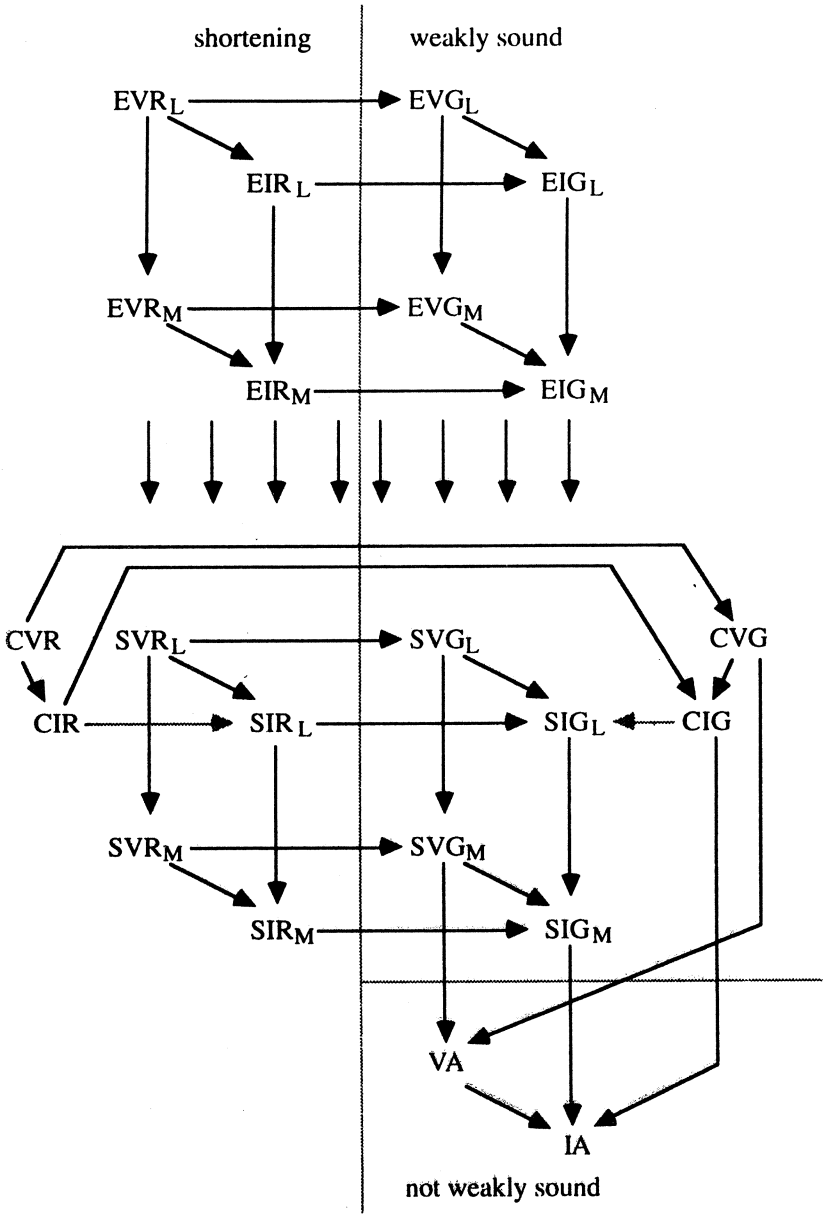

shortening
weakly sound
EVR$_L$
EVG$_L$
EIR$_L$
EIG$_L$
EVR$_M$
EVG$_M$
EIR$_M$
EIG$_M$
CVR
SVR$_L$
SVG$_L$
CVG
CIR
SIR$_L$
SIG$_L$
CIG
SVR$_M$
SVG$_M$
SIR$_M$
SIG$_M$
VA
IA
not weakly sound

FIGURE 3.4.3

That the equality checks and the context checks are incomparable (even when we consider only derivations via local selection rules) is shown in the following example.

EXAMPLE 3.4.10.
The derivation $\leftarrow p \Rightarrow_{p\leftarrow p,q} \leftarrow p,q$ is pruned by each context check, but not by any of the equality checks. Conversely, the derivation $\leftarrow r \Rightarrow_{r\leftarrow p(x),q(x),\varepsilon} \leftarrow p(x),q(x) \Rightarrow_{p(y)\leftarrow,\{y/x\}} \leftarrow q(x) \Rightarrow_{q(z)\leftarrow p(w),q(w),\{z/x\}} \leftarrow p(w),q(w)$ is pruned by each equality check, but not by any of the context checks ($q(x)$ produces $q(w)$, but x occurs both inside and outside $q(x)$, and the substitution $\{x/w\}$ does not agree with $\{y/x\}\{z/x\}$ on x). □

Completeness
Again we shift our attention to completeness issues. We first prove that, like the equality checks and the subsumption checks, the context checks are complete w.r.t. the leftmost selection rule for function-free restricted programs.

THEOREM 3.4.11. *The CVR check is complete w.r.t. the leftmost selection rule for function-free restricted programs.*

PROOF. Let P be a function-free restricted program and let G_0 be a goal in L_P. Let $k = \text{weight}(G_0)$. Consider an infinite SLD-derivation $D = (G_0 \Rightarrow_{C_1,\theta_1} G_1 \Rightarrow \ldots \Rightarrow G_{i-1} \Rightarrow_{C_i,\theta_i} G_i \Rightarrow \ldots)$ of $P \cup \{G_0\}$. By Corollary 3.2.14 for every $i \geq 0$: $|G_i| \leq k$. Every goal G_i is a goal in L_P and hence every resultant $G_0\theta_1 \ldots \theta_i \leftarrow G_i$ belongs to an equivalence class of $\sim_{\emptyset,G_0,k}$. L_P satisfies the conditions of Lemma 3.2.17, so $\sim_{\emptyset,G_0,k}$ has only finitely many equivalence classes. Thus the set $E = \{\underline{e} \mid \underline{e}$ is an equivalence class of $\sim_{\emptyset,G_0,k}$ and for infinitely many resultants R associated to goals in D: $R \in \underline{e}\}$ is nonempty. For simplicity, we shall say that the goal G_i is in an equivalence class $\underline{e}$, when in fact $(G_0\theta_1 \ldots \theta_i \leftarrow G_i) \in \underline{e}$.

For every equivalence class $\underline{e}$ of $\sim_{\emptyset,G_0,k}$, we define the length of $\underline{e}$, denoted by $|\underline{e}|$, as the length of the goals in $\underline{e}$. Since $E \neq \emptyset$, we can define $l = \min\{|\underline{e}| \mid \underline{e} \in E\}$. Now we choose an equivalence class $e \in E$ with $|e| = l$. According to the choice of e, D contains infinitely many goals in e and a finite number of shorter goals (since the number of equivalence classes of $\sim_{\emptyset,G_0,k}$ is finite).

Let G_i and G_k be (the first) two goals in D that are in e such that no goal lying in D between them is shorter. Since G_i and G_k are in the same equivalence class e, we have $G_k = G_i\tau$ and $G_0\theta_1\ldots\theta_k = G_0\theta_1\ldots\theta_i\tau$ for some renaming τ.

Let A be the leftmost atom in G_i and let S be the rest of G_i. A is selected in G_i. However, A is not completely resolved between G_i and G_k, otherwise a goal shorter than G_i, namely an instance of S, would appear between G_i and G_k in D. Therefore the atom $A\tau$ in G_k is the result of resolving A. Furthermore, no atom of S is selected between G_i and G_k, so $G_k = (A\tau,S\theta_{i+1}\ldots\theta_k)$. Hence $S\theta_{i+1}\ldots\theta_k = S\tau$.

When in the resultant $R_i\theta_{i+1}\ldots\theta_k$, we replace $A\theta_{i+1}\ldots\theta_k$ by $A\tau$, we obtain $(G_0\theta_1\ldots\theta_k\leftarrow A\tau,S\theta_{i+1}\ldots\theta_k) = (G_0\theta_1\ldots\theta_i\tau\leftarrow A\tau,S\tau)$, which is a variant of R_i. Therefore D is pruned by the CVR check. □

COROLLARY 3.4.12 (Context Completeness 1). *All context checks are complete w.r.t. the leftmost selection rule for function-free restricted programs.*

PROOF. By Theorem 3.4.11 and the Relative Strength Theorem 2.2.12. □

Besnard [B] claims without much proof that the CIG check is complete for function-free nvi programs. It appears that even the weakest of the four context checks, the CVR check, is complete for function-free nvi programs.

THEOREM 3.4.13. *The CVR check is complete for function-free nvi programs.*

PROOF. Let P be an nvi program, G_0 a goal in L_P and $D = (G_0 \Rightarrow_{C_1,\theta_1} G_1 \Rightarrow_{C_2,\theta_2} G_2 \Rightarrow \ldots)$ an infinite SLD-derivation of $P\cup\{G_0\}$.

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 $G_{m_0},G_{m_1},\ldots$ $(0 \le m_0 < m_1 < \ldots)$ containing atoms $A_0,A_1,\ldots$ such that for every $i \ge 0$:

- 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 3.4.4 (selected atoms are underlined).

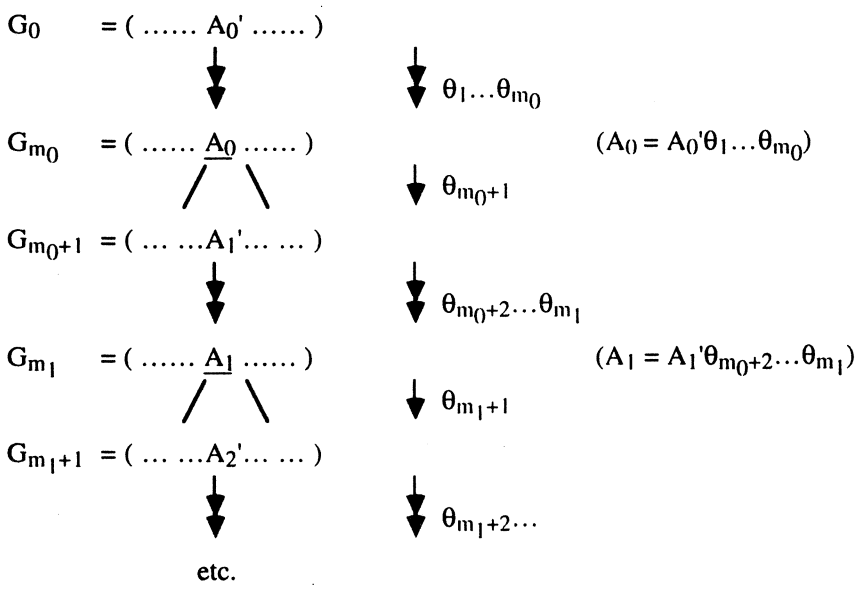


FIGURE 3.4.4

We now consider the resultants $G_0\theta_1\theta_2\ldots\theta_{m_i} \leftarrow A_i$ ($i \geq 0$). These resultants belong to equivalence classes of the relation $\sim_{var(G_0),G_0,1}$, which has by Lemma 3.2.17 only finitely many equivalence classes. Hence for some p and $q > p$: $(G_0\theta_1\theta_2\ldots\theta_{m_p} \leftarrow A_p) \sim_{var(G_0),G_0,1} (G_0\theta_1\theta_2\ldots\theta_{m_q} \leftarrow A_q)$.
So by Definition 3.2.15, there exists a renaming ρ such that:
- $G_0\theta_1\theta_2\ldots\theta_{m_p}\rho = G_0\theta_1\theta_2\ldots\theta_{m_q}$,
- $A_p\rho = A_q$,
- ρ does not act on the variables of G_0.

When this is compared with the definition of the CVR check, taking $i = j = m_p$, $k = m_q$, $A = A_p$ and $\tau = \rho$, it appears that the only additional condition for pruning is that 'for every variable x that occurs both inside and outside of A_p in G_{m_p}: $x\theta_{m_p+1}\ldots\theta_{m_q} = x\rho$'. We now prove that this condition is also satisfied, which proves that D is pruned by the CVR check.

By Lemma 1.2.5, 3.3.15 and 3.4.5 we may assume that D is an nvi derivation. Then $var(G_{m_p}) \subseteq var(G_0)$, so for every variable x in G_{m_p}: $x\rho = x$. In particular, it follows that $A_q = A_p\rho = A_p$.

Now suppose that x occurs both inside and outside of A_p in G_{m_p}. Then x occurs in A_q, hence in G_{m_q}. By Lemma 1.2.1, x occurs in every goal between G_{m_p} and G_{m_q} and $x\theta_{m_p+1} = \ldots = x\theta_{m_q} = x$. Hence $x\theta_{m_p+1}\ldots\theta_{m_q} = x = x\rho$. □

COROLLARY 3.4.14 (Context Completeness 2). *All context checks are complete for function-free nvi programs.*

PROOF. By Theorem 3.4.13 and the Relative Strength Theorem 2.2.12. □

Finally, it appears that the context checks are also complete for function-free svo programs.

THEOREM 3.4.15. *The CVR check is complete for function-free svo programs.*

PROOF. The proof of this theorem is identical to the proof of Theorem 3.4.13, up to the point where the condition 'for every variable x that occurs both inside and outside of A_p in G_{m_p}: $x\theta_{m_p+1}\ldots\theta_{m_q} = x\rho$' must be proved. In the case of P being an svo program, this condition is proved as follows. If x occurs both inside and outside of A_p in G_{m_p}, then x occurs more than once in G_{m_p}, so by Lemma 3.3.19 $x \in var(G_0)$. Thus $x\rho = x$. Hence x occurs in A_q, hence in G_{m_q}. By Lemma 1.2.1, x occurs in every goal between G_0 and G_{m_q} and $x\theta_{m_p+1} = \ldots = x\theta_{m_q} = x$. Hence $x\theta_{m_p+1}\ldots\theta_{m_q} = x = x\rho$. □

COROLLARY 3.4.16 (Context Completeness 3). *All context checks are complete for function-free svo programs.*

PROOF. By Theorem 3.4.15 and the Relative Strength Theorem 2.2.12. □

Now combining Corollary 2.2.6 and Corollary 2.2.7 with the Context Soundness Corollary 3.4.7 and the Context Completeness Corollaries 3.4.12 and 3.4.14 and 3.4.16, we conclude that all context checks lead to an implementation of CWA for restricted programs, nvi programs and svo programs without function symbols. Moreover, the context checks based on resultants also lead to an implementation of query processing for these programs.

More results on the completeness of loop checks can be found in [FPS], and in the next chapter.

4. Generalizing Completeness Results for Loop Checks

This chapter is devoted to the Generalization Theorem. This theorem presents a method to extend (under certain conditions) a class of programs for which a given loop check is complete to a larger class, for which the loop check is still complete. These conditions and classes of programs are presented in Section 4.1. The theorem is proved in Section 4.2.

In Section 4.3 the theorem is applied to Corollaries 3.3.18, 3.3.21, 3.4.14 and 3.4.16 ('The subsumption and context checks are complete for function-free nvi and svo programs.'), giving rise to stronger completeness theorems. The proof of the result based on nvi programs is straightforward, whereas the result based on svo programs requires a more elaborate analysis.

4.1. Preparation

Introduction

In the previous chapter, a number of natural simple loop checks was introduced. These loop checks were proven to be sound, but only complete for certain classes of programs. For each of these loop checks, one or more such classes were determined.

Here, the problem of finding classes of programs for which a simple loop check is complete is addressed in more generality. We prove the *Generalization Theorem*, which allows us to generalize certain completeness results: given that a loop check L is complete for a class of programs $\mathcal{C}$, we may conclude that L is also complete (w.r.t. the leftmost selection rule) for a class of programs extending $\mathcal{C}$, provided that L and $\mathcal{C}$ satisfy some natural conditions.

Basically, the theorem is only applicable to a class of programs $\mathcal{C}$ if $\mathcal{C}$ = {P | every clause in program P satisfies Pr[1]}, for some property Pr of clauses that is 'local' to clauses (that is, whether a clause satisfies Pr is independent of the rest of the program). We say that $\mathcal{C}$ is the class of *Pr programs*. By allowing

[1] We do not give a formal definition of a property: we assume that the notion 'a clause C satisfies a property Pr' is given.

the addition of nonrecursive atoms (Definition 3.2.10), the class of *nr-extended Pr programs* is obtained.

The Generalization Theorem states that if the loop check L is complete for function-free Pr programs, then L is also complete for function-free nr-extended Pr programs, provided that the nonrecursive atoms are resolved before other atoms are selected. For simplicity, this is achieved by using the leftmost selection rule, and putting the nonrecursive atoms left from the other atoms in the clause. Formally, we have the following definitions.

Definitions

DEFINITION 4.1.1.
Let Pr be a property of clauses. A program P *satisfies* Pr (P *is a Pr program*) if every clause in P satisfies Pr. □

DEFINITION 4.1.2 (Nr-extended Pr program).
Let P be a program. A clause C = (H←NR,R) is *nr-extended Pr w.r.t. P* if the clause H←R satisfies Pr and NR contains no recursive atoms. NR is called the *nonrecursive part* of C and R is called the *Pr-part*.
A program P is *nr-extended Pr* if every clause in P is nr-extended Pr w.r.t. P. □

EXAMPLE 4.1.3.
If Pr is the property 'C is a unit clause', then the class of nr-extended Pr programs is the class of hierarchical normal programs ([L]): programs in which no recursion is allowed.
If Pr is the property 'the body of C has at most one atom', then the class of nr-extended Pr programs is the class of restricted programs (Definition 3.2.10). □

Note that the property of being a nonrecursive atom is *not* local to clauses; therefore the construction cannot be applied repeatedly. We must make one more restriction on the properties we consider.

DEFINITION 4.1.4.
A property of clauses Pr is *closed under instantiation* if for every clause C that satisfies Pr and for every substitution σ, Cσ satisfies Pr. □

Note that $C\sigma$ is not necessarily a ground instance of C. The Generalization Theorem is only valid for properties that are closed under instantiation, such as the properties of Example 4.1.3 and the property 'C is non-variable introducing' (Definition 3.3.10). However, in the next section, where we study some applications of the Generalization Theorem, we also consider a property that is *not* closed under instantiation, namely the single variable occurrence (svo) property (Definition 3.3.11). A detailed inspection of the proof of the Generalization Theorem enables us to derive useful results for this property as well.

The Generalization Theorem is only valid for loop checks satisfying certain conditions. These conditions are formalized here. The first condition is that the loop check is 'safe for goal extension'. Informally, this means that when we have a derivation that is pruned by the loop check, adding some atoms to the initial goal that are never selected (before the derivation is pruned), yields a pruned derivation again.

DEFINITION 4.1.5.
A loop check L is *safe for goal extension* if for every SLD-derivation D of $P\cup\{\leftarrow G\}$ that is pruned by L, an SLD-derivation of $P\cup\{\leftarrow(G,H)\}$ which selects the same atoms, and uses the same input clauses and mgu's as D is also pruned by L. □

The second condition is that the loop check is 'safe for initialization'. Informally, this means that when we have a derivation that is pruned by the loop check, adding some derivation steps in front of it ('initialization steps'), yields a pruned derivation again.

DEFINITION 4.1.6.
A loop check L is *safe for initialization* if for every SLD-derivation $D = (G_i \Rightarrow_{C_{i+1},\theta_{i+1}} G_{i+1} \Rightarrow_{C_{i+2},\theta_{i+2}} G_{i+2} \Rightarrow \ldots)$ that is pruned by L ($i > 0$), every derivation $(G_0 \Rightarrow_{C_1,\theta_1} G_1 \Rightarrow \ldots \Rightarrow G_i \Rightarrow_{C_{i+1},\theta_{i+1}} G_{i+1} \Rightarrow_{C_{i+2},\theta_{i+2}} G_{i+2} \Rightarrow \ldots)$ in which in $G_i, G_{i+1}, \ldots$ the same atoms are selected as in D, is pruned by L. □

The third condition is that the loop check is 'safe for detailing'. Informally, this means that when we have a derivation that is pruned by the loop check,

replacing every derivation step by one or more steps giving the same computed answer ('showing the details of one step in several steps'), yields a pruned derivation again.

DEFINITION 4.1.7.
A loop check L is *safe for detailing* if for every SLD-derivation
$D = (G_0 \Rightarrow_{C_1,\theta_1} G_1 \Rightarrow \ldots)$ that is pruned by L, every derivation of the form

$$(G_0 \Rightarrow_{C_1^1,\tau_1^1} H_1^1 \Rightarrow \ldots \Rightarrow H_{n_1-1}^1 \Rightarrow_{C_{n_1}^1,\tau_{n_1}^1} G_1$$

$$\Rightarrow_{C_1^2,\tau_1^2} H_1^2 \Rightarrow \ldots \Rightarrow H_{n_2-1}^2 \Rightarrow_{C_{n_2}^2,\tau_{n_2}^2} G_2 \Rightarrow \ldots)$$

with for every $i > 0$:

$$\tau_1^i \ldots \tau_{n_i}^i |_{var(G_0,G_1,\ldots,G_{i-1})} = \theta_i |_{var(G_0,G_1,\ldots,G_{i-1})}$$

in which in G_0, G_1, ... the same atoms are selected as in D, is pruned by L. □

4.2. The Generalization Theorem

We can now formulate the Generalization Theorem.

THEOREM 4.2.1 (Generalization Theorem). *Let Pr be a property of clauses that is closed under instantiation. Let L be a loop check such that*
- L is complete for function-free Pr programs and
- L is safe for goal extension, initialization and detailing.
Then L is complete w.r.t. the leftmost selection rule for function-free nr-extended Pr programs.

In the rest of this section, we shall assume that Pr is a property and L is a loop check satisfying the above conditions. For proving this theorem, we use the following lemma.

LEMMA 4.2.2. *Let P be a function-free nr-extended Pr program and let G_0 be a goal in L_P. Let D be an infinite SLD-derivation of $P \cup \{G_0\}$ via the leftmost selection rule. Suppose that*
for no goal $G_i = \leftarrow (G,H)$ in D ($i \geq 0$), the derivation of $P \cup \{\leftarrow G\}$ (using the same input clauses, mgu's and selection rule as D) is pruned by L. (*)
Then D is pruned by L.

Before proving this lemma, we show that the Generalization Theorem is an immediate consequence of it.

PROOF OF THE GENERALIZATION THEOREM. Let P be a function-free nr-extended Pr program, G_0 a goal in L_P and D an infinite SLD-derivation of $P \cup \{G_0\}$. Two cases arise.

i) For no goal $\leftarrow(G,H)$ in D, the derivation of $\leftarrow G$ (using the same input clauses, mgu's and selection rule as D) is pruned by L. Then by Lemma 4.2.2, D is pruned by L.

ii) Otherwise, there is a goal $\leftarrow(G,H)$ in D for which the derivation of $\leftarrow G$ (using the same input clauses, mgu's and selection rule as D) is pruned by L. Then the tail of D starting at this goal $\leftarrow(G,H)$ is pruned, since L is safe for goal extension. So D is pruned by L too, since L is safe for initialization. □

PROOF OF LEMMA 4.2.2. Assume that D is an infinite SLD-derivation of $P \cup \{G_0\}$ that satisfies (*). The dependency graph D_P of P (see Definition 3.2.9) defines a (well founded) partial ordering $\leq$ of the set $\{cl_P(p) \mid p$ is a predicate symbol in $L_P\}$. Therefore we may assume as induction hypothesis (by a complete induction on $\leq$), that this lemma has been proved for every derivation of $P \cup \{G\}$ where G contains only strict $\leq$-smaller predicate symbols than the $\leq$-largest predicate symbol in G_0.

CLAIM 1. *D is of the form*

$$(G_0 \Rightarrow_{C_1^1, \tau_1^1} H_1^1 \Rightarrow \ldots \Rightarrow H_{n_1-1}^1 \Rightarrow_{C_{n_1}^1, \tau_{n_1}^1} G_1$$

$$\Rightarrow_{C_1^2, \tau_1^2} H_1^2 \Rightarrow \ldots \Rightarrow H_{n_2-1}^2 \Rightarrow_{C_{n_2}^2, \tau_{n_2}^2} G_2 \Rightarrow \ldots)$$

for some derivation $D' = (G_0 \Rightarrow_{C_1, \theta_1} G_1 \Rightarrow_{C_2, \theta_2} G_2 \Rightarrow \ldots)$, *with for every* $i > 0$:

$$\tau_1^i \ldots \tau_{n_i}^i \mid_{var(G_0, G_1, \ldots, G_{i-1})} = \theta_i \mid_{var(G_0, G_1, \ldots, G_{i-1})},$$

and where C_1, C_2, ... *all satisfy Pr. Moreover, in the goals* G_0, G_1, ..., *the same atoms are selected in D and D'.*

Lemma 4.2.2 follows from Claim 1: D' is a derivation of $\{G_0, C_1, C_2, \ldots\}$, $\{C_1, C_2, \ldots\}$ is a function-free Pr program and L is complete for function-free Pr

programs, so D' is pruned by L. As L is safe for detailing, D is pruned by L too.

PROOF OF CLAIM 1. We prove the claim by induction. Suppose we have constructed D' and proved the claim up to the goal G_i. (Up to G_0, the claim is trivial.)

Let $G_i = \leftarrow A_1,\ldots,A_n$, let $C = C_1^{i+1} = (A \leftarrow NR,R)$ and let $\tau = \tau_1^{i+1}$. Suppose that NR is the nonrecursive part of the body of C and that R is the Pr-part. The next step in D is $G_i \Rightarrow_{C,\tau} \leftarrow(NR,R,A_2,\ldots,A_n)\tau$. Let D_1 be the SLD-derivation of $P \cup \{\leftarrow NR\tau\}$ that uses the same input clauses, mgu's and selection rule as the tail of D starting at $\leftarrow(NR,R,A_2,\ldots,A_n)\tau$. Four cases arise.

1) NR is empty. This is a special case of case 4: $P \cup \{\leftarrow NR\tau\}$ is immediately successfully refuted. (If G_0 is $\leq$-minimal, then this is the only possible case, since then $rel(A_1) = rel(A)$ is $\leq$-minimal and by definition every predicate symbol in NR is strictly $\leq$-smaller than rel(A).)
2) D_1 is failed. Then D is failed too, which contradicts the assumption that D is infinite.
3) D_1 is infinite. By definition, every predicate symbol in NR is strictly $\leq$-smaller than $rel(A_1)$, which is $\leq$-smaller than the $\leq$-largest predicate symbol in G_i (hence in G_0), so by induction on $\leq$ we can apply Lemma 4.2.2 on D_1. Hence D_1 is pruned by L or D_1 does not satisfy (*). However, this contradicts the assumption that D satisfies (*): if D_1 is pruned then we take $G = NR\tau$ and $H = (R,A_2,\ldots,A_n)\tau$; if D_1 does not satisfy (*), say for $H_{i'}^{i+1} = \leftarrow(G',H')$, then we take $G = G'$ and $H = (H',(R,A_2,\ldots,A_n)\tau_1^{i+1}\ldots\tau_{i'}^{i+1})$.
4) D_1 is successful, yielding a computed answer substitution σ (if NR is empty then $\sigma = \varepsilon$). This is the only remaining case. In this case we have in D the goal $G_{i+1} = \leftarrow(R,A_2,\ldots,A_n)\tau\sigma$, immediately after NR is completely resolved.

CLAIM 2. *The sequence of resolution steps between G_i and G_{i+1} in D can be mimiced by one resolution step $G_i \Rightarrow_{C_{i+1},\theta_{i+1}} G_{i+1}$ in D', where C_{i+1} is an instance of $A \leftarrow R$ and $\tau\sigma|_{var(G_0,G_1,\ldots,G_i)} = \theta_{i+1}|_{var(G_0,G_1,\ldots,G_i)}$.*

Claim 1 follows from Claim 2: since Pr is closed under instantiation, C_{i+1} satisfies Pr. So we have constructed D' and proved Claim 1 up to the goal G_{i+1}.

Now the construction of the resolution step $G_i \Rightarrow_{C_{i+1},\theta_{i+1}} G_{i+1}$ remains.

PROOF OF CLAIM 2. First, we define C_{i+1} and θ_{i+1}, then we prove that $G_i \Rightarrow_{C_{i+1},\theta_{i+1}} G_{i+1}$ is indeed a derivation step. Finally, we check the other requirements on C_{i+1} and θ_{i+1}. By Lemma 1.2.5 we may assume that D is normal.

For every chain C in NR, we fix a substitution ρ_C such that for every $x \in \mathrm{var}(C\tau)$, $x\rho_C \in C$ and $x\rho_C\tau = x$. Moreover, if $x \in \mathrm{var}((\mathrm{var}(R) \cap C)\tau)$, then $x\rho_C \in \mathrm{var}(R)$. For every chain, such a substitution exists: if $x \in \mathrm{var}(C\tau)$, then $\{y \in C \mid y\tau = x\} \neq \emptyset$. If $\{y \in \mathrm{var}(R) \cap C \mid y\tau = x\} \neq \emptyset$, then $x\rho_C$ must be chosen from the latter set, otherwise any element of the former set will do.

Now we can define ψ by: $x\psi = \begin{cases} x & \text{if } x \notin \mathrm{var}(NR) \\ x\tau\sigma\rho_{C(x)} & \text{if } x \in C \subseteq \mathrm{var}(NR) \end{cases}$.

(Note that if $x\tau\sigma$ is a variable, then $x\tau\sigma \in \mathrm{var}(C(x)\tau\sigma) \subseteq \mathrm{var}(C(x)\tau)$ by Corollary 1.2.8, since D is normal.)

Finally, we define $C_{i+1} = (A \leftarrow R)\psi$ and $\theta_{i+1} = \tau\sigma_{|\mathrm{var}(A_1,A\psi)}$. Now we must prove that $G_i \Rightarrow_{C_{i+1},\theta_{i+1}} G_{i+1}$ is indeed a resolution step. That is:

CLAIM 3. *$(A \leftarrow R)\psi$ is properly standardized apart.*

CLAIM 4. *θ_{i+1} is an idempotent mgu of $A\psi$ and A_1.*

CLAIM 5. *$(R\psi, A_2, \ldots, A_n)\theta_{i+1} = (R, A_2, \ldots, A_n)\tau\sigma$.*

In the proofs of these claims, we take $C(x) = C_{NR}(x)$.

PROOF OF CLAIM 3. We prove that $\mathrm{var}((A \leftarrow R)\psi) \subseteq \mathrm{var}(A \leftarrow NR, R)$.
Let $x \in \mathrm{var}(A \leftarrow R)$. Then:
if $x\psi = x$, then $x\psi \in \mathrm{var}(A \leftarrow R)$;
if $x\psi \neq x$, then $x \in C(x) \subseteq \mathrm{var}(NR)$, so if $x\psi$ is a variable, then $x\psi = x\tau\sigma\rho_{C(x)} \in C(x) \subseteq \mathrm{var}(NR)$.

Before proving Claim 4, we prove an additional claim.

CLAIM 6. *ψ is idempotent.*

PROOF. Let x be a variable. If $x\psi = x$, then $x\psi\psi = x\psi$.
Otherwise, $x\psi\psi = x\tau\sigma\rho_{C(x)}\psi =$ (since $x\tau\sigma\rho_{C(x)} \in C(x) \subseteq \mathrm{var}(NR)$ or $x\tau\sigma$ is a constant) $= x\tau\sigma\rho_{C(x)}\tau\sigma\rho_{C(x)} = x\tau\sigma\sigma\rho_{C(x)} =$ (σ is idempotent) $= x\tau\sigma\rho_{C(x)} = x\psi$.

PROOF OF CLAIM 4. We prove that for every relevant unifier η of A_1 and $A\psi$: $\eta = \theta_{i+1}\eta$. Let η be a relevant unifier of A_1 and $A\psi$: $A_1\eta = A\psi\eta$.

By standardizing apart, $var(A_1) \cap var(NR) = \emptyset$, so we have $A_1 = A_1\psi$. Therefore, $\psi\eta$ is a unifier of A_1 and A. Since τ is an idempotent mgu of A_1 and A, we have $\psi\eta = \tau\omega = \tau\tau\omega = \tau\psi\eta$ ($\tau \leq \psi\eta$, so for some ω: $\tau\omega = \psi\eta$).

Let x be a variable. If $x \notin var(A_1,A\psi)$, then $x = x\theta_{i+1}$, so $x\eta = x\theta_{i+1}\eta$.
If $x \in var(A_1)$, then at the corresponding position in A, we find a term (constant or variable) t such that $x\eta = t\psi\eta$ and $x\tau = t\tau$. Two cases arise.
- $x\tau = x\tau\sigma$. Then $x\eta = t\psi\eta = t\tau\psi\eta = x\tau\psi\eta =^* x\tau\eta = x\tau\sigma\eta = x\theta_{i+1}\eta$.
  * : $x\tau \notin var(NR)$, since either $x\tau$ is ground, or $x\tau \in var(A_1\tau) \subseteq var(A_1)$ (the latter inclusion by Corollary 1.2.10, since D is normal).
- $x\tau \neq x\tau\sigma$. Then $x\tau \in var(NR\tau)$, so for some $v \in var(NR)$: $v\tau = x\tau$ and $v\psi = v\tau\sigma\rho_{C(v)}$. Now $x\eta = t\psi\eta = t\tau\psi\eta = x\tau\psi\eta = v\tau\psi\eta = v\psi\eta =$ (by Claim 6) $= v\psi\psi\eta = v\psi\tau\psi\eta = v\tau\sigma\rho_{C(v)}\tau\psi\eta = v\tau\sigma\psi\eta = x\tau\sigma\psi\eta =^* x\tau\sigma\eta = x\theta_{i+1}\eta$.
  * : $x\tau\sigma \notin var(NR)$, since either $x\tau\sigma$ is ground, or $x\tau\sigma \in var(A_1\tau\sigma) \subseteq var(A_1)$. (The latter inclusion by Corollary 1.2.10, since D is normal).

If $x \in var(A\psi)$, then for some $y \in var(A)$ we have $y\psi = x$. At the corresponding position in A_1, we find a term t such that $x\eta = t\eta$ and $y\tau = t\tau$. Again two cases arise.
- $y \notin var(NR)$. Then $y\psi = y$ and $y\tau\sigma = y\tau$. Therefore we have $x\eta = y\psi\eta = y\tau\psi\eta =^* y\tau\eta = y\tau\sigma\eta = y\psi\tau\sigma\eta = x\tau\sigma\eta = x\theta_{i+1}\eta$.
  * : $y\tau \notin var(NR)$, since either $y\tau$ is ground, or $y\tau \in var(A\tau) = var(A_1\tau) \subseteq var(A_1)$.
- $y \in var(NR)$. Then $y\psi = y\tau\sigma\rho_{C(y)}$, so (see Claim 6), $y\psi = y\psi\psi = y\psi\tau\sigma\rho_{C(y\psi)} = x\tau\sigma\rho_{C(x)}$. Therefore we have $x\eta = y\psi\eta =$ (by Claim 6) $y\psi\psi\eta = y\psi\tau\psi\eta = x\tau\sigma\rho_{C(x)}\tau\psi\eta = x\tau\sigma\psi\eta =^* x\tau\sigma\eta = x\theta_{i+1}\eta$. * : again, $x\tau\sigma \notin var(NR)$.

PROOF OF CLAIM 5.
If $x \in var(A_i)$ $(2 \leq i \leq n)$, then
  if $x \notin var(A_1)$ then $x\theta_{i+1} = x = x\tau\sigma$;
  if $x \in var(A_1)$ then by definition $x\theta_{i+1} = x\tau\sigma$.
If $x \in var(R)$, then two cases arise.
- $x\psi \in var(A\psi)$. Then $x\psi\theta_{i+1} = x\psi\tau\sigma =$
  (if $x \notin var(NR)$) : $x\tau\sigma$.
  (if $x \in var(NR)$) : $x\tau\sigma\rho_{C(x)}\tau\sigma = x\tau\sigma\sigma = x\tau\sigma$.

- $x\psi \notin var(A\psi)$. Then either $x\psi$ is ground or for no $y \in var(A)$: $y\psi = x\psi$.
If $x\psi$ is ground, then $x\tau\sigma$ is ground, so $x\psi = x\tau\sigma\rho_{C(x)} = x\tau\sigma$.
If for no $y \in var(A)$: $y\psi = x\psi$, then in particular, $x \notin var(A)$, so $x\tau = x$.

Now, if $x \notin var(NR)$, then $x\psi = x = x\tau = x\tau\sigma$.
If $x \in var(NR)$, then $x\psi = x\tau\sigma\rho_{C(x)}$. Also, $x\tau\sigma \in C(x)\tau\sigma \subseteq C(x)\tau$ (by Corollary 1.2.8, since D is normal), so for some $z \in C(x)$: $z\tau = x\tau\sigma$ (and $C(z) = C(x)$). Then $z\tau\sigma = x\tau\sigma\sigma = x\tau\sigma$, so $z\psi = x\psi$. Hence $z \notin var(A)$, so $z\tau = z = z\rho_{C(z)}\tau$, so $z = z\rho_{C(z)} = z\rho_{C(x)}$. Therefore $x\tau\sigma\rho_{C(x)} = z\tau\rho_{C(x)} = z\rho_{C(x)} = z = z\tau = x\tau\sigma$.

Obviously, C_{i+1} is an instance of $A \leftarrow R$. Also, $\theta_{i+1}|_{var(G_0,G_1,\ldots,G_i)} = \tau\sigma|_{var(A_1,A_\psi)\cap var(G_0,G_1,\ldots,G_i)} = \tau\sigma|_{var(A_1)} = \tau\sigma|_{var(G_0,G_1,\ldots,G_i)})$, by Corollary 1.2.10, since D is normal and a local selection rule is used. This concludes the proof of Claim 2 and thereby the proof of Lemma 4.2.2. □

4.3. Applications of the Generalization Theorem

Introduction

A simple example of the application of the Generalization Theorem, based on the first observation in Example 4.1.3, is the following.

COROLLARY 4.3.1. *If P is a function-free hierarchical normal program, then every SLD-derivation of $P \cup \{G\}$ via the leftmost selection rule is finite.*

PROOF. We prove an equivalent proposition, namely that the empty loop check is complete w.r.t. the leftmost selection rule for function-free hierarchical normal programs. This follows from the Generalization Theorem and the following observations.

- The empty loop check is complete for 'unit-programs', programs that consist solely of unit clauses.
- The 'unit' property is closed under instantiation.
- The empty loop check is safe for goal extension, initialization and detailing.
- Nr-extended unit-programs are hierarchical normal programs. □

Of course, this result is well known, even for arbitrary selection rules and programs with function symbols. More interesting results can be obtained by using the Generalization Theorem to extend the completeness results presented

in Chapter 3. The first result presented there is the completeness of equality checks for function-free restricted programs w.r.t. the leftmost selection rule. The Generalization Theorem cannot be applied on this proposition. In contrast, the Generalization Theorem provides an alternative proof for this proposition, based on the lemma 'the equality checks are complete for function-free programs in which the body of each clause contains at most one atom'.

A generalized completeness result for subsumption and context checks

The other results of Chapter 3 are only valid for the subsumption and context checks. Therefore we shall now prove that the weakest of those checks, the SVR_L check and the CVR check, satisfy the conditions of the Generalization Theorem, i.e., that they are safe for goal extension, initialization and detailing.

LEMMA 4.3.2. *The SVR_L check and the CVR check are safe for goal extension.*

PROOF. Let D be an SLD-derivation of $P\cup\{\leftarrow G_0\}$. Let D' be an SLD-derivation of $P\cup\{\leftarrow(G_0,H_0)\}$, in which the same atoms are selected and the same input clauses and mgu's are used as in D. Thus D cannot contain any variable occurring in H_0 but not in G_0. Denote by θ_n the mgu used in the n-th resolution step of D and D' ($n \geq 1$).

If D is pruned by the SVR_L check resp. the CVR check, then we have for some renaming τ two goals G_i and G_k in D with $G_0\theta_1\ldots\theta_k = G_0\theta_1\ldots\theta_i\tau$ and $G_k \supseteq_L G_i\tau$ resp. (A in G_i 'produces' $A\tau$ in G_k and τ and $\theta_{i+1}\ldots\theta_k$ agree on $var(G_i)\cap var(A)$). Assuming that τ acts only on the variables in D, we have that $\theta_1\ldots\theta_k$ and $\theta_1\ldots\theta_i\tau$ coincide on *all* variables of H_0. So $(G_0,H_0)\theta_1\ldots\theta_k = (G_0,H_0)\theta_1\ldots\theta_i\tau$ and $(G_k,H_0\theta_1\ldots\theta_k) \supseteq_L (G_i,H_0\theta_1\ldots\theta_i)\tau$, resp. τ and $\theta_{i+1}\ldots\theta_k$ agree also on $var(H_0\theta_1\ldots\theta_i)\cap var(A)$. This means that D' is pruned by SVR_L, respectively CVR, as well. □

Notice that it is essential to consider loop checks based on resultants here. It is easy to see that the loop checks based on goals are *not* safe for goal extension (consider e.g. the program of Example 3.2.3 and the goal $\leftarrow p(x),p(x)$).

LEMMA 4.3.3. *The SVR_L check and the CVR check are safe for initialization.*

PROOF. Let $D' = (G_0 \Rightarrow_{C_1,\theta_1} G_1 \Rightarrow_{C_2,\theta_2} G_2 \Rightarrow \ldots)$ be an SLD-derivation. Suppose that for some $i > 0$ the derivation $D = (G_i \Rightarrow_{C_{i+1},\theta_{i+1}} G_{i+1} \Rightarrow_{C_{i+2},\theta_{i+2}}$

$G_{i+2} \Rightarrow \ldots$) is pruned by SVR_L resp. CVR. Clearly for some j, k > j and renaming τ (acting only on variables in D): τ 'proves' that G_j and G_k are 'sufficiently similar' for SVR_L, resp. CVR, and $G_i\theta_{i+1}\ldots\theta_k = G_i\theta_{i+1}\ldots\theta_j\tau$. So it remains to prove that $G_0\theta_1\ldots\theta_k = G_0\theta_1\ldots\theta_j\tau$.

Let $x \in var(G_0\theta_1\ldots\theta_i)$. Two cases arise.

i) $x \notin var(G_i)$. Then x does not occur in D, hence $x\theta_{i+1}\ldots\theta_k = x\theta_{i+1}\ldots\theta_j\tau = x$.

ii) $x \in var(G_i)$. Then $G_i\theta_{i+1}\ldots\theta_k = G_i\theta_{i+1}\ldots\theta_j\tau$ yields $x\theta_{i+1}\ldots\theta_k = x\theta_{i+1}\ldots\theta_j\tau$.

Hence D' is pruned by SVR_L, respectively CVR, as well. □

LEMMA 4.3.4. *The SVR_L check and the CVR check are safe for detailing.*

PROOF. Let $D = (G_0 \Rightarrow_{C_1,\theta_1} G_1 \Rightarrow \ldots)$ be an SLD-derivation that is pruned by SVR_L resp. CVR and let D' be an SLD-derivation of the form

$$(G_0 \Rightarrow_{C_1^1,\tau_1^1} H_1^1 \Rightarrow \ldots \Rightarrow H_{n_1-1}^1 \Rightarrow_{C_{n_1}^1,\tau_{n_1}^1} G_1$$

$$\Rightarrow_{C_1^2,\tau_1^2} H_1^2 \Rightarrow \ldots \Rightarrow H_{n_2-1}^2 \Rightarrow_{C_{n_2}^2,\tau_{n_2}^2} G_2 \Rightarrow \ldots)$$

with for every $i > 0$: $\tau_1^i\ldots\tau_{n_i}^i|_{var(G_0,G_1,\ldots,G_{i-1})} = \theta_i|_{var(G_0,G_1,\ldots,G_{i-1})}$

in which in G_0, G_1, ... the same atoms are selected as in D. Since D is pruned by SVR_L resp. CVR, we have for some j, k > j and renaming τ: τ 'proves' that G_j and G_k are 'sufficiently similar' for SVR_L, resp. CVR and $G_0\theta_1\ldots\theta_k = G_0\theta_1\ldots\theta_j\tau$. For CVR this proof includes that 'for every variable x that occurs both inside and outside of A in G_i, $x\theta_{i+1}\ldots\theta_k = x\tau$'. It follows immediately that

$$G_0\tau_1^1\ldots\tau_{n_1}^1\tau_1^2\ldots\tau_{n_2}^2\ldots\tau_1^k\ldots\tau_{n_k}^k = G_0\tau_1^1\ldots\tau_{n_1}^1\tau_1^2\ldots\tau_{n_2}^2\ldots\tau_1^j\ldots\tau_{n_j}^j\tau, \text{ and}$$

for CVR that 'for every variable x that occurs both inside and outside of A in G_i, $x\tau_1^{i+1}\ldots\tau_{n_2}^{i+1}\ldots\tau_1^k\ldots\tau_{n_k}^k = x\tau$'. Hence D' is pruned by SVR_L, resp. CVR. □

Now we can use the Generalization Theorem together with the fact that the subsumption and context checks are complete for function-free nvi programs.

COROLLARY 4.3.5. *The subsumption and context checks are complete w.r.t. the leftmost selection rule for function-free nr-extended nvi programs.*

PROOF. The nvi property is obviously closed under instantiation. Therefore by Theorem 3.3.17 respectively 3.4.13, the Generalization Theorem 4.2.1, and the

Lemmas 4.3.2, 4.3.3 and 4.3.4, the SVR_L check and the CVR check are complete w.r.t. the leftmost selection rule for function-free nr-extended nvi programs. Since the SVR_L check is the weakest of the subsumption checks and the CVR check is the weakest of the context checks, by the Relative Strength Theorem 2.2.12 the same holds for the other subsumption and context checks. □

The single variable occurrence property

Finally, in Chapter 3 it was proved that the subsumption and context checks are also complete for function-free svo programs. However, the single variable occurrence property is *not* closed under instantiation, so we cannot immediately use the Generalization Theorem. In fact, this should not come as a surprise, since *every* program can be converted into a 'computationally equivalent' nr-extended svo program. This can be done by replacing the $k > 1$ occurrences of a variable x in the body of a clause by 'new' variables $x_1,\ldots x_k$ and adding the nonrecursive atoms $eq(x,x_1),\ldots,eq(x,x_k)$ in the body of the clause. Finally the clause eq(x,x) is added to the program (assuming that *eq* is a new predicate symbol in P).

In the proof of Lemma 4.2.2, we need that the clause $C_{i+1} = (A\leftarrow R)\psi$ satisfies the property Pr considered, given that the clause $A\leftarrow R$ satisfies Pr. Up till now, this was derived immediately from the assumption that Pr should be closed under instantiation. Since for the svo property this is not true, we shall derive conditions that ensure directly that C_{i+1} satisfies the svo property, i.e., that every variable in $R\psi$ occurs only once (provided that every variable in R occurs only once).

Formally, let $x,y \in var(R)$ such that $x \neq y$ and $x\psi,y\psi \in VAR$. We shall derive conditions on the program ensuring that $x\psi \neq y\psi$.

If $x \notin var(NR)$, then $x\psi = x$.

Then, if $y \notin var(NR)$, $y\psi = y \neq x$, and

if $y \in var(NR)$, $y\psi = y\tau\sigma\rho_{C(y)} \in C(y) \subseteq var(NR)$, so $y\psi \neq x$.

The same argument holds if $y \notin var(NR)$. So a problem can only arise in the case that $x,y \in var(NR)$: then $x\psi = x\tau\sigma\rho_{C(x)} \subseteq C(x)$ and $y\psi = y\tau\sigma\rho_{C(y)} \subseteq C(y)$.

One solution is demanding that for every pair of distinct variables $x,y \in var(R) \cap var(NR)$, $C(x) \neq C(y)$. Then $C(x) \cap C(y) = \emptyset$, so $x\psi \neq y\psi$. This means that two variables in R should not be 'linked' by a chain through NR, thus the addition of the eq-atoms in the construction above is disallowed.

Another solution is to avoid that different variables in a (sub)goal are unified while the (sub)goal is refuted. (That is: to ensure that for every variable x in a goal, and for every unifier σ in the derivation, either $x\sigma = x$ or $x\sigma$ is a constant.) This condition can be met (for normal derivations) by the demand that variables do not occur more than once in the *head* of a clause. This disallows the addition of the clause eq(x,x)←.

In this case such a condition yields $x\psi = x\tau\rho_{C(x)}$ ($x\tau\sigma$ cannot be a constant, since $x\psi$ is a variable). Then $x\tau = x\tau\rho_{C(x)}\tau = x\psi\tau$. Using the condition again (but now w.r.t. τ), we obtain $x = x\psi$ (still, $x\tau$ cannot be a constant). Similarly we obtain $y = y\psi$, so $x\psi \neq y\psi$.

These two solutions give rise to two classes of programs for which the subsumption and context checks are complete w.r.t. the leftmost selection rule.

DEFINITION 4.3.6 (Chain-restricted svo program).
Let P be a program. A clause C = (A←NR,R) is *chain-restricted svo w.r.t. P* if C is nr-extended svo w.r.t. P, where NR is the nonrecursive part and R is the svo-part of C, and for every pair of distinct variables $x,y \in var(R)$, $C_{NR}(x) \neq C_{NR}(y)$. A program P is *chain-restricted svo* if every clause in P is chain-restricted svo w.r.t. P. □

DEFINITION 4.3.7 (Head-restricted svo program).
Let P be a program. A clause C is *head-restricted svo w.r.t. P* if C is nr-extended svo w.r.t. P and in the head of C, no variable occurs more than once. A program P is *head-restricted svo* if every clause in P is head-restricted svo w.r.t. P. □

COROLLARY 4.3.8. *The subsumption and context checks are complete w.r.t. the leftmost selection rule for function-free chain-restricted svo programs.*
PROOF. By Theorem 3.3.20 respectively 3.4.15, the Generalization Theorem 4.2.1, the Lemmas 4.3.2, 4.3.3 and 4.3.4 and the considerations above, the SVR_L check and the CVR check are complete w.r.t. the leftmost selection rule for function-free chain-restricted svo programs. Since the SVR_L check is the weakest of the subsumption checks and the CVR check is the weakest of the context checks, by the Relative Strength Theorem 2.2.12, the same holds for the other subsumption and context checks. □

COROLLARY 4.3.9. *The subsumption and context checks are complete w.r.t. the leftmost selection rule for function-free head-restricted svo programs.*

PROOF. By Theorem 3.3.20 respectively 3.4.15, the Generalization Theorem 4.2.1, the Lemmas 4.3.2, 4.3.3 and 4.3.4 and the considerations above, the SVR_L check and the CVR check are complete w.r.t. the leftmost selection rule for function-free head-restricted svo programs. Since the SVR_L check is the weakest of the subsumption checks and the CVR check is the weakest of the context checks, by the Relative Strength Theorem 2.2.12, the same holds for the other subsumption and context checks. □

We give now an example of a function-free head-restricted svo program that does not fall into any other class of programs discussed so far.

EXAMPLE 4.3.10.

The program $NONREC_P$ characterizes the predicates that are defined *without* recursion in a given program P (predicates in P are constants in $NONREC_P$). D_P (see Definition 3.2.9) cannot be represented in $NONREC_P$ by $\{dep(p,q)\leftarrow. \mid (p,q) \in D_P\}$, because (without the use of negation) these clauses fail to express that $(p,q) \notin D_P$ for some p and q. Instead, let $\{p_1,\ldots,p_n\}$ be the predicates in L_P. For every predicate p_i, there is only one ground clause $dep(p_i,q_1,\ldots,q_n)\leftarrow$ in $NONREC_P$ such that for some m: $\{(p_i,q_1),\ldots,(p_i,q_m)\} \subseteq D_P$ and $q_{m+1} = \ldots = q_n = nil$ (nil is a constant in $NONREC_P$ that differs from $p_1,\ldots,p_n$).

Furthermore, $NONREC_P$ contains the following two clauses:

$nonrec(nil) \leftarrow.$

$nonrec(x) \leftarrow dep(x,x_1,\ldots,x_n),nonrec(x_1),\ldots,nonrec(x_n).$

Without loop checking, this program goes into an infinite loop if and only if the predicate p is defined in P by means of recursion. As the program is head-restricted svo and function-free, the subsumption and context checks prune all its infinite loops, thus making the program work properly. □

Of course, in this example it is easier to write a restricted program that succeeds on predicates defined using recursion and that fails otherwise (using $\{dep(p,q)\leftarrow. \mid (p,q) \in D_P\}$ and computing the transitive closure of 'dep'). Then 'nonrec' can be defined via negation. Yet the combination of negation and loop checking is a delicate matter. That it is indeed possible is shown in Chapter 5.

5. Loop Checking and Negation

5.1. Introduction

In Chapter 2 a formal framework is given for loop checking mechanisms that operate on top-down interpreters for positive logic programs. This chapter extends this framework to interpreters for general logic programs, i.e., logic programs allowing negative literals in clauses' bodies. Several problems arising in the presence of negation are identified and solved. In a different setting, a combination of loop detection and negation was also studied in [KT] and [SI].

Loop checks are used to prune the search space generated by a top-down interpreter. Therefore, before loop checks can be defined, this search space needs to be described properly. The search space must in turn agree with the intended semantics of the program. For positive programs the choice was obvious: least Herbrand models and SLD-trees. However, in the presence of negation, we must pay more attention to this problem.

In the most well-known approach, introduced in [Cl2] and treated in [L], the intended semantics of general logic programs is derived from the *completion* of a program; the corresponding search space consists of *SLDNF-trees*, obtained when the interpreter is equipped with the *negation as finite failure* rule. Informally this rule states that $\neg$A may be inferred when an attempt to prove A (again by SLDNF-resolution) fails after a finite number of resolution steps. According to Theorem 16.5 of [L], for positive programs the completion semantics corresponds exactly to finite failure, i.e., $\neg$A is a logical consequence of the completion of P if and only if P$\cup\{\leftarrow$A$\}$ fails finitely. Due to the restriction to *finite* failure, this approach is hardly compatible with the use of a loop check. Indeed, the intention of loop checking is to turn infinite (hence failed) paths in the search space into finitely failed paths. Thus if P$\cup\{\leftarrow$A$\}$ fails finitely due to the use of a loop check, $\neg$A is not entailed by the completion-semantics. So completion semantics is inappropriate for our purposes.

Numerous alternative semantics have been proposed; see [ABo] for an overview. Here we adopt an approach of Przymusinski, which is based on *perfect model semantics* ([P1]). Furthermore, we restrict our attention to *locally stratified programs*; it is shown in [P1] that these programs have a unique perfect

Herbrand model (a 'perfect' model is defined as being minimal w.r.t. a certain partial ordering on models, which is a refinement of the usual subset ordering). The perfect model of a locally stratified program is also its unique stable model [GL] and its (total) well founded model [vGRS], [PW].

In [P2] a corresponding search space, called *SLS-trees*, is defined for stratified programs; this definition is generalized here to locally stratified programs. As pointed out by Przymusinski, an SLS-tree represents the search space of a top-down interpreter, equipped with the 'negation as failure' (not necessarily finite failure) rule.

Obviously this rule is in general not effective, so SLS-resolution cannot be effectively implemented, but only approximated.[1] However, as Przymusinski [P2] suggests, loop checks can yield such approximations:

> 'Suitable loop checking can be added to SLS-resolution without destroying its completeness. For large classes of stratified programs, SLS-resolution with subsumption check will result in finite evaluation trees and therefore can be implemented as a complete and always terminating algorithm. This is the case, in particular, for function-free programs.'

One of the contributions of this chapter is a substantiation of this claim.

It appears that for our purposes the standard SLS-trees do not present enough detail in the treatment of negative literals. Therefore these SLS-trees are augmented with *justifications*, which show explicitly the construction of a subsidiary SLS-tree of $\leftarrow A$, when $\neg A$ is selected.

A further major problem (not treated in [KT] and [SI]) is the occurrence of floundering: when only substitutions are used as computed answers, a nonground negative literal cannot be answered properly: the derivation is said to *flounder*. Floundering lies between success and failure, making it hard to determine which floundering derivations can be pruned. This problem is solved by considering floundering derivations as *potentially* successful, and giving a *potential* answer substitution. These substitutions 'cover' the semantically correct answers (which cannot be expressed as substitutions), but are possibly more general. A new completeness theorem for SLS-resolution, based on these

---

[1] In contrast, it is decidable whether a certain tree is an SLDNF-tree, but not by using SLDNF-resolution: one selection rule may result in an infinite SLDNF-tree, whereas another selection rule yields a finitely failed one.

potential answers, is proposed. The proof of this theorem requires generalized versions of the Mgu and Lifting Lemma (as presented in [Ca]).

In order to keep the potential answer substitutions as specific as possible, a selection mechanism is proposed that postpones floundering as long as possible. It appears that the restriction to these selection rules allows us to prove a form of the 'independence of the selection rule' property, which is well-known for positive programs.

In Section 5.2 we formalize this approach. Loop checks for locally stratified programs are formally defined in Section 5.3. Apart from the replacement of SLD-derivations by SLS-derivations, the definitions hardly differ from those in Section 2.1. Only the effect of applying a loop check on an SLS-tree with justifications appears to be complicated.

In Section 5.4 the soundness (no potential answer is lost) and completeness (the search space becomes finite) of loop checks for locally stratified programs are studied. Soundness becomes even more important than it was in the positive case: if a loop check prunes a (potential) success in a subsidiary SLS-tree, then the 'parent' SLS-tree should be extended; this extension might contain unsound answers. It is shown that a top-down SLS-interpreter remains sound and complete when it is augmented with a sound loop check.

Finally, in Section 5.5 it is shown how loop checks for positive programs can be turned into loop checks for locally stratified programs. The main observation is that in locally stratified programs negative literals cannot give rise to a loop. Thus any loop is caused by positive literals and can be detected by a loop check for positive programs; the negative literals are simply removed. It is shown that this construction preserves the completeness of the loop checks. Soundness is not preserved for every possible loop check (a counterexample using a highly nontypical loop check is given), but for 'reasonable' loop checks, including the ones studied in Chapter 3, soundness is preserved.

5.2. Declarative and Procedural Semantics of General Programs

SLS-resolution

DEFINITION 5.2.1 (Local stratification).
Let P be a program. P is *locally stratified* if there exists a mapping *stratum* from

the set of ground atoms of L_P to the countable ordinals, such that for every clause $(H \leftarrow A_1,\ldots,A_m,\neg B_1,\ldots,\neg B_n) \in$ ground(P):
for $1 \leq i \leq m$, stratum(A_i) $\leq$ stratum(H) and
for $1 \leq i \leq n$, stratum(B_i) $<$ stratum(H). □

Obviously, stratified programs ([ABW]) and programs without negation (*positive* programs) are locally stratified. *From now on, only locally stratified programs shall be considered*, therefore we usually omit the qualification 'locally stratified'. Consequently, we assume that for every considered program a mapping *stratum*, satisfying the above requirements, is available.

DEFINITION 5.2.2.
Let P be a program. We extend the mapping *stratum* as follows.
1. For an atom A, not necessarily ground,
   stratum(A) = sup { stratum(A_0) | A_0 is a ground instance of A in L_P }.
2. For a negative literal $\neg$A, not necessarily ground,
   stratum($\neg$A) = stratum(A) + 1
3. For a goal G,
$$\text{stratum(G)} = \begin{cases} 0 & \text{if } G = \square, \\ \max \{ \text{stratum}(L_i) \mid 1 \leq i \leq n \} & \text{if } G = \leftarrow L_1,\ldots,L_n. \end{cases}$$ □

A *selection rule* determines which literal is selected in a goal of a derivation. A well-known problem concerning the 'negation as (finite) failure' rule is *floundering*: the selection of a nonground negative literal (cf. [Cl2], [L]). We assume that such a selection is avoided whenever possible.

DEFINITION 5.2.3.
A selection rule is *safe* if it never selects a nonground negative literal in a goal containing positive and/or ground negative literals. □

Following Przymusinski's presentation for stratified programs in [P2], we now define for a given program P and goal G the SLS-tree of P$\cup${G}, together with some related notions. The definition uses induction on stratum(G).

DEFINITION 5.2.4 (SLS-tree).

Let P be a program and G a goal. Let **R** be a fixed safe selection rule. Assume that SLS-trees have already been defined for goals H such that stratum(H) < stratum(G). We define the *SLS-tree* T *of* $P \cup \{G\}$ *via* ***R***. (In fact this tree is not uniquely defined, as the choice of the names of auxiliary variables is left free.)

The root node of T is G. For *any* node H in T, its immediate descendents are obtained as follows:

- if $H = \square$, then H has no descendents and is a success leaf.
- if **R** selects a nonground negative literal in H, then H has no descendents and is a flounder leaf.
- if **R** selects a positive literal L in H, then H has as immediate descendants: for every applicable program clause C in P, a goal K such that $H \Rightarrow_{C',\theta} K$ is a derivation step and C' is a (properly standardized apart) variant of C.

  If no program clauses are applicable, then H is a failure leaf.
- if **R** selects a ground negative literal $L = \neg A$ in H, then the SLS-tree T' of $P \cup \{\leftarrow A\}$ via **R** has already been defined.

  (Either some ground instance $B\gamma$ of an atom B in G depends negatively on A, therefore $stratum(G) \geq stratum(B) \geq stratum(B\gamma) > stratum(A)$; or $\neg A$ is an instance of a negative literal in G, so again stratum(G) > stratum(A).)

  T' is called a *side-tree of H* (or, *of T*). We consider three cases:
  - if all leaves of T' are failed, then H has only one immediate descendant, namely the goal $K = H - \{L\}$, i.e., the goal H with L removed (such a derivation step is denoted as $H \Rightarrow K$).
  - if T' contains a success leaf, then H has no immediate descendants and is a failure leaf.
  - otherwise, H has no immediate descendants and is a flounder leaf.

If T has a success (flounder) leaf then T is *successful* (*floundered*); hence an SLS-tree may be both successful and floundered. T is *failed* if *all* of its leaves are failed (note that a failed SLS-tree may contain infinite branches).

An *SLS-derivation* (*of* $P \cup \{G\}$) is an initial segment of a branch of an SLS-tree (of $P \cup \{G\}$). An SLS-derivation ending in a success (flounder) leaf is called *successful* (*floundered*). An SLS-derivation is *failed* if it is infinite or ends in a failure leaf. Otherwise it is called *unfinished*.

A successful SLS-derivation (or *SLS-refutation*) of $P \cup \{G\}$ yields a *computed answer substitution* σ in the same way an SLD-refutation does:

whenever in a refutation step a negative literal is selected, such a step does not contribute to the computed answer substitution. $G\sim\sigma$ is called the *computed answer* of the derivation.

An SLS-derivation or -tree of $P\cup\{G\}$ is *potentially successful* if it is successful or floundered. The *potential answer substitution* σ of a potentially successful SLS-derivation is again the sequential composition of the mgu's of the derivation (thus the potential answer substitution of a refutation coincides with its computed answer substitution). Its *potential answer* is again $G\sim\sigma$. □

Soundness and completeness of SLS-resolution

We need the following soundness and completeness results, which strengthen the results of [Ca], [KT] and [PP].

THEOREM 5.2.7 (Soundness and strong completeness of SLS-resolution).

Let P be a program and G a goal. Let M_P be the unique perfect Herbrand model of P as defined in [P1], but based on the canonical language of [Ku1]. Let ***R*** *be a safe selection rule and θ a substitution.*

i) If $G\sim\theta$ is a computed answer for $P\cup\{G\}$ then $\forall(G\sim\theta)$ is true in M_P.

ii) If $P\cup\{G\}$ has a failed SLS-tree, then $\neg\exists(G\sim)$ is true in M_P.

iii) If $\forall(G\sim\theta)$ is true in M_P, then there exists a potentially successful SLS-derivation of $P\cup\{G\}$ via ***R*** *giving a potential answer $G\sim\sigma \leq G\sim\theta$.*

iv) If $\neg\exists(G\sim)$ is true in M_P, then the SLS-tree for $P\cup\{G\}$ via ***R*** *is not successful.*

PROOF. iv) follows immediately from i) and ii) follows immediately from iii). i) and ii) are proved in [Ca, Theorem 5.3(i)]. So iii) remains to be proved.

We introduce the following terminology.

An SLS-derivation is *unrestricted* if instead of mgu's, arbitrary unifiers are used.

An (unrestricted) SLS-derivation is *grounded* if every goal in it is ground.

An *oracle* SLS-derivation differs from the standard SLS-derivation in the treatment of selected ground negative literals: such a literal $\neg A$ is removed if $A \notin M_P$ and the derivation fails if $A \in M_P$ (and floundering does not occur).

From this it follows that a grounded (oracle) SLS-derivation never flounders. Now assume that $\forall(G\sim\theta)$ is true in M_P. It can then be shown (similarly to other completeness proofs, e.g. in [Ca] and [KT]) that there exists a grounded

oracle SLS-refutation of $P \cup \{G\theta\gamma\}$ via **R**, where $\gamma = \{x_1/a_1,\ldots, x_m/a_m\}$ binds all variables $x_1,\ldots,x_m$ in $G\theta$ to new constants $a_1,\ldots,a_m$. (More precisely, these constants are added to L_P. Notice that P remains locally stratified under this extension of the Herbrand Universe. More importantly, the oracle in the oracle SLS-refutation uses the model M_P w.r.t. the extended Herbrand Universe. The use of the oracle replaces the more usual induction on *stratum* .)

In this grounded oracle SLS-refutation, we can textually replace the constants $a_1,\ldots,a_m$ by $x_1,\ldots,x_m$ again. Thus we obtain a 'derivation' of $P \cup \{G\theta\}$ of which the unifiers do not act on the variables of $G\theta$. However, it is possible that some a_i is replaced by x_i in a selected negative literal, causing this 'derivation' to flounder, in which case the rest of the derivation must be discarded. Thus we obtain a potentially successful unrestricted oracle SLS-derivation of $P \cup \{G\theta\}$ of which the unifiers do not act on the variables of $G\theta$.

Now we supply side-trees for the remaining successful oracle steps (in which a ground negative literal $\neg A$ is selected and removed). As in such a case $A \notin M_P$, from iv) it follows that the constructed side-tree, an SLS-tree of $P \cup \{\leftarrow A\}$ via **R**, is not successful. If it is failed, then we have found the desired side-tree. If it flounders, then again our derivation flounders at this point and the rest of it is discarded. So we obtain a potentially successful unrestricted SLS-derivation of $P \cup \{G\theta\}$, of which the unifiers do not act on the variables of $G\theta$.

Now we need the following generalizations of the well-known Mgu Lemma and Lifting Lemma (see e.g. Lemma 5.2 and 5.3 in [Ca]).

LEMMA 5.2.8 (Mgu Lemma). *Let P be a program and G a goal. Suppose that $P \cup \{G\}$ has a potentially successful unrestricted SLS-derivation using the unifiers $\theta_1,\ldots,\theta_n$. Then there exists a potentially successful SLS-derivation of $P \cup \{G\}$ using the mgu's $\theta'_1,\ldots,\theta'_m$, such that $G\sim\theta'_1,\ldots,\theta'_m \leq G\sim\theta_1,\ldots,\theta_n$ and $m \leq n$.*

PROOF. First the construction of the proof of the original Mgu Lemma can be applied, disregarding floundering. The resulting 'derivation' uses the mgu's $\theta'_1,\ldots,\theta'_n$, and $G\sim\theta'_1,\ldots,\theta'_n \leq G\sim\theta_1,\ldots,\theta_n$. It is a valid SLS-derivation up to the first selection of a nonground negative literal. At this goal (obtained after m steps if it exists, otherwise m = n) floundering occurs and the rest of the 'derivation' is discarded. The result is a potentially successful SLS-derivation with a potential answer $G\sim\theta'_1,\ldots,\theta'_m \leq G\sim\theta'_1,\ldots,\theta'_n \leq G\sim\theta_1,\ldots,\theta_n$; and $m \leq n$. □

LEMMA 5.2.9 (Lifting Lemma). *Let P be a program, G a goal and θ a substitution. Suppose that $P \cup \{G\theta\}$ has a potentially successful SLS-derivation using the mgu's $\theta_1,\ldots,\theta_n$. Then there exists a potentially successful SLS-derivation of $P \cup \{G\}$ using the mgu's $\theta'_1,\ldots,\theta'_m$, such that $G\sim\theta'_1,\ldots,\theta'_m \leq G\sim\theta\theta_1,\ldots,\theta_n$ and $m \leq n$.*

PROOF. First the construction of the proof of the original Lifting Lemma can be applied, disregarding floundering. The resulting 'derivation' uses the mgu's $\theta'_1,\ldots,\theta'_n$, and $G\sim\theta'_1,\ldots,\theta'_n \leq G\sim\theta\theta_1,\ldots,\theta_n$. It is a valid SLS-derivation up to the first selection of a nonground negative literal. At this goal (obtained after m steps if it exists, otherwise m = n) floundering occurs and the rest of the 'derivation' is discarded. The result is a potentially successful SLS-derivation with a potential answer $G\sim\theta'_1,\ldots,\theta'_m \leq G\sim\theta'_1,\ldots,\theta'_n \leq G\sim\theta\theta_1,\ldots,\theta_n$; and $m \leq n$. □

Applying these lemmas on the potentially successful unrestricted SLS-derivation of $P \cup \{G\theta\}$ proves the existence of a potentially successful SLS-derivation of $P \cup \{G\}$, giving a potential answer $G\sim\sigma \leq G\sim\theta$. □

Theorem 5.2.7 allows us to omit in further considerations the perfect model semantics: in order to show that a loop check respects this semantics it is sufficient to compare pruned SLS-trees with original, unpruned trees.

The following example shows why a stronger result than the one presented in [Ca] is needed here.

EXAMPLE 5.2.10.
Let P = { p(1) ←.
p(y) ← p(y),¬q(y).
q(1) ← ¬r(x). }.

Figure 5.2.1 shows an SLS-tree of $P \cup \{\leftarrow p(x)\}$ via the leftmost selection rule. Since the tree flounders, ordinary completeness results like the one in [Ca] cannot be used. However, a loop check might very well prune the goal ←p(x), ¬q(x).

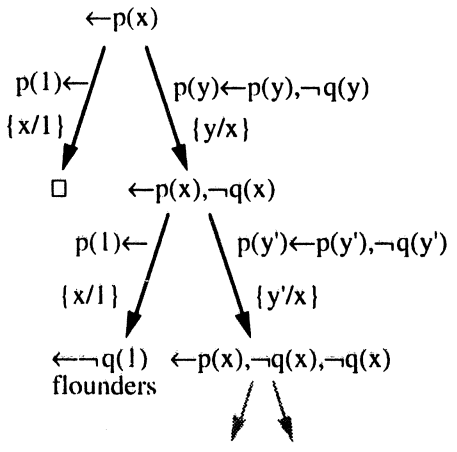


FIGURE 5.2.1

In this case the pruned tree does not flounder, so it is expected to be complete. Indeed this completeness follows from Theorem 5.2.7 (as the only potential answer occurring in the tree is p(1)). □

A more precise description of the search space

In the positive case, when a program P and a goal G are input to the interpreter, only an SLD-tree of P∪{G} is searched. However, in the presence of negation, not only an SLS-tree of P∪{G} is searched, but also its side-trees, and the side-trees of its side-trees, et cetera. We call such a construct consisting of an SLS-tree and its side-trees (to the required depth) a *justified SLS-tree*. As in Definition 5.2.4, induction on *stratum* is used.

DEFINITION 5.2.11 (Justified SLS-tree).

Let P be a program and G a goal. Let **R** be a fixed safe selection rule. A *justified SLS-tree* T of P∪{G} via **R** consists of an SLS-tree T_{top} of P∪{G} via **R**, which is, for every goal H in T_{top} in which a ground negative literal ¬A is selected, augmented with a justified SLS-tree T' of P∪{←A} via **R**. Such a tree T' is called a *justification of H* (or, *of T*), T_{top} is called the *top level* of T. T is successful (potentially successful, floundered, failed) if T_{top} is successful (potentially successful, floundered, failed). The computed/potential answers of T are those of T_{top}. □

Figure 5.2.2 shows a justified SLS-tree.

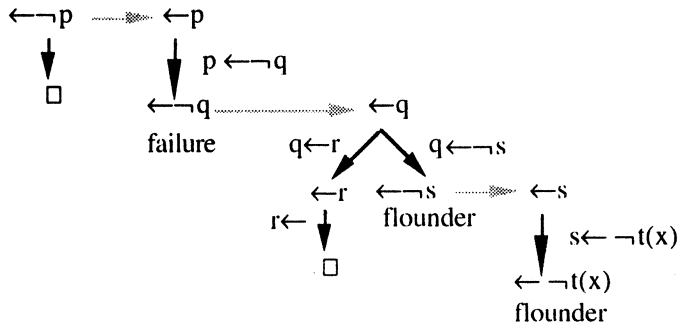


FIGURE 5.2.2

Notice the relationship between a side-tree T of H (an SLS-tree), and a justification J of H (a justified SLS-tree): T is the top level of J.

In order to render potential answers as specific as possible, it is worthwhile to 'postpone' floundering until all nonfloundering literals are resolved. This is achieved by considering the class of *deeply safe* justified SLS-trees defined below. The definition uses induction on *stratum* again.

DEFINITION 5.2.12.
A justified SLS-tree is *deeply safe* if for every flounder leaf $\leftarrow L_1,\ldots,L_n$ in it, every literal L_i ($1 \leq i \leq n$) is a negative literal $\neg A_i$, and either A_i is nonground or every deeply safe justified SLS-tree of $P \cup \{\leftarrow A_i\}$ flounders unsuccessfully. □

In a deeply safe justified SLS-tree, all justifications are also deeply safe (as the definition refers to *every* flounder leaf, not only those at the top level). Using a safe selection rule alone is not enough to obtain deeply safe trees: a ground negative literal $\neg A$ may be selected in a goal that still contains positive literals; then the side-tree of $\neg A$ may unsuccessfully flounder.

In this chapter, deeply safe trees are only used as a theoretical construct: Theorem 5.4.6 implies that the interpreter to which our loop checking mechanism is added can be allowed to use any safe selection rule. Nevertheless, it might prove profitable in practice to construct deeply safe trees, for this reduces the occurence of floundering to a minimum.

At first, it seems that checking whether 'every deeply safe justified SLS-tree of $P \cup \{\leftarrow A_i\}$ flounders' requires the construction of deeply safe trees of $P \cup \{\leftarrow A_i\}$ via every possible selection rule. The following lemma shows that this is not the case, as for deeply safe trees the independence of the selection rule holds.

LEMMA 5.2.13 (Independence of the selection rule for deeply safe trees). *Let P be a program and G a goal. Let T_1 and T_2 be deeply safe justified SLS-trees of $P \cup \{G\}$. Then there exists a bijection φ from the potentially successful branches in T_1 to the potentially successful branches in T_2 such that $|B| = |\varphi(B)|$ and the potential answers of B and $\varphi(B)$ are variants. Moreover, B is successful if and only if $\varphi(B)$ is.*

PROOF. Remove all negative literals from T_1 that are never selected (since T_1 is deeply safe, precisely these literals remain in the flounder leaves). The resulting tree is successful, hence the Switching Lemma (Lemma 3.3 in [KT]) can be applied repeatedly. In this way, a successful tree can be obtained in which the selections take place in the same order as in T_2. Now adding the 'floundering literals' in their place yields exactly T_2: because T_2 is deeply safe, the added literals are never selected before the flounder leaves.

Notice that induction on *stratum* is needed to show that whether a literal is a 'floundering literal' or not is independent of the selection rule. □

Therefore a valid method for creating deeply safe justified SLS-trees is to create only one (again deeply safe) justification for a selected negative literal. If this justification flounders unsuccessfully, then the literal is marked as 'floundering' and the interpreter 'backtracks' over this selection (that is, this selection is 'undone', and another literal is selected). Only if all literals in a goal are marked as 'floundering', the goal is a flounder leaf.

The final lemma of this section shows that in deeply safe trees the occurrence of floundering is indeed reduced to a minimum, i.e., given a program and a goal, every computed answer that can be obtained is present (modulo variants) in every deeply safe tree. Conversely, if the deeply safe tree has a floundering branch, such a branch is also present in every other tree (with a more general potential answer, indicating the possibility that the other tree flounders sooner).

LEMMA 5.2.14. *Let P be a program and G a goal. Let T_1 and T_2 be justified SLS-trees of $P \cup \{G\}$ and let T_1 be deeply safe.*

i) For every computed answer in T_2, T_1 contains a variant of it.

ii) For every potential answer in T_1, T_2 contains a more general potential answer.

PROOF. For both claims, we need to consider only the top-level of the trees, as the justifications can be treated by induction on *stratum*.

i) Suppose that T_2 contains a successful branch D. As far as D is concerned (without its justifications), T_2 is deeply safe. In other words, D can be embedded in a deeply safe justified SLS-tree T_3 of $P \cup \{G\}$. Now apply Lemma 5.2.13 on T_1 and T_3.

ii) Suppose that T_1 contains a potentially successful branch D. Consider a deeply safe justified SLS-tree T_3 of $P \cup \{G\}$ that follows the selections of T_2 as long as they are deeply safe. By Lemma 5.2.13, T_3 contains a potentially successful branch D' of which the potential answer is a variant of the answer of D. T_2 contains either D' or an initial segment of D' that flounders (on a goal in which the selection is not deeply safe). The potential answer of such an initial segment of D' is more general than the potential answer of D' itself. □

The approach to floundering we have sketched here tries to avoid floundering (by using deeply safe trees), and when floundering occurs, it tries to prove that the occurrence is harmless (i.e., that the returned potantial answer is less general than some computed answer). In the remaining cases, the potential answer can be used, but no attempt is made to get more information from the floundering goal.

Methods for trying to get more information, called constructive negation, have been studied both in the setting of SLDNF-resolution [Ch], [Dr] and SLS-resolution [Dr], [P3].

Both approaches are complementary. Because constructive negation is computationally expensive, it makes sense to limit its use to those cases where it is really needed. These cases are identified by our approach. In fact, as is shown by Example 5.5.12, it is almost unavoidable that the application of loop checking turns some successful SLS-trees into floundering ones, a behaviour which becomes more acceptable when constructive negation is used. On a technical level, one might expect a relation between Theorem 5.2.7 iii) and completeness results for constructive negation.

5.3. Loop Checks for Locally Stratified Programs

In this section we give a formal definition of loop checks for locally stratified programs (based on SLS-derivations), closely following the presentation of loop checks for positive programs in Chapter 2. The purpose of augmenting an interpreter with a loop check is to prune the generated search space while retaining its soundness and completeness. We define and study those properties of loop checks for locally stratified programs that are needed to achieve this goal.

Definitions

Since loop checks can be used to prune every part of a justified SLS-tree, one might define a loop check as a function on justified SLS-trees, directly showing where the trees are changed. However, this would be a very general definition, allowing practically everything. A first restriction we make is that a loop check acts only *within* an SLS-tree, disregarding its justifications and the possibility that this SLS-tree itself may be part of a justification in another SLS-tree. We shall formally call such loop checks for locally stratified programs *one level* loop checks. Nevertheless, we usually omit the qualification 'one level', unless confusion with *positive* loop checks (loop checks for positive programs, as defined in Section 2.1) can arise. This restriction leaves the possibility open that loop checks are used to prune more than one tree in a justified SLS-tree.

Similar to positive loop checks, we restrict the scope of one level loop checks even more, namely from SLS-trees to SLS-derivations. As in Section 1.3 and 2.1 we define:

- a node in an SLS-tree is *pruned* if all its descendants are removed.
- by pruning some of its nodes we obtain a pruned version of an SLS-tree (an *unfinished* SLS-tree).
- whether a node of an SLS-tree is pruned by a loop check depends only upon its ancestors in the tree, that is on the SLS-derivation from the root to this node.

So we define a one level loop check as a set of derivations (possibly 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) SLS-derivations L(P). Such a loop check L can be extended in a canonical way to a function $f_L{}^1$ from SLS-trees to unfinished SLS-trees: $f_L{}^1$ prunes in an SLS-tree of $P \cup \{G_0\}$ the nodes in $\{G \mid$ the SLS-derivation from G_0 to G is in L(P)$\}$. Extending L to a function $f_L{}^*$ from justified SLS-trees to 'pruned' justified SLS-trees, showing the effect of applying L to all SLS-trees within the original justified tree, is less straightforward, because pruning a justification can affect the 'parent' tree. This subject is discussed later in this section.

Again we also introduce the notion of a simple one level loop check, in which the set of pruned derivations is independent of the program. This leads us to the following definitions, which are almost identical to the ones in Section 2.1.

DEFINITION 5.3.1.
Let L be a set of SLS-derivations. *Initials(L)* = {D ∈ L | L does not contain a proper initial subderivation of D}. L is *subderivation free* if L = Initials(L). □

DEFINITION 5.3.2 (Simple one level loop check).
A *simple one level loop check* is a computable set L of finite SLS-derivations such that L is closed under variants and subderivation free. □

DEFINITION 5.3.3 (One level loop check).
A *one level loop check* is a computable function L from programs to sets of SLS-derivations such that for every program P, L(P) is a simple one level loop check. □

DEFINITION 5.3.4.
Let L be a loop check. An SLS-derivation D of P∪{G} is *pruned by* L if L(P) contains D or a proper initial subderivation of D. □

Pruning a justified SLS-tree
We now formalize how a justified SLS-tree is pruned. To simplify the definition, we assume that only one loop check L is used to prune a justified SLS-tree T: both the top level of T and (recursively) all justifications of T are pruned by L.

A problem arises when L prunes the justification of a goal G to such an extent that (potential) success in it is lost: instead of being a failure (flounder) leaf, G should now obtain a descendant, i.e., the search space of an interpreter with such a loop check *extends* the original search space beyond G. Modelling this additional search space is problematic, as there is no original tree to follow.

We avoid this problem temporarily by turning such a leaf G into an *extension leaf*. In this way the pruned tree remains a subtree of the original one. This property can be well exploited in the proof of the soundness and completeness of SLS-resolution with loop checking, where pruned trees are compared with original ones and Theorem 5.2.7 is used.

DEFINITION 5.3.5 (Pruning a justified SLS-tree).
Let P be a program and G a goal. Let L be a loop check and let T be a justified SLS-tree of P∪{G}. Then the tree $T_p = f_L^*(T)$, the pruned version of T, is defined as follows.

The root node of T_p is G. For *any* node H in the top level of T_p, the same literal as in T is selected; the immediate descendants of H in T_p are:

- if the SLS-derivation from G to H is pruned by L, then H has no descendants and is a *pruned leaf.*
- otherwise:
  - if a ground negative literal is selected in H, then H has a justification T' in T. The pruned version of T', $T_p' = f_L^*(T')$, is already defined by induction (on *stratum*). T_p' is the (pruned) justification of H in T_p. We consider the top level of T_p':
    - if it contains a success leaf, then H has no immediate descendants and is a failure leaf.
    - otherwise, if it contains a flounder leaf, then H has no immediate descendants and is a flounder leaf.
    - otherwise, if it contains an extension leaf or if H has no descendants in T, then H has no immediate descendants in T_p and is an *extension leaf.*
    - otherwise, H has in T_p the same immediate descendant as in T.
  - otherwise H has in T_p the same descendants (or the same leaf-type) as in T.

A pruned justified SLS-tree is *successful* (etc.) if one of its top level leaves is successful (etc.). It is *failed* if all its top level leaves are either failed or pruned.□

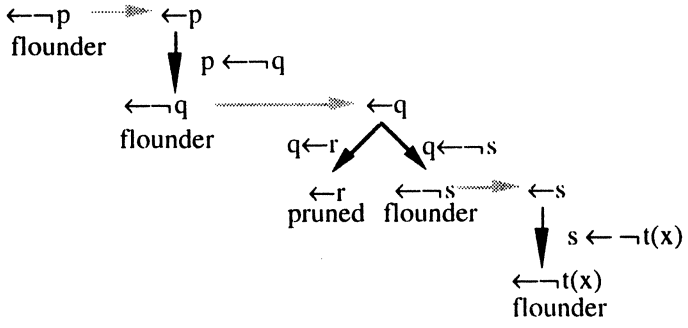


FIGURE 5.3.1

EXAMPLE 5.3.6.

When a loop check pruning the goal ←r is applied to the SLS-tree in Figure 5.2.2, the tree depicted in Figure 5.3.1 is obtained. When the goal ←s is also pruned, then the tree of Figure 5.3.2 is obtained. □

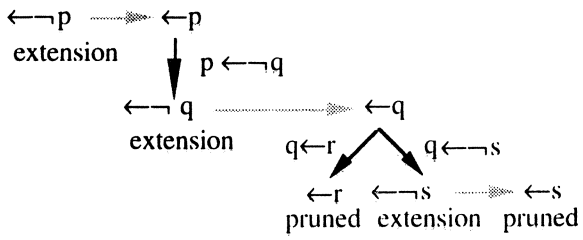


FIGURE 5.3.2

5.4. Soundness and Completeness

In this section a number of properties of one level loop checks is defined. These definitions are only concerned with the effect of applying a loop check on the top level of a justified SLS-tree. Thereafter the influence of applying loop checks satisfying these definitions on all levels of a justified SLS-tree is studied.

Definitions

As was pointed out before, using a loop check should not result in losing potential success. In order to retain completeness, an even stronger condition is needed: we may not lose any individual solution. Since Theorem 5.2.7(iii) involves potential answers, pruning the tree should preserve those successful and floundering branches that indicate (the possibility of) solutions not otherwise found. That is, if the original SLS-tree contains a potentially successful branch (giving some computed answer), then the pruned tree should contain a potentially successful branch giving a more general answer.

In order to consider only those potential answers that are as specific as possible, only deeply safe justified SLS-trees are taken into account. (Otherwise we would not be allowed to prune a floundering derivation like '$\leftarrow p \Rightarrow_{p \leftarrow p, \neg r} \leftarrow p, \neg r$', where $\neg r$ is selected and its side-tree flounders.)

DEFINITION 5.4.1 (Soundness).
Let **R** be a safe selection rule and let L be a loop check.

i) L is *weakly sound* if for every program P and goal G, and potentially successful deeply safe justified SLS-tree T of P∪{G}, $f_L^1(T_{top})$ is potentially successful.

ii) L is *sound* if for every program P and goal G, and deeply safe justified SLS-tree T of P∪{G}: if T contains a potentially successful branch giving a potential answer G~σ, then $f_L^1(T_{top})$ contains a potentially successful branch giving a potential answer G~σ' ≤ G~σ. □

The following lemma is an immediate consequence of these definitions.

LEMMA 5.4.2. *Every sound loop check is weakly sound.* □

Moreover, if the initial goal is ground (which is always the case for side-trees), then the notions weakly sound and sound coincide.

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 when every infinite derivation is pruned.

DEFINITION 5.4.3 (Completeness).
A loop check L is *complete w.r.t. a selection rule **R** for a class of programs C*, if for every program P ∈ C and goal G in L_P, every infinite SLS-derivation of P∪{G} via **R** is pruned by L. □

As in Section 2.2 we have overloaded the terms 'soundness' and 'completeness'. These terms refer both to loop checks and to interpreters (with or without a loop check). We now study how the soundness and completeness of a loop check affects the soundness and completeness of the interpreter augmented with it.

Interpreters and loop checks
We prove that under the right conditions an SLS-interpreter augmented with a loop check remains sound and complete (in the sense of Theorem 5.2.7). Due to the introduction of extension leaves, a pruned justified SLS-tree does generally *not* cover the entire search space for the SLS-interpreter augmented with a loop

check. For, whether a node is an extension leaf depends (partly) on the unpruned SLS-tree. This tree is not available for the loop-checked interpreter, so it cannot decide to stop at an extension leaf. Beyond an extension leaf, it might find incorrect answers. Therefore we must ensure the absence of extension leaves in soundness results.

As a first step, we do so for deeply safe justified SLS-trees by comparing their pruned and unpruned versions directly. The enumeration in this lemma links up with Theorem 5.2.7.

LEMMA 5.4.4. *Let P be a program and G a goal. Let* $\mathbf{R}$ *be a safe selection rule and* θ *a substitution. Let T be a deeply safe justified SLS-tree of* $P \cup \{G\}$ *via* $\mathbf{R}$*. Let L be a weakly sound loop check and let* $T_p = f_L^*(T)$*.*

-) T_P has no extension leaves.

i) If $G\sim\theta$ is a computed answer in T_p then $G\sim\theta$ is a computed answer in T.

ii) If T_p is failed, then T is failed.

iii) If L is sound and T contains a potential answer $G\sim\theta$, then T_p contains a potential answer $G\sim\sigma \leq G\sim\theta$.

iv) If T is not successful, then T_p is not successful.

PROOF. -) Suppose (in order to obtain a contradiction) that G is an extension leaf in T_p. Then a ground negative literal is selected in G. Let T' be the justification of G in T and let $T_p' = f_L^*(T')$ be the justification of G in T_p. By induction (on *stratum*), we may assume that T_p' has no extension leaves. So the only case left is that G is a leaf in T, and T_p' is failed. Obviously G is not a success leaf. So G is a failure leaf or flounder leaf in T. Hence T' is potentially successful. Since L is weakly sound and T' is deeply safe, we may conclude inductively from ii) that T_p' is potentially successful. Contradiction.

i) and iv) T_p is a subtree of T.

ii) Suppose (in order to obtain a contradiction) that T_p is failed, whereas T is potentially successful. Consider a potentially successful branch D in T. All justifications of D are either failed or floundered. Inductively by iv) the pruned justifications are also failed or floundered. Thus T_p can only be failed if D itself is pruned by L. This holds for every potentially successful branch in T, thus $f_L^1(T_{top})$ is failed. However, since L is weakly sound and T is deeply safe and potentially successful, by Definition 5.4.1(i) $f_L^1(T_{top})$ must be potentially successful. Contradiction.

iii) As ii), considering a branch only (potentially) successful if its potential answer is more general than G~θ. (Notice that if a failed justification of D in T is replaced by a floundering pruned justification in T_p, the potential answer of the remaining part of D in T_p is more general than the potential answer of D.) In this case Definition 5.4.1(ii) must be used. □

Indeed, combining Lemma 5.4.4(-) and (i)-(iv) with Theorem 5.2.7(i)-(iv) gives the required soundness and completeness results for deeply safe trees. The following theorem shows that it is not really necessary to use deeply safe selections. Only some parts of the unpruned tree (which are never constructed by the interpreter, but just used for comparison reasons) must be deeply safe. We need one more definition.

DEFINITION 5.4.5.
A loop check L is *selection-independent*
if for every program P and for every D ∈ L(P),
{D' | D' differs from D only in the selection of the literal in its last goal}
⊆ L(P). □

The restriction to selection-independent loop checks is not a severe one. Intuitively, after the creation of a new goal the loop check is performed first. Only when no loop is detected a further resolution step is attempted; to this end a literal is selected. All loop checks defined in Chapter 3 are selection-independent.

THEOREM 5.4.6 (Soundness and strong completeness of SLS-resolution with loop checking).
Let P be a program and G a goal. Let ***R*** *be a safe selection rule and θ a substitution. Let T be a justified SLS-tree of P∪{G} via* ***R****. Let L be a weakly sound selection-independent loop check. Then there exists a justified SLS-tree T' of P∪{G} such that:*
-) $T_p' = f_L^*(T')$ represents the search space for P∪{G} of a top-down SLS-interpreter using ***R****, augmented with the loop check L (i.e., T_p' has no extension leaves and makes all selections according to* ***R****, except for the selections in pruned leaves).*

i) If $G\sim\theta$ is a computed answer in T_p' then $G\sim\theta$ is a computed answer in T'.

ii) If T_p' is failed, then T' is failed.

iii) If L is sound and T' contains a potential answer $G\sim\theta$, then T_p' contains a potential answer $G\sim\sigma \leq G\sim\theta$.

iv) If T' is not successful, then T_p' is not successful.

v) If T is successful, then T' is successful.

vi) If T is failed, then T' is failed.

PROOF. First we give a construction of T'. By induction on *stratum*, we may assume that for each justification J of T, a justification J' is defined such that J' is derived from J as T' will be derived from T.

As a first step we obtain T'' by replacing in T every justification J by such a justification J'. Furthermore, if a floundering justification J of a leaf H is replaced by a failed justification J', then H has must obtain a descendant in T'' and T'' is expanded beyond H. This expansion takes place via **R**, except that the justifications in the expansion are still the ones inductively derived from the justifications via **R**. By v) and vi) this replacement of justifications cannot give rise to other problems.

For every justification J' in T'', it follows from -) that $f_L^*(J')$ has no extension leaves. Moreover, it follows that $T_p'' = f_L^*(T'')$ has no extension leaves. (For suppose that H is such an extension leaf, then the justification J' of H in T'' must be potentially successful, whereas $f_L^*(J')$ is failed. This contradicts ii), applied inductively on J'.)

We obtain the tree T' by expanding T_p'' beyond its pruned leaves, where at those pruned leaves and beyond, selections are made in a deeply safe way (thus: not necessarily via **R**). Notice that differences between T'' and T' do not occur before the selection in a goal where T'' is pruned, so by the assumption that L is selection-independent it follows that $T_p' = f_L^*(T') = T_p''$ (except possibly in selections in pruned leaves). Now we prove our claims.

-) For the justifications in T_p', this is true by induction. As was remarked, the top level of $T_p' = T_p''$ contains no extension leaves. Finally, the top level of T'' (and T_p'') follows **R**, so T_p' does (except possibly in pruned leaves).

i) and iv) T_p' is a subtree of T'.

ii) Suppose that T_p' is failed. Then T_p'' is failed, so apparently no floundering in the justifications of T_p'' reaches its top level. So we may 'pretend' that T_p'' is deeply safe, apart from selections in its pruned leaves (i.e., using Lemma

5.2.14 we could replace every justification of T_p" by a deeply safe one, without changing its top level). T' is an expansion of T_p" that is deeply safe in and beyond the pruned leaves of T_p". Thus in the same way, we may 'pretend' that T' is deeply safe. Then by Lemma 5.4.4(ii), T' is failed.

iii) First consider the tree T_{ds}, which is obtained from T' by expanding T' in a deeply safe way beyond every flounder leaf that is not deeply safe (either by making an other selection or by replacing the justification). Consider a potentially successful branch D in T' that is pruned in T_p'. As D is pruned in T_p", the tail (the part that is pruned out) of D in T' is already constructed in a deeply safe way. Therefore D occurs in T_{ds} unexpanded (w.r.t. T').

By its construction, we may again 'pretend' that T_{ds} is deeply safe. Thus if D yields a potential answer $G\sim\theta$, then, by Lemma 5.4.4(iii) and assuming that L is sound, $f_L^*(T_{ds})$ yields a potential answer $G\sim\sigma' \leq G\sim\theta$. The branch D' giving this answer $G\sim\sigma'$ is either fully present in T' ($\sigma = \sigma'$) or an initial fragment of it is present which flounders (giving a potential answer $G\sim\sigma \leq G\sim\sigma'$). D' cannot be pruned in T_p', because a goal pruned in T_p' is also pruned in $f_L^*(T_{ds})$ (as T_{ds} is an expansion of T', and L is selection-independent).

v) Consider a successful branch D in T. All its justifications are failed. So from vi) it follows inductively that D is still present in T'". If D is not pruned in T_p", then it is present in T'. If D is pruned in T_p", then in T' it is extended beyond the pruned goal in a deeply safe way. By Lemma 5.2.14(i) this extension is successful.

vi) If T is failed, then T has no floundering justifications. So inductively by v) and vi) the top levels of T and T" are identical. T_p" may contain pruned leaves, but by Lemma 5.2.14(ii) expanding them again in a deeply safe way can only lead to failure again. □

Thus combining Theorem 5.4.6(-) and (i)-(iv) with Theorem 5.2.7(i)-(iv) (applied on T') gives the final soundness and completeness results. However, the (loop-checked) interpreter need not be effective: in general traversing infinite justifications is required. Any real interpreter can only traverse a finite part of a (justified) SLS-tree, and is therefore incomplete, unless the use of the loop check has resulted in a finite search space.

THEOREM 5.4.7. *Let P be a program and G a goal in Lp. Let L be a loop check. Let* ***R*** *be a safe selection rule, let T be a justified SLS-tree of P∪{G} via* ***R*** *and let $T_p = f_L^*(T)$. If L is complete w.r.t.* ***R*** *for a class of programs C containing P, then T_p is finite.*

PROOF. The theorem follows immediately from Definition 5.4.3. □

Applying Theorem 5.4.7 on the tree T' as constructed in Theorem 5.4.6 shows that using a *complete* loop check (on all levels) ensures that the pruned justified SLS-tree is finite. If also the conditions of Theorem 5.4.6 are met (thus excluding extension leaves), then it follows that indeed the search space of the interpreter is finite. In this case the interpreter is *really* sound and complete.

Finally we must answer the question: 'Can the use of a (sound) loop check introduce floundering?'. The answer depends on what is exactly meant by the word 'introduce'. The following theorem indicates that the pruned tree can only flounder if somewhere in the original tree (but not necessarily at the top level) floundering occurs.

THEOREM 5.4.8. *Let P be a program and G a goal in Lp. Let L be a loop check. Let* ***R*** *be a safe selection rule, let T be a justified SLS-tree of P∪{G} via* ***R*** *and let $T_p = f_L^*(T)$. If a flounder leaf occurs in T_p, then a flounder leaf occurs in T.*

PROOF. Suppose that G is a flounder leaf in T_p, so a negative literal is selected in G. If this negative literal is not ground, then G itself is a flounder leaf in T. Otherwise, let T' denote the justification of G in T, and let $T_p' = f_L^*(T')$. T_p' must be floundered. By induction (on *stratum*), a flounder leaf occurs in T', hence in T. □

The fact that a flounder leaf occurs in T does not imply that T flounders: the flounder leaf might occur in a successful justification. If the pruned version of this justification is not successful (but only potentially successful), then it is possible that T is failed, whereas its pruned version flounders. Example 5.5.11 illustrates this effect. The tree in Figure 5.5.2 is successful; for most of the loop checks applied there its pruned version is not. Thus if this tree is the side-tree of a goal ←¬p, then the application of one of these loop checks turns ←¬p from a failure leaf into a flounder leaf.

Nevertheless, the result of Theorem 5.4.8 is significant. It guarantees that if the program P does not cause floundering (i.e., for every ground goal G, an SLS-tree of P∪{G} via a safe selection rule does not flounder; a property that is undecidable but for which sufficient syntactical conditions are known), the use of a loop check does not cause floundering too. Notice that it is not required in Theorem 5.4.8 that the loop check used is weakly sound. But if it is not, the pruned tree might contain extension leaves and the interpreter might find an occurrence of floundering beyond such a leaf.

5.5. Deriving One Level Loop Checks from Positive Loop Checks

Definitions

In this section we show how one level loop checks can be derived from positive loop checks. Since a successfully resolved negative literal is simply removed from a goal, negative literals cannot give rise to loops. (Thanks to the fact that we consider only locally stratified programs, looping 'through negation' cannot occur.) Therefore the basic idea is to remove all negative literals in a derivation. Then an SLD-derivation remains, to which a positive loop check is applied.

NOTATION 5.5.1.

For every (goal- and program-) clause, program, SLS-derivation and -tree X, X^+ denotes the object obtained from X by removing all negative literals. Thus if X is an SLS-derivation or -tree, then in X^+ every derivation step $G \Rightarrow H$ in which a negative literal is selected in G is deleted, since in this case $G^+ = H^+$. □

Notice that for every SLS-derivation D of P∪{G}, D^+ is an SLD-derivation of $P^+ \cup \{G^+\}$. For an SLS-tree T of P∪{G}, T^+ is an unfinished SLD-tree of $P^+ \cup \{G^+\}$ (due to failure or floundering of a negative selected literal in T, T^+ is not necessarily completed).

In fact the above definition is not completely precise: suppose that in the last goal G of an SLS-derivation D, a negative literal is selected. Then it is not clear which atom is selected in G^+ in D^+. Nevertheless, as the positive loop checks we are interested in are all selection-independent (Definition 5.4.5 does also apply to positive loop checks) we do not need to be more precise.

DEFINITION 5.5.2.

Let L be a positive loop check. *The one level loop check derived from L,* $O_L = \lambda P.\text{Initials}(\{D \mid D \text{ is an SLS-derivation and } D^+ \in L(P^+)\})$. □

The following lemmas establish the required relationships between a positive loop check and the one level loop check derived from it.

LEMMA 5.5.3. *For every positive loop check L, O_L is a one level loop check. Moreover, O_L is simple iff L is simple.*

PROOF. Immediately by the definitions. □

LEMMA 5.5.4. *Let L be a positive loop check, D an SLS-derivation and P a program. D is pruned by $O_L(P)$ iff D^+ is pruned by $L(P^+)$.*

PROOF. D is pruned by $O_L(P)$ iff some initial part of D, $D_{in} \in O_L(P)$, iff some initial part of D^+, $D_{in}^+ \in L(P^+)$, iff D^+ is pruned by $L(P^+)$. □

Soundness

Unfortunately, as is shown in the following counterexample, it is not the case that a one level loop check derived from a (weakly) sound positive loop check (as defined in Section 2.2) is again (weakly) sound.

COUNTEREXAMPLE 5.5.5.

Let P = { p $\leftarrow$ q(1),q(2). (C1),
q(x) $\leftarrow \neg$ r(x). (C2),
q(2) $\leftarrow$ q(1). (C3),
r(2) $\leftarrow$. (C4) },

and let G = $\leftarrow$p.

P is (locally) stratified and Figure 5.5.1 shows an SLS-tree T of P$\cup${G} via the leftmost selection rule. Let D denote the successful branch in T.

Then D^+ is the SLD-derivation $\leftarrow p \Rightarrow_{(C1)} \leftarrow q(1),q(2) \Rightarrow_{q(x)\leftarrow} \leftarrow q(2) \Rightarrow_{(C3)} \leftarrow q(1) \Rightarrow_{q(x)\leftarrow} \square$. Even a *simple* sound loop check L might prune the goal $\leftarrow$q(1) in D^+: it is visible in the second step of D^+ that the clause q(x)$\leftarrow$ is present in P^+; this clause allows for a shorter way to refute $\leftarrow$q(2) than via (C3) and $\leftarrow$q(1).

Unfortunately, this shortcut fails in the SLS-tree because it introduces the literal $\neg r(2)$ instead of $\neg r(1)$, and $\neg r(2)$ fails. So O_L prunes D, hence O_L is not weakly sound (note that the tree is the top level of a deeply safe justified tree). □

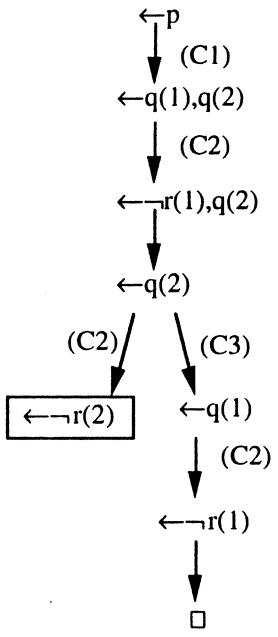


FIGURE 5.5.1

Although the loop check used in the counterexample formally satisfies the definitions, it is highly nontypical. We shall now show that more usual (weakly) sound positive loop checks, notably the ones defined in Chapter 3, derive again (weakly) sound one level loop checks. To this end we introduce a soundness condition, which is very similar (also in its proof) to Lemma 3.2.5.

LEMMA 5.5.6 (Soundness Condition). *Let L be a one level loop check. If, for every program P, goal G_0 and potentially successful branch $D = (G_0 \Rightarrow_{\theta_1} G_1 \Rightarrow \ldots \Rightarrow G_{k-1} \Rightarrow_{\theta_k} G_k \Rightarrow \ldots \Rightarrow_{\theta_m} H)$ $(0 < k \leq m)$ of a deeply safe justified SLS-tree T of $P \cup \{G_0\}$:*

[G_k is pruned by L] implies

[for some goal G_i $(0 \leq i < k)$ in D and for some $n < m-i$, there exists a potentially successful branch $G_i \Rightarrow_{\sigma_1} \ldots \Rightarrow_{\sigma_n} H'$ of a deeply safe justified SLS-tree of $P \cup \{G_i\}$],

then L is weakly sound.

Moreover, if also $G_0\theta_1 \ldots \theta_i\sigma_1 \ldots \sigma_n \leq G_0\theta_1 \ldots \theta_k\theta_{k+1} \ldots \theta_m$ is implied, then L is sound.

PROOF. First we focus on the weakly sound case. Let P be a program, G_0 a goal and T a deeply safe justified SLS-tree of $P \cup \{G_0\}$. Suppose that T_{top} contains a potentially successful branch $D = (G_0 \Rightarrow_{\theta_1} G_1 \Rightarrow \ldots \Rightarrow G_{i-1} \Rightarrow_{\theta_i} G_i \Rightarrow \ldots \Rightarrow G_{k-1} \Rightarrow_{\theta_k} G_k \Rightarrow \ldots \Rightarrow_{\theta_m} H)$ that is pruned by L at G_k. We use here induction on m, i.e., we assume that for every successful branch B in T_{top} shorter than D, $f_L{}^1(T_{top})$ contains either B or a potentially successful branch shorter than B.

We prove that $f_L{}^1(T_{top})$ contains a potentially successful branch D' that is shorter than D. By assumption a potentially successful SLS-derivation $D_1 = (G_i \Rightarrow_{\sigma_1} \ldots \Rightarrow_{\sigma_n} H')$ of $P \cup \{G_i\}$ exists. Adding (a properly renamed version of) D_1 to the initial part of D gives the derivation $D_2 = (G_0 \Rightarrow_{\theta_1} G_1 \Rightarrow \ldots \Rightarrow G_{i-1} \Rightarrow_{\theta_i} G_i \Rightarrow_{\tau_1} \ldots \Rightarrow_{\tau_n} H')$. By the independence of the selection rule (Lemma 5.2.13), T_{top} contains a branch D_3 such that $|D_3| = |D_2|$ and the potential answers of D_3 and D_2 are variants. Since D_3 is shorter than D ($|D_3| = i+n < i+(m-i) = m = |D|$), by the induction hypothesis $f_L{}^1(T_{top})$ contains either $D' = D_3$ or a potentially successful branch D' shorter than D_3, which proves the claim.

For the sound case, it remains to prove that $G_0\sigma' \leq G_0\theta_1 \ldots \theta_m$, where σ' is the potential answer substitution of D'. First we strengthen the induction hypothesis: for every potentially successful branch B in T_{top} shorter than D giving a potential answer $G\sim\sigma$, $f_L{}^1(T_{top})$ contains either B or a potentially successful branch shorter than B, giving a potential answer $G_0\sim\sigma' \leq G_0\sim\sigma$.

Then either since $D' = D_3$ or by the new induction hypothesis, and since the potential answers of D_3 and D_2 are variants, $G_0\sigma' \leq G_0\theta_1 \ldots \theta_i\tau_1 \ldots \tau_n \leq G_0\vartheta_1 \ldots \theta_i\sigma_1 \ldots \sigma_n \leq G_0\theta_1 \ldots \theta_m$. □

Indeed, the one level loop checks derived from the positive loop checks defined in Chapter 3 satisfy the above soundness condition. So we can prove that they are (weakly) sound.

THEOREM 5.5.7 (Soundness of Conversion).

i) The one level loop checks derived from the equality, subsumption and context checks based on goals are weakly sound.

ii) The one level loop checks derived from the equality, subsumption and context checks based on resultants are sound.

PROOF (Sketch). The proofs of Theorem 3.2.6, 3.3.7 and 3.4.6, in which it is shown that the positive loop checks mentioned satisfy the soundness condition (for the positive case), are straightforwardly generalized to the present case. Every successful SLD–derivation must be replaced by a potentially successful branch of a deeply safe justified SLS-tree. The Mgu Lemma, Lifting Lemma and Independence of the Selection Rule of [L] (used in the positive case) must be replaced by Lemma 5.2.8, 5.2.9 and 5.2.13 respectively. □

Completeness

Since some completeness properties of positive loop checks depend on the selection rule used, these selection rules are adapted to the presence of negation.

DEFINITION 5.5.8.

Let **R** be a selection rule for SLD-derivations.

An *extension of* ***R*** is a selection rule **R'** for SLS-derivations such that for every SLS-derivation D via **R'**, D^+ is an SLD-derivation via **R**. □

Unlike soundness, completeness carries over from positive to one level loop checks without much difficulty.

THEOREM 5.5.9 (Completeness of Conversion). *If L is complete w.r.t. a selection rule* ***R*** *for a class of programs C, then* O_L *is complete w.r.t. any safe extension of* ***R*** *for the class of programs* $\{ P \mid P^+ \in C \text{ and } L_P = L_{P^+} \}$.

PROOF. Let P be a program, G a goal in L_P (both possibly containing negative literals) and **R** a selection rule for SLD-derivations. Let **R'** be an arbitrary safe extension of **R**. Let D be an infinite SLS-derivation of $P \cup \{G\}$ via **R'**. Then D^+ is an infinite SLD-derivation of $P^+ \cup \{G^+\}$ via **R**. Let L be a positive loop check that is complete w.r.t. **R** for (a class of programs containing) P^+: for every goal H in L_{P^+}, every infinite SLD-derivation of $P^+ \cup \{H\}$ is pruned by $L(P^+)$. Since G^+ is a goal in $L_P = L_{P^+}$, D^+ is pruned by $L(P^+)$. Hence by Lemma 5.5.4, D is pruned by $O_L(P)$. □

Notice that the requirement $L_P = L_{P^+}$ is just a technicality which can be met easily by adding some irrelevant clauses to P. Combining the completeness results presented in Chapter 3 and 4 with Theorem 5.5.9 yields the following results.

COROLLARY 5.5.10.

i) The one level loop checks derived from the equality, subsumption and context checks are complete w.r.t. any safe extension of the leftmost selection rule for locally stratified function-free programs P such that P^+ *is a restricted program.*

ii) The one level loop checks derived from the subsumption and context checks are complete for locally stratified function-free programs P such that P^+ is an nvi program or an svo program.

iii) The one level loop checks derived from the subsumption and context checks are complete w.r.t. any safe extension of the leftmost selection rule for locally stratified function-free programs P such that P^+ is an nr-extended nvi-program, a chain-restricted svo program or a head-restricted svo program. □

The following example illustrates the application of several one level loop checks derived from positive loop checks defined in Chapter 3.

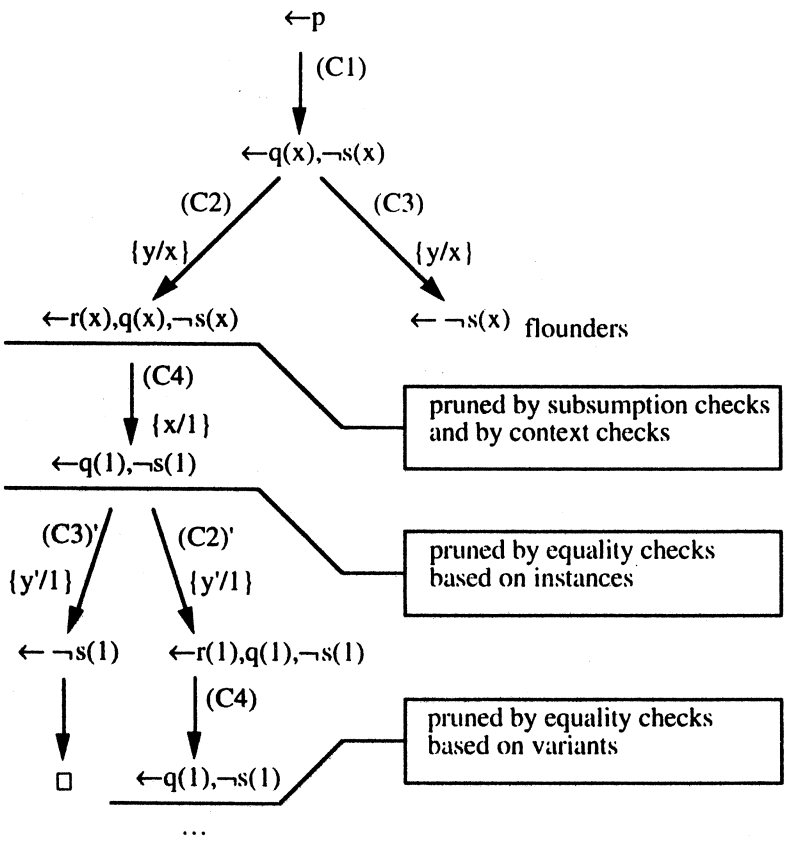


FIGURE 5.5.2

EXAMPLE 5.5.11.

Let P = { p ← q(x),¬s(x). (C1),
q(y) ← r(y),q(y). (C2),
q(y) ←. (C3),
r(1) ←. (C4) },

and let G = ←p. In Figure 5.5.2 an SLS-tree T of P∪{G} via (a safe extension of) the leftmost selection rule is depicted. It is shown where T is pruned by various loop checks.

For every loop check used, the pruned tree is finite. This was to be expected, as P^+ is a restricted program. Furthermore, each loop check retains potential success in the pruned tree (they even retain the most general potential answer, since G is ground). However, it appears that only the equality checks based on variants retain a successful branch. Obviously the extra instantiation in this branch, which was superfluous in the positive case, serves here to prevent floundering. □

But the equality checks based on variants do not always retain at least one successful branch, as is shown in the following example. As we remarked at the end of Section 5.2, the use of constructive negation can make this behaviour more acceptable.

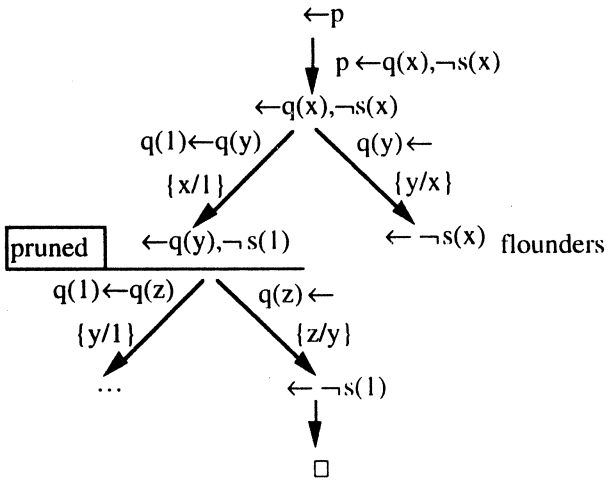


FIGURE 5.5.3

EXAMPLE 5.5.12.

Consider the program {p←q(x),¬s(x). q(1)←q(y). q(y)←.} and the goal ←p (see Figure 5.5.3). In order to avoid floundering of ←¬ s(x), the clause q(1)←q(y) must instantiate it to ¬s(1). But the resulting refutation is pruned by the one level loop check derived from EVR_L.

6. Loop Checking in Partial Deduction

This chapter introduces an alternative application of loop checks, namely in partial deduction. In Section 6.1 we recall the framework for partial deduction presented in [LS], illustrated by an example. In this framework, partial deduction involves the creation of SLD-trees[1] for a given program and some goals, up to certain halting points.

In the literature it is often noted that the problem of finding good halting points (loop prevention) is '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 identified as being 'difficult', or for practical purposes ad hoc solutions were used.

In the previous chapters we presented a framework for the analysis of loop checking mechanisms, 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 chapter is to show that this framework is sufficiently general for describing loop checks suitable for partial deduction as well.

In Section 6.2 we show that halting criteria for partial deduction can indeed be described as loop checks, but that their characteristics are different from the 'traditional' loop checks used at run-time. More precisely, a loop check used for partial deduction must first of all be complete, whereas in the traditional application the soundness of a loop check is more important.

We also show in Section 6.2 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 obtained by partial deduction. To this end the example of Section 6.1 is reconsidered. The addition of such a sound loop check

---

[1] In [LS], programs with negation are considered and SLDNF-resolution is used. As shown in Chapter 5, 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 there. In order to avoid unnecessary complications, we restrict ourselves here to positive programs and SLD-resolution.

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 and in previous chapters. Complete loop checks have not yet received that much attention. Section 6.3 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.

6.1. Partial Deduction

Although partial evaluation dates back to the 1970s, and was introduced into logic programming in the early 1980s ([K]), 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 extralogical 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.'

Below we define how exactly P' is derived from P and we formalize the requirement that G should have the same answers w.r.t. P and P'. We say that an unfinished SLD-derivation or -tree is *trivial* if it consists solely of an initial goal.

DEFINITION 6.1.1 (Partial deduction).
Let P be a program, A an atom and T a finite nontrivial 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 (notice that having a single atom as

initial goal implies that these resultants are Horn-clauses). The set $\{R_1,\ldots,R_r\}$ is called a *partial deduction for A in P.*

For a set of atoms $\mathbf{A} = \{A_1,\ldots,A_s\}$, 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 6.1.2 (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 ([Cl2]) 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 '$comp(P)\models\not\equiv A$' and '$comp(P)\not\models A$ and $comp(P)\not\models\not\equiv A$'. In terms of SLDNF-derivations, this relates to the distinction between finite and infinite failure.

It appears that partial deduction is always sound for positive programs, but only complete under a certain condition.

DEFINITION 6.1.3.
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 6.1.4. ([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'∪{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 6.2 this example reoccurs in combination with loop checking.

EXAMPLE 6.1.5.
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)]) ←.

Taking **A** = {solve([tc(x,c)])}, the SLD-tree of Figure 6.1.1 can be constructed (the resultants are given).

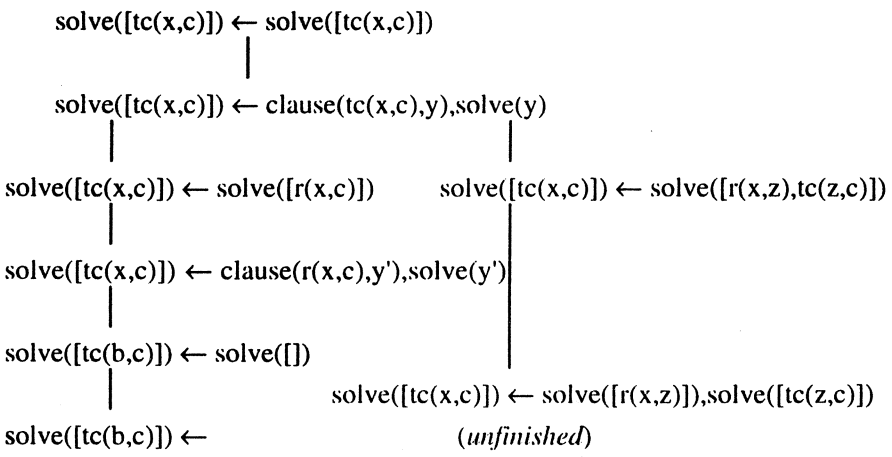


FIGURE 6.1.1

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 (Figure 6.1.2).

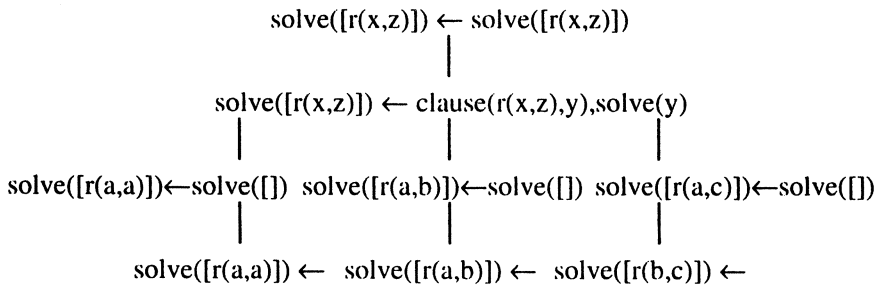


FIGURE 6.1.2

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.

Both questions relate to the termination of the partial deduction procedure. For the second one, this is obvious: if one of the SLD-trees is expanded infinitely deeply, then the procedure cannot terminate. However, if the expansion of an SLD-tree is stopped at an unfortunate moment, the resultant that is delivered might not be **A**-closed. When the 'missing' atoms are simply added to **A**, this requires the creation of more SLD-trees. These can, in turn, deliver new resultants that are not **A**-closed, and so on. Here we address only the second question, by relating the stopping criteria for the expansion of the SLD-trees used in partial deduction to loop checking.

6.2. 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. It ensures that the constructed tree is finite, thus enforcing termination of the partial deduction procedure (assuming that the closedness condition is reached within finite time). 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 *nontrivial*, 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 6.2.1 (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 nontrivial loop checks such that L_C is complete. Let $G_1,\ldots,G_r$ be the leaves of $f_{L_C+L_S}(T)$ (this unfinished SLD-tree is obviously finite and nontrivial) 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. □

THEOREM 6.2.2. *Let P be a program, G a goal and **A** a finite set of atoms. Let L_S and L_C be two nontrivial 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 $P' \cup \{G\}$ is **A**-closed and L_S is sound, then P' is complete w.r.t. P and G.*

PROOF. i) The tree $f_{L_C+L_S}(T)$ in Definition 6.2.1 is precisely the finite nontrivial SLD-tree required in Definition 6.1.1. 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 $P' \cup \{G\}$ is also a correct answer for $P'' \cup \{G\}$, and hence by Theorem 6.1.4 also for $P \cup \{G\}$.

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

For $|D| = 0$, i.e., $G = \square$, 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 $P \cup \{G\}$ is **A**-closed; say $A_1 \in \mathbf{A}$ and $A_1\gamma = A$. The steps in the refutation of $P \cup \{G\}$ in which A and its derived atoms are selected, constitute a refutation of $P \cup \{\leftarrow A\}$. Hence the completed version of the SLD-tree of $P \cup \{\leftarrow A_1\}$ 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 $A\sigma$), 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 $P \cup \{\leftarrow A_1\}$ 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 $P \cup \{G\}$: 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 $G\sim\sigma" \leq G\sim\sigma$.

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 $G\sigma\sigma^* \leq G\sigma\sigma \leq G\theta$.

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 6.1.5. Especially the effect of the addition of a sound loop check is remarkable.

EXAMPLE 6.2.3.

Suppose that the SLD-tree of $P \cup \{\leftarrow solve([tc(x,c)])\}$ of Example 6.1.5 had not been finished at the resultant solve([tc(x,c)])←solve([r(x,z)]),solve([tc(z,c)]), but continued as shown in Figure 6.2.1.

The resultant solve([tc(a,c)])←solve([tc(a,c)]) could well be pruned by a sound loop check, e.g. EIR_L (in fact it is pruned by all loop checks defined in Chapter 3 that are based on instances). The two other resultants could be pruned by some complete, but unsound loop check L_c (see Section 6.3).

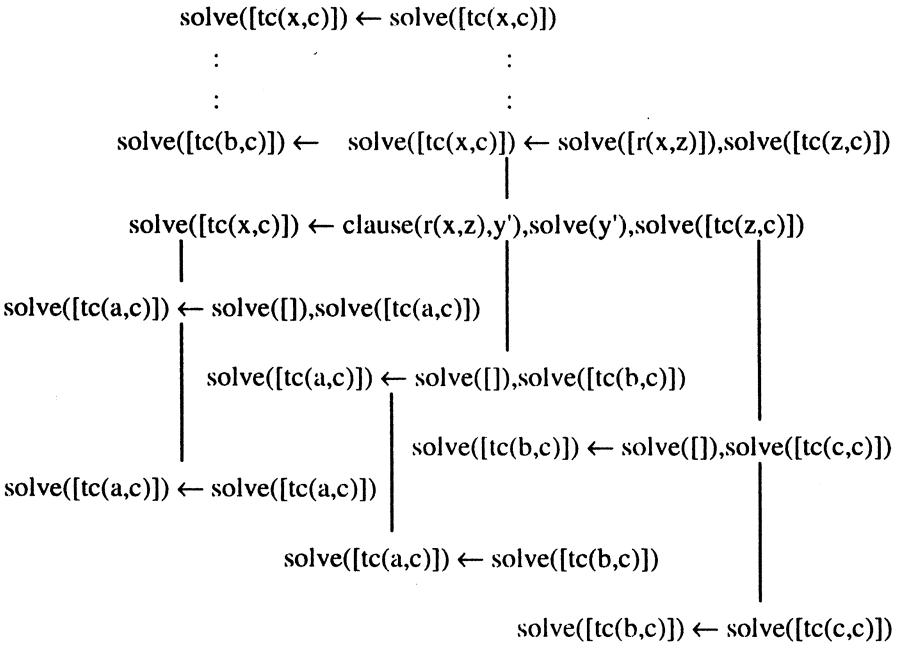


FIGURE 6.2.1

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)]).

In contrast to Example 6.1.5, where **A** had to be extended, P_3 is already **A**-closed. So by Theorem 6.2.2, 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. □

6.3. Complete Loop Checks

The previous chapters and most papers on loop checking 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.

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 (they are *selection-independent* , see Definition 5.4.5).

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: $A \leq B$ and $A' \leq B'$).

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 $\leftarrow p(a)$.

It would be too simple to say that these loop checks are too weak for this program: if a stronger loop check prunes the derivation arising from this program and goal at some place, then the resultant delivered is not $\{p(a)\}$-closed. Continuing naively by adding the required atom, and doing so repeatedly, will never result in the closedness condition being satisfied. Thus, although it allows only finite SLD-trees to be produced, such a stronger loop check alone cannot enforce termination of the partial deduction procedure. The

solution is obviously to add sometimes (but when?) a more general atom to the set **A** than is strictly needed. But this solution cannot be applied if only the one infinite SLD-tree is created. So, also in this case, a complete loop check must be preferred.

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**.

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 6.3.1 (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) =

$\text{Initials}(\{G_0 \Rightarrow G_1 \Rightarrow \ldots \Rightarrow G_k \mid \text{for } 0 \leq i \leq k,\ A_i \text{ is selected in } G_i \text{ and}$

$\#\{i \mid 0 \leq i < k,\ \text{rel}(A_i) = \text{rel}(A_k),\ A_k \text{ is the result of resolving } A_i \text{ and } \text{size}(A_i) \leq \text{size}(A_k)\} \geq d\})$. □

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

REMARK 6.3.2.

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 6.3.2, 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 6.3.7 shows an application of these versions. We now prove that OSC is complete if *size* returns natural numbers.

THEOREM 6.3.3. *Let $d \geq 1$ and let for every atom A, size(A) $\in$ ¯. 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 similar to the one in Theorem 3.4.13, see Figure 3.4.4.) 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 6.3.4.
A set U is *well-quasi-ordered* under a quasi-ordering $\geq$ 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 6.3.5. *Let U be a well-quasi-ordered set under $\geq$ 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 $k > 0$, $j_k = end(row((u_i)_{i>j_{k-1}}))$. 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 $row((u_i)_{i>j_{p-1}})$ 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 6.3.6. *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 6.3.3. By Lemma 6.3.5 the sequence $(size(A_i))_{i\in I}$ 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

$U = \{(p,\underline{n}) \mid p$ is a predicate symbol with arity k and $\underline{n} \in {}^{-k}\}$ 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 ≥. Defining termsize as size was defined in version 2, the size of an atom $A = p(t_1,\ldots,t_k)$ is defined as

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

EXAMPLE 6.3.7.

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 0s 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 6.3.1 shows where the three versions of OSC prune the SLD-tree of P∪{←reverse([0,s(0),s(s(0)) | x], [ ], y)}.

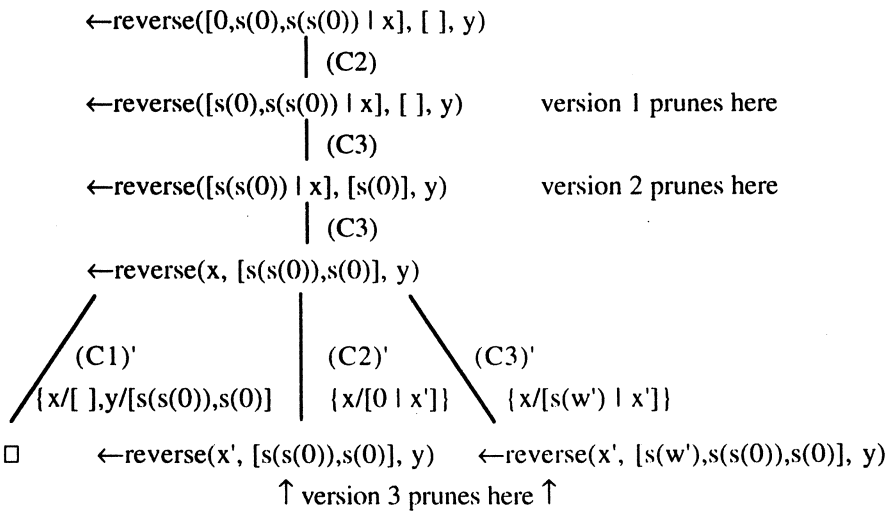


FIGURE 6.3.1

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 6.3.1 is superfluous. But normally it serves well to simplify the definitions of U, ≤ and size. Moreover it highlights that the Ove SizeCheck 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.

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 6.3.8.

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 $G_T = \{(G,i) \mid G$ 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 6.3.9.
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 6.3.10 ([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 6.3.10 this tree is finite, and hence T' is finite.

The only-if part of Theorem 6.3.10 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 nonrecursive predicate symbol, and the empty goal). They are not compared to any other goal.

DEFINITION 6.3.11.
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 6.3.12 ([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 ←q(...) producing a goal ←p(...),p(...) 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.

6.4. Conclusions

Summarizing, we have the following results.

- Loop prevention methods for partial deduction can be formulated within the framework of loop checking.

- 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.

- Nontermination of the partial deduction procedure can be caused by the creation of an infinite SLD-tree, but also by never reaching the closedness condition. 'Loop prevention' as discussed here only deals with the first cause.

- Sound loop checks can be added in a useful way to the partial deduction scheme, as outline in Section 6.2. This can result in the removal of loops from the generated program.

- This addition of a sound loop check does not agree with the completion semantics and SLDNF-resolution, but with the perfect model semantics and SLS-resolution, as was explained in Chapter 5.

- Further research on complete loop checks is required. In this respect, it is improtant 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.

Acknowledgement

This chapter benefitted from discussions with Kerima Benkerimi, John Gallagher, Jan Komorowski and Dan Sahlin.

7. Towards the Implementation of Loop Checking

Finally we shall pay attention to some practical aspects of loop checking. The considerations in this chapter could contribute to an efficient implementation of loop checks. 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. This means that the number of comparisons is quadratic in the number of goals generated. In practice this might turn out to be too expensive.

In Section 7.1 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 [vG1] 'tortoise-and-hare' technique, where each goal is compared with only one of its ancestors (namely the goal 'halfway'). Unfortunately, this technique does not preserve completeness. Here we propose to make a selection of goals (on account of their level in the SLD-tree) and to compare only selected goals. We prove that this solution preserves most completeness results when applied to the loop checks defined in Chapter 3. We study one selection in particular, namely the one that selects the goals of which the level in the SLD-tree is a triangular number. It appears that this selection renders the number of comparisons linear in the number of goals.

The theory of loop checking has been studied in the literature (for an overview see Chapter 8), but the practical implementation of such systems was hardly addressed (yet see [vG1]). In Section 7.2 we present two ways of implementing loop checks. The first implementation consists of a meta-interpreter. The second implementation avoids the extra layer of interpretation; instead it transforms the program given by the user to include loop checking. We give a global description of both implementations; more details about them can be found in [He]. We did not aim at writing maximally efficient implementations: our implementations must be seen as prototypes of future systems. However, the measurements we performed on our implementations seem to suggest that it is indeed possible to implement loop checking in an efficient way.

7.1. More Efficient Loop Checks

Introduction

Most loop checking mechanisms for logic programming proposed in the previous chapters and elsewhere in the literature are based on comparing goals: a derivation is pruned if 'sufficiently similar' goals are detected. In theory, a goal is usually compared with every previous goal in the derivation.

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, such as the mgu's used, the initial goal (for the loop checks based on resultants) 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 7.1.1 (Full comparison loop checks).

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

$$L(\varphi) = \text{Initials}(\{D \mid D = (G_0 \Rightarrow_{C_1,\theta_1} G_1 \Rightarrow \ldots \Rightarrow G_{k-1} \Rightarrow_{C_k,\theta_k} G_k) \text{ and for some } i < k \ \varphi(G_i,G_k,D) \text{ holds}\}),$$

where $\varphi(G_i,G_k,D)$ 'essentially compares 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 beginning of this chapter. For full-comparison loop checks, the number of comparisons performed is quadratic in the number of goals generated.

LEMMA 7.1.2 [Co1]. *On a finite SLD-derivation D, a full-comparison loop check performs* $\frac{1}{2}|D|(|D|+1)$ *comparisons.*

PROOF. Obvious. □

An interpreter equipped with a full-comparison loop check might 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.

We discuss in this section 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 loop checking criterion of the original loop check. For the new loop checks thus obtained we investigate the soundness, completeness and the number of comparisons performed (relative to the number of goals generated).

The first method, originally proposed by Van Gelder [vG1], is called the 'tortoise-and-hare technique'. Roughly speaking, this method compares every newly generated goal in a derivation with only one ancestor, namely the goal that is currently 'halfway' in the derivation. In this way the number of comparisons performed is clearly equal to the number of goals generated. Unfortunately, this construction does not preserve completeness in general.

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 [Co1]. It appears that the soundness and (in most cases) completeness of selected loop checks is independent of the selection.

The 'density' of the selection determines the efficiency of a selected loop check. The original full-comparison 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 [Co1], 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. At the end of this section we prove in detail that for a double selected loop check this effect is obtained with the selection $\{\frac{1}{2}i(i+1) \mid i \in \bar{}\}$ (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.

The tortoise-and-hare technique

A first attempt to reduce the number of comparisons performed by a loop check is presented in [vG1]. 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 7.1.3 (Tortoise-and-hare technique).

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

The following theorem is an immediate consequence of the previous definitions.

THEOREM 7.1.4 (Relative strength and soundness of L^{th}).

Let φ be a loop checking criterion.

i) $L(\varphi)$ is stronger than $L^{th}(\varphi)$.

ii) If $L(\varphi)$ is weakly sound (sound, shortening) then $L^{th}(\varphi)$ is weakly sound (sound, shortening).

PROOF. i) is obvious.

ii) follows from i) and the Relative Strength Theorem 2.2.12. □

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 d (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 d.

In [vG1] 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 loop check that is proposed in [vG1], 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 7.1.3 (and explicitly mentioned in [vG1]), 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 7.1.5 (Incompleteness of L^{th}).

Let P = { $p \leftarrow p.$ $p \leftarrow q.$
$q \leftarrow p.$ $q \leftarrow q.$}.

Let φ be a loop checking criterion such that for every derivation D: $\varphi(\leftarrow p,\leftarrow p,D) = \varphi(\leftarrow q,\leftarrow q,D)$ = true and $\varphi(\leftarrow p,\leftarrow q,D) = \varphi(\leftarrow q,\leftarrow p,D)$ = false (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. □

Selected loop checks

An easy generalization of Counterexample 7.1.5 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 7.1.6 (Selected loop checks).

Let φ be a loop checking criterion and let S be an infinite subset of $^-$.

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

$$L^1(\varphi,S) = \text{Initials}(\{D \mid D = (G_0 \Rightarrow_{C_1,\theta_1} G_1 \Rightarrow \ldots \Rightarrow G_{k-1} \Rightarrow_{C_k,\theta_k} G_k) \text{ and for some } k \in S \text{ and } i < k\text{: } \varphi(G_i,G_k,D) \text{ holds}\}).$$

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

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

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 $S = ^-$, we obtain the full-comparison loop checks again, for which the number of comparisons is quadratic in the number of goals generated. In the next subsection, the efficiency of double selected loop checks with $S = \{\frac{1}{2}i(i+1) \mid i \in ^-\}$ is studied in detail. For now, we do not consider any specific selection, but rather study the soundness and completeness of selected loop checks in general.

LEMMA 7.1.7 (Relative strength of selected loop checks).

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 $L(\varphi) = L^1(\varphi,^-) = L^2(\varphi,^-)$ 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 7.1.8 (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 7.1.7 and the Relative Strength Theorem 2.2.12. □

Combining this theorem with the soundness results for the simple loop checks presented in Chapter 3 yields the following results.

COROLLARY 7.1.9. *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 Equality Soundness Corollary 3.2.7, Subsumption Soundness Corollary 3.3.8, Context Soundness Corollary 3.4.7 and Theorem 7.1.8. □

Unfortunately, an equally general completeness result cannot be obtained using Lemma 7.1.7. Instead, generalizing the completeness results from the simple loop checks of Chapter 3 to the corresponding selected loop checks requires a detailed analysis of the completeness proofs in Chapter 3. (However, by Lemma 7.1.7 it suffices to consider only double selected loop checks.)

For the equality checks and subsumption checks, this generalization is straightforward. By definition, a full-comparison 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. But in the relevant proofs in Chapter 3 (notably Theorem 3.2.18, Lemma 3.3.16 and Theorem 3.3.20) a stronger result is proved: every infinite sequence of *unrelated* goals contains two 'similar' goals. Although in Chapter 3 this sequence is always taken $\{G_i \mid i \in {}^-\}$, the sequence $\{G_i \mid i \in S\}$ can be used for any selection S. Hence the completeness results for equality and subsumption

checks (Corollaries 3.2.19, 3.3.9, 3.3.18 and 3.3.21) generalize immediately to selected equality and subsumption checks.

THEOREM 7.1.10 (Completeness of selected equality and subsumption checks).

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 some of the completeness results is less straightforward, but still possible.

THEOREM 7.1.11 (Completeness of selected context checks).

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 (Theorems 3.4.13 and 3.4.15), an infinite sequence of goals $G_{m_0}, G_{m_1}, \ldots$ $(0 \leq m_0 < m_1 < \ldots)$ 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 $t_{i-1} \leq m < s\}$ 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 3.4.13 exactly one resolution step on A_i occurs between A_i and A_{i+1}, whereas here it can be more than one step.)

In the rest of the proof of Theorem 3.4.13 (and 3.4.15), 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 7.1.7 ($T \subseteq S$) and the Relative Strength Theorem 2.2.12, 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 7.1.12.

Let P = { $p(y) \leftarrow q(x),p(x)$.
$q(1) \leftarrow$. },
and let $G_0 = \leftarrow q(x_0),p(x_0)$.
Recall Definition 3.4.1 (the CIG check) and consider the following infinite SLD-derivation $D = (G_0 \Rightarrow_{C_1,\theta_1} G_1 \Rightarrow \ldots)$ of $P \cup \{G_0\}$ via the leftmost selection rule:

$\leftarrow q(x_0),p(x_0)$
↓ $q(1)\leftarrow$ $\{x_0/1\}$
$\leftarrow p(1)$
↓ $p(y_1)\leftarrow q(x_1),p(x_1)$ $\{y_1/1\}$
$\leftarrow q(x_1),p(x_1)$
↓ $q(1)\leftarrow$ $\{x_1/1\}$
$\leftarrow p(1)$
↓ $p(y_2)\leftarrow q(x_2),p(x_2)$ $\{y_2/1\}$
$\leftarrow q(x_2),p(x_2)$
...

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

This counterexample also shows that it is generally impossible to apply the Generalization Theorem 4.2.1 on completeness results for selected loop checks

(namely: the program used in the counterexample is an nr-extended version of the program $\{p(y) \leftarrow p(x)\}$ and the selected context checks are complete for the programs $\{p(y) \leftarrow p(x)\}$ and $\{p(y) \leftarrow p(1)\}$). The explanation is obviously that selected loop checks are not safe for initialization and detailing (see Definitions 4.1.6 and 4.1.7), as these operations affect the goals that are selected for comparison.

Nevertheless we can conclude that 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 subsection we take a more constructive attitude towards selected loop checks and we investigate how much is gained by using them.

Triangular loop checks: a case study

This subsection presents a detailed study of double selected loop checks with the selection $S = \{\frac{1}{2}i(i+1) \mid i \in \bar{}\}$. 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 7.1.13. *Let D be a finite SLD-derivation. The number of comparisons performed on D by a triangular loop check is at most* $|D|$.

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 $n = \frac{1}{2}i(i+1)$ $(0 \leq n \leq k)$, 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 $\sum_{i=0}^{j} i = \frac{1}{2}j(j+1) = k = |D|$. □

The following arrangement of goals may help the intuition:

$G_0 : G_1\ G_2\ G_4\ G_7\ \ldots$
$\quad\ G_3\ G_5\ G_8$
$\quad\quad G_6\ G_9$
$\quad\quad\ G_{10}$

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 7.1.14. *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·n comparisons on T.*

PROOF. We may assume that $b > 1$ (for $b = 1$ (or $b < 1$), see Theorem 7.1.13) 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 $\frac{1}{2}(j-1)(j-2)$; $1 + \frac{1}{5} + \frac{1}{25} + \ldots = \frac{5}{4}$).

Therefore $\frac{\text{the number of comparisons in T}}{\text{the number of goals in T}} = \frac{5}{4} \cdot \frac{j \cdot b^k}{n} = \frac{5}{4} \cdot \frac{j \cdot b^k \cdot (b-1)}{(b^{k+1}-1)} \approx \frac{5j(b-1)}{4b}$. (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 $j \leq \sqrt{-\frac{7}{4} + 2 \cdot {}^b\log(10^{10}(b-1))} - \frac{1}{2}$. A numeric analysis[1] of the function $\frac{5(b-1)}{4b} \cdot (\sqrt{-\frac{7}{4} + 2 \cdot {}^b\log(10^{10}(b-1))} - \frac{1}{2})$ 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.

THEOREM 7.1.15. *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,

---

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

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. □

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 7.1.12).

A minor disadvantage of the selection technique (compared to the tortoise-and-hare technique) might be that the comparisons are not distributed smoothly over the goals, which makes the timing of the interpreter less predictable.

An important question is the choice of the selection: the sparser the selection is, the more efficient the resulting 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. The optimal choice will definitely depend on the program that is interpreted. We shall briefly return to this subject at the end of this chapter.

In practice, it might pay off to postpone the invocation of a loop check until a loop is suspected, which is the case when a derivation is growing 'unusually long'. This technique can be formalized by using selections $S = \{i + N \mid i \in \bar{}\}$, or $S = \{\frac{1}{2}i(i+1) + N \mid i \in \bar{}\}$ for some 'large' number N. Obviously using such a selection does not affect the completeness of the loop check.

7.2. Two Simple Implementations of Loop Checking

In this section we give a global description of two simple implementations of loop checking (for more details we refer to [He]). The first implementation has the form of a meta-interpreter. Every part of the computation process, deriving new goals as well as loop checking, is programmed explicitly in C-PROLOG. This means that the program is slow, but allows to measure the time spent on loop checking separately from the time spent on finding the next goal.

In the second, more efficient implementation of loop checking the program given by the user is transformed into a new program that includes loop checking. Thus the additional layer of interpretation is removed. The search for applicable clauses and unification is now done by the underlying C-PROLOG system, but the computations performed for the loop check are still explicit. This inequality between the efficiency of generating the next goal and loop checking makes the implementation less suitable for judging the relative cost of loop checking.

Finally the results obtained from the two implementations are discussed. We address the question 'How costly is loop checking?'. This question suggests a comparison between existing PROLOG systems and an equally efficient system with loop checking. Although we did not build such an efficient implementation, our results indicate that it is indeed feasible to implement loop checking. But we should point out here that this conclusion relies on the assumption that such efficient implementations distribute their time similar to our implementations (that is: mostly on manipulating substitutions).

Of course some loop checks are more demanding than others. So we also give a comparison between the various loop checks that we implemented.

A loop checking meta-interpreter

A meta-interpreter seems to be the easiest way to incorporate loop checking in a logic programming interpreter. Due to the extra layer of interpretation, the resulting implementation is not very efficient. This section contains only a description of the meta-interpreter; a discussion of the results obtained from it is deferred to the end of this section.

First we describe how the meta-interpreter is used. The user adds his program to the meta-interpreter (as usual, the program must not use predicates

defined by the meta-interpreter) and presents a goal ←`solveI(Check,Goal)` to it. The representation of the program and `Goal` is discussed later.

The parameter `Check` specifies the loop check that is to be used. Currently we have implemented the equality and subsumption checks (list-versions) described in Chapter 3 (`evg`, `eig`, `evr`, `eir`, `svg`, `sig`, `svr`, `sir`) and their single triangular `(***,st)` and double triangular `(***,dt)` variants described in the previous section. Furthermore the user can choose not to use any loop check by specifying `empty` or `non`. The difference between the two is that `empty` collects data like all other loop checks (without using it), whereas `non` does not collect any data.

The procedure `solveI` merely initializes the data-structures and counters. Then an attempt to solve the goal is made. This attempt stops at every leaf of the (pruned) SLD-tree, giving the derivation leading to that leaf, the counters and, in case of a success leaf, the computed answer. The user can use backtracking of the underlying PROLOG-interpreter to find all leaves of the SLD-tree.

The main procedure of the meta-interpreter is `solve`. It has eight input parameters and one output parameter. When the derivation $G_0 \Rightarrow_{C_1,\theta_1} G_1 \Rightarrow \ldots \Rightarrow G_{k-1} \Rightarrow_{C_k,\theta_k} G_k$ has been constructed, these parameters have the following meaning.

Input for constructing the derivation:

`Goal`	: G_k (also used for loop checking),
`Subst`	: $\theta_1 \ldots \theta_k$, restricted to the variables of G_0,
`LastVar`	: the number of variables used (needed for standardizing apart).

Input for loop checking:

`Check`	: the loop check that is used,
`Result`	: $G_0\theta_1 \ldots \theta_k$ (so a *result* is the left-hand side of a *resultant*),
`ListGoal`	: $[G_{k-1}, G_{k-2}, \ldots, G_0]$,
`ListResult`	: $[G_0\theta_1 \ldots \theta_{k-1}, G_0\theta_1 \ldots \theta_{k-2}, \ldots, G_0]$,
`Depth`	: k.

When a double triangular loop check is used, `ListGoal` and `ListResult` contain only the goals (results) with a triangular index. When `non` is used, these lists are not maintained.

Output (provided that a finite derivation is generated):

`Derivation` : a derivation [G_{k+1}, G_{k+2}, ..., G_n, endmark],
where endmark = `true` means $G_n = \square$,
endmark = `prune` means G_n is pruned and
endmark = `[false|...]` means G_n is failed.

The four clauses for solve correspond to the four possible situations for the current goal: respectively success leaf, failure leaf, not a leaf and pruned leaf.

```
solve([],_,_,_,_,_,_,_,[true]) :- !.          /* success leaf */
solve([false|Goal],_,_,_,_,_,_,_,[]) :- !.    /* failed leaf */
solve(Goal,Subst,LastVar,Check,Result,ListGoal,ListResult,
      Depth,[NewGoal|Derivation]) :-
   check(Check,Goal,Result,ListGoal,ListResult,Depth,
         NewListGoal,NewListResult),
   !,          /* no loop has been detected: continued branch */
   find_new_goal_result(Goal,Result,Subst,LastVar,
                        NewGoal,NewResult,NewSubst,NewLastVar),
   NewDepth is Depth + 1,
   solve(NewGoal,NewSubst,NewLastVar,Check,NewResult,
         NewListGoal,NewListResult,NewDepth,Derivation).
solve([A|Goal],_,_,_,_,_,_,_,[prune])./* a loop: pruned leaf */
```

The first clause is obvious. The second clause deals with the case that the *previous* goal is a failed leaf. This is detected by the procedure `find_new_goal_result`, as it cannot find a clause of which the head unifies with the selected (leftmost) atom. It then replaces this atom by `false`. Thus the next goal of such a derivation is a goal of which the leftmost atom is `false`. We use this goal as the endmark of the derivation.

The third clause invokes the loop check. If `check` succeeds, i.e. if no loop is detected, then `find_new_goal_result` computes one derivation step, giving the new goal to be solved. The list of previous goals and results is updated by `check`: if a double selected loop check is used, then only selected goals and results are included in those lists. If `non` is used, no updating is done at all.

If the loop check in the third clause fails, a loop has been detected and the fourth clause applies. If the user does not want the interpreter to stop at failed

leaves, he can simply add `fail` after the cut in the second clause, enforcing automatic backtracking. Similarly, if the user does not want the interpreter to signal pruned leaves, he can omit the fourth clause of `solve`.

The procedure `check` selects the appropriate loop checking procedure and calls it with the relevant parameters. Some typical clauses are:

```
check(non,_,_,ListGoal,ListResult,_,ListGoal,ListResult).
check(empty,Goal,Result,ListGoal,ListResult,_,
          [Goal|ListGoal],[Result|ListResult]).
check(evr,Goal,Result,ListGoal,ListResult,_,
          [Goal|ListGoal],[Result|ListResult]) :-
      check_EVR(Goal,Result,ListGoal,ListResult).
```

The triangular loop checks are treated by four clauses, two for single and two for double triangular loop checks. We give here the clauses for the double triangular loop checks.

```
check((Full,dt),Goal,Result,ListGoal,ListResult,Depth,
                [Goal|ListGoal],[Result|ListResult]) :-
      I is sqrt(1+8*Depth),integer(I),!,
      /* the check is (***,dt) and Depth is triangular */
      check(Full,Goal,Result,ListGoal,ListResult,_,_,_).
check((Full,dt),_,_,ListGoal,ListResult,_,ListGoal,ListResult).
      /* the check is (***,dt) but Depth is not triangular */
```

First it is checked if Depth is a triangular number (it is easy to show that d is triangular if and only if $\sqrt{1+8d}$ is an integer). If this is the case then the first clause applies; the lists of goals and results are updated and the corresponding full-comparison check is invoked. If not, no check is performed and the lists of goals and results remain unchanged.

The meta-interpreter handles several objects occurring in SLD-derivations, like variables, goals and substitutions. Here we describe how all such items are represented in it.

In simple meta-interpreters, like the well-known 'vanilla'-interpreter, object variables are represented by variables of the meta-interpreter (meta-variables). In

this way, unification and the subsequent application of the unifier can be done implicitly by the underlying PROLOG-interpreter. For our purposes this approach is difficult to use (although it can be done): a variable that is bound by solving the current goal must remain a variable in the list of previous goals. Therefore an object-variable A is represented by the meta-constant $A. Consequently, the user must put a $-sign in front of all variables in the program and the goal. Furthermore, unification and the renaming of program clauses must be done explicitly by the meta-interpreter (hence the need for `LastVar`).

Object atoms become terms in the meta-interpreter (an object-predicate is a meta-function) and goals are represented as lists of (object-)atoms. The initial goal that is specified by the user must also be represented as a list, but the bodies of program clauses have the usual shape: they are transformed into lists by the meta-interpreter. A substitution $\{x_1/t_1,\ldots,x_n/t_n\}$ is represented by the list of bindings $[eq(x_1/t_1),\ldots,eq(x_n/t_n)]$.

Compiling loop checks into the program

In this section we present a more efficient implementation of loop checking. The program given by the user is transformed into a new program. Interpreting this new program by means of an ordinary PROLOG-interpreter has the same effect as using a loop-checked (meta-)interpreter. The transformation, which can be done automatically, consists of four parts.

1. A call to the predicate `loop_check` is added as the first literal in the body of every clause.
2. Extra parameters are added to every user-defined predicate. These parameters are needed to pass relevant data to the `loop_check` procedure.
3. A clause is added to augment the query specified by the user (with only the 'normal' parameters and the loop check to be used) with the proper initializations of the new parameters. For simplicity it is assumed that the user's query will always consist of *one atom* of a *fixed predicate*.
4. The procedure `loop_check` and its auxiliary procedures are added to the program.

We shall now treat these modifications in more detail.

The call to the predicate `loop_check` that is added to a clause C checks whether, given the preceding derivation, applying C would lead to a loop. If indeed a loop is detected then the call fails (with the side-effect that the user is

informed), thus preventing C to be used. If no loop is detected then the call succeeds and the output parameters are filled with updated versions of the required data. A special case occurs when an empty goal (a solution) is found: the user is informed and loop checking is not necessary.

The required data strongly resemble the data used by `check` in the meta-interpreter, but there are some notable differences.

1. The procedure `check` gets the current goal as an input parameter. When `loop_check` is called in a clause C, the current goal (i.e. the goal that would result from applying C) is assembled from the previous goal and the body of C. The current goal is returned in an output parameter.
2. The counter `Depth` is updated by `loop_check`. The new value is also returned in an output parameter.

Thus `loop_check` has as input parameters: `Check` (the loop check), `Body` (the body of the clause), `Goal` (the *previous* goal), `Result` (the current result), `ListGoal`, `ListResult` (as in `check`) and `Depth` (not yet updated, thus one less than in `check`). Its output parameters are `CurrentGoal`, `NewListGoal`, `NewListResult` and `NewDepth`.

The procedure `loop_check` consists of the following three clauses, dealing with respectively the empty goal, no loop being detected and a loop being detected. Instantiating the output variables is only needed in the second case.

```
loop_check(_,[],[H],_,ListGoal,_,_,_,_,_,_) :-
      !, /* a success leaf is reached: no loop check needed */
      nl, write('true'), nl, write([[]|ListGoal]).
loop_check(Check,Body,[H|Goal],Result,ListGoal,ListResult,Depth
            ,CurrentGoal,NewListGoal,NewListResult,NewDepth) :-
      NewDepth is Depth + 1,
      append(Body,Goal,CurrentGoal),
      check(Check,CurrentGoal,Result,ListGoal,ListResult,
                  NewDepth,NewListGoal,NewListResult),
      !. /* no loop has been detected */
loop_check(_,Body,[H|Goal],_,ListGoal,_,_,_,_,_,_) :-
      /* a loop has been detected: inform the user and fail. */
      append(Body,Goal,CurrentGoal),
      nl, write('prune'), nl, write([CurrentGoal|ListGoal]),
      !, fail.
```

The procedure `check` that is called in the second clause has exactly the same functionality as it has in the meta-interpreter, but due to a different representation of variables its clauses differ slightly. As there is only one level here, it is not possible to represent object-level variables by meta-level constants. But we must still avoid that a variable that is bound by solving the current goal is also bound in the list of previous goals (and results). To this end each goal (result) that is added to such a list is renamed first, using fresh variables (of course the same renaming is applied to a goal G_k and to the corresponding result $G_0\theta_1 \ldots \theta_k$). The following clause for `check` is a typical example (the second and third parameter of `rename_var` are used to hold intermediate substitutions).

```
check(evr,Goal,Result,ListGoal,ListResult,_,
          [NewGoal|ListGoal],[NewResult|ListResult]) :-
      rename_var([Result|Goal],[],_,[NewResult|NewGoal]),
      check_EVR(NewGoal,NewResult,ListGoal,ListResult).
```

The literals specified by the user must pass the parameters needed for the loop check (`Check`, `Goal`, `Result`, `ListGoal`, `ListResult` and `Depth`). Except for `Check`, which remains unchanged, each literal must pass an input value (when it is selected) as well as an output value (when it is resolved) for these parameters. So a pair of each is added, except for `Result`: the application of the subsequent unifiers automatically changes the input value for `Result` into the output value for `Result`.

Thus a clause $h(\underline{X}) :- b_1(\underline{Y}_1), \ldots, b_n(\underline{Y}_n)$ is transformed to:

```
h(X,Check,Goal,Result,ListGoal,ListResult,Depth,
          Goal_{n+1},ListGoal_{n+1},ListResult_{n+1},Depth_{n+1}) :-
      loop_check(Check,[b1(Y1),...,bn(Yn)],Goal,Result,ListGoal,
          ListResult,Depth,Goal1,ListGoal1,ListResult1,Depth1)
          ,
      b1(Y1,Check,Goal1,Result,ListGoal1,ListResult1,Depth1,
          Goal2,ListGoal2,ListResult2,Depth2),
      ...,
      bn(Yn,Check,Goaln,Result,ListGoaln,ListResultn,Depthn,
          Goal_{n+1},ListGoal_{n+1},ListResult_{n+1},Depth_{n+1}).
```

Moreover, if `h` is the predicate that the user will use in the query, a clause is added to properly instantiate the input parameters:

```
h(X,Check) :-
  rename_var([h(X)],[],_,RenamedGoal),
  h(X,Check,[h(X)],h(X),[RenamedGoal],[RenamedHead],0,_,_,_,_).
```

Notice that the instantiation is only correct under the assumption that the query consists *solely* of the atom `h(X)`. In the meta-interpreter, `Result` was always a list of atoms, containing as many atoms as the initial goal. Now that the initial goal is known to consist of a single atom, the list construction is unnecessary.

Results

In its most unspecific form, the question we try to answer in this section is: 'How costly is loop checking?'. The answer depends not only on the loop check that is used and its implementation, but also on the object program.

Often there is a trade-off between using a weak loop check, which detects loops late (or not at all), and using a stronger loop check, which is usually costlier. The object program, especially the amount and the length of its loops, determines which loop check gives the best results. On one side, for programs without loops the empty loop check is certainly optimal. On the other hand, when a loop check fails to detect a loop, the resulting computation is infinite, a result that is difficult to compare to any finite figure. When a loop check is compared with the empty loop check, one of the two extremes is bound to occur.

Existing benchmarks are not very useful in this case. First of all, our meta-interpreter is only capable of handling pure logic programs, which benchmarks rarely are. Secondly, benchmarks usually terminate without loop checking: the effort needed to enforce termination is already encoded explicitly in the program.

Our choice is not to measure the benefit of using a stronger loop check, but only its cost. That is, if two loop checks are compared, the object program and goal are chosen such that the resulting SLD-tree is pruned by the two loop checks at the same place(s). In particular, when a loop check is compared to the empty loop check, the object program does not loop.

In practice we used the standard transitive closure program:

```
tc(X,Y) :- r(X,Y).
tc(X,Z) :- r(X,Y),tc(Y,Z).
```

augmented with facts giving the relation `r`. This program allows us to control the presence and length of refutations and loops easily by modifying the graph of `r`. Furthermore the program is restricted and function-free, so all equality and subsumption checks (including their single and double selected variants) enforce termination of this program for any relation `r`. Finally it is easy to see that only the selection, but not the underlying full-comparison check determines where derivations from such programs and goals are pruned.

The numerical results mentioned below were obtained using a linear relation {`r(a,b)`,...,`r(m,n)`} and a circular relation {`r(a,b)`,...,`r(m,n)`,`r(n,a)`}. The goals used were `?-tc(a,n)` and `?-tc(a,n),fail,fail,fail`. All loop checks are applied using the linear relation; obviously the resulting derivations are never pruned. All loop checks are also applied using the circular relation, although the computations using `non` and `empty` diverge where the others are pruned. The length of the loop in `r` is chosen to be 14, which means that a loop occurs after 28 derivation steps. As 28 is a triangular number, the triangular loop checks detect the loop immediately. So all loop checks prune the resulting derivations at the same place.

The question 'How costly is loop checking?' suggests a comparison between an efficient PROLOG interpreter with loop checking and existing PROLOG interpreters. As developing a really efficient PROLOG interpreter with loop checking involves much more work than making the two relatively simple implementations presented in this paper, it would be helpful if our implementations could yield some predictions about more efficient ones.

It appears that the most time- and space-consuming component of our implementations is the explicit manipulation of substitutions, which occurs both in the construction of the derivation (in the form of unification) and in the loop check (in the form of matching). Consequently, predictions about efficient implementations (comparing those with and without loop checking) can only be made under the assumption that they also spend most of their time manipulating substitutions.

linear relation r		non	empty	eig	eir	evg	evr
solving	check	0.197	0.247	1.285	1.390	1.345	1.397
←tc(a,n)	derive	7.593	7.603	7.965	8.046	7.794	7.990
finishing	check	0.222	0.282	1.546	1.672	1.620	1.700
←tc(a,n)	derive	8.894	8.907	9.399	9.473	9.384	9.405
failing	check	0.412	0.425	1.716	1.847	1.785	1.857
←tc(a,n),fail,fail,fail		9.331	9.308	9.730	9.764	9.764	9.819

linear relation r		sig	sir	svg	svr	(sir,st)	(sir,dt)
solving	check	1.754	1.790	1.755	1.798	0.504	0.409
←tc(a,n)	derive	8.003	8.009	7.960	7.987	7.726	7.663
finishing	check	2.164	2.217	2.152	2.233	0.540	0.439
←tc(a,n)	derive	9.449	9.442	9.386	9.416	9.019	8.967
failing	check	4.193	4.316	4.209	4.306	0.693	0.563
←tc(a,n),fail,fail,fail		10.560	10.513	10.481	10.559	9.332	9.308

TABLE 7.2.1

circular relation r		non	empty	eig	eir	evg	evr
solving	check	0.210	0.271	1.279	1.408	1.303	1.409
←tc(a,n)	derive	7.638	7.607	8.295	8.259	8.324	8.325
finishing	check	diverges		1.612	1.770	1.640	1.782
←tc(a,n)	derive	diverges		9.920	9.899	9.960	9.967
failing	check	diverges		1.784	1.958	1.864	1.972
←tc(a,n),fail,fail,fail		diverges		10.136	10.238	10.246	10.261

circular relation r		sig	sir	svg	svr	(sir,st)	(sir,dt)
solving	check	1.775	1.859	1.758	1.852	0.524	0.412
←tc(a,n)	derive	8.363	8.412	8.391	8.378	8.143	8.058
finishing	check	2.264	2.374	2.262	2.377	1.207	0.642
←tc(a,n)	derive	10.041	10.098	10.045	10.037	9.656	9.570
failing	check	4.318	4.466	4.366	4.491	1.705	0.889
←tc(a,n),fail,fail,fail		10.863	10.930	10.932	11.004	9.831	9.760

TABLE 7.2.2

In the meta-interpreter, both unification and matching are performed explicitly, whereas in the transformed programs only matching is performed explicitly: unification is performed by the PROLOG-interpreter itself. In a practical PROLOG-interpreter with loop checking, both unification and matching will be done in an efficient way. Predictions about such practical interpreters must thus be based on measurements of the meta-interpreter, because it treats unification and matching on an equal level.

Tables 7.2.1 and 7.2.2 show such measurements (in seconds cpu-time used). In Table 7.2.1 the linear relation `r` is used. The first pair of rows shows how much time is needed to find the first (and only) solution for `?-tc(a,n)`, the second pair shows the time needed until the computation of `?-tc(a,n)` halts and the third the time needed until the computation of `?-tc(a,n),fail,fail,fail` fails. Table 7.2.2 shows the same for the circular relation `r`.

It is noteworthy that the *measured* time spent on generating the derivation is related to the loop check. This implies that a fraction of the time measured as 'der ve'-time is actually 'check'-time. A numerical analysis shows that about 30% of the 'check'-time is involved. Making this correction we see in this example that the equality checks cost about 24% of the 'derive'-time and the subsumption checks about 32% (for the goal `?-tc(a,n)`), respectively 60% (for `?-tc(a,n),fail,fail,fail`).

The results show that there is not much difference among the various equality checks (and among the various subsumption checks). Not surprisingly, the checks based on goals are slightly cheaper than those based on resultants. In contrast to the rule-of-thumb that stronger checks are costlier, the checks testing for instances are slightly cheaper than the checks testing for variants. This is due to the implementation: first a substitution is computed, then it is tested if this substitution is a renaming.

The difference between the equality checks on the one hand and subsumption checks on the other hand is more significant. Due to the small goals in the refutations of `?-tc(a,n)` (only one or two atoms), the subsumption checks are approximately only 34% more expensive. But for the larger goals in the derivations of `?-tc(a,n),fail,fail,fail`, this figure is already 134%!

The advantage of the triangular loop checks is evident: they need much less time. But in this test their disadvantage does not show: here they have the same effect as the full-comparison loop checks, but in general they prune derivations

(much) later. In small examples like these, where the cost of loop checking is rather small compared to the cost of generating derivation steps, full-comparison loop checks will usually show a better performance.

But one may suspect that it costs too much to apply the full-comparison loop checks on a large example with few loops. In any case the use of a triangular loop check definitely beats using no loop check at all. This applies in particular to programs that are still being tested/debugged: they are not supposed to loop, but some loops may be present. A triangular loop check that also reports pruned derivations would probably be a useful debugging tool.

We can only briefly address the issue of optimisations here. Our meta-interpreter does not use any of the common optimisation techniques, such as last call optimisation. It is conceivable that such optimisations are seriously hampered by the introduction of loop checking, making loop checking more costly than our figures show.

On the other hand, our loop checking procedure itself could certainly be improved considerably, for example by using 'incremental' testing: an equality check tests first if two goals have the same length, then whether they have the same predicates in the same order and so on. Also the storage and retrieval of previous goals could be improved by hashing techniques. These optimisations would make loop checking less costly. At this time we cannot predict how those two effects accumulate.

8. Related Work

Termination of logic programs can be enforced in many ways, of which our form of loop checking is just one. In this chapter we outline a spectrum of methods for enforcing termination and we discuss the place of loop checking therein. The place of a method within this spectrum shall be determined by the amount of change it involves for the interpreter, compared with the standard pure logic programming interpreter based on SLD-resolution.

Proving termination of logic programs

The first technique we encounter consists of identifying those logic programs that terminate without any change of the interpreter. By rephrasing it as 'identifying those logic programs for which the empty loop check is complete', this technique could be considered as a special case of the issues treated previously. However, we feel that the study of terminating logic programs is a research area of its own right. Without aiming at completeness, we mention here the following work.

Vasak and Potter [VP] distinguish *existential* and *universal* termination. Given a goal, a program is said to terminate existentially if it either fails finitely or produces at least one solution. This depends on both the selection rule and the search rule. A program P is said to terminate universally on a goal G if the SLD-tree for $P \cup \{G\}$ is finite. This still depends on the selection rule used, but not on the search rule. Usually, termination refers to *universal* termination.

The dependency on the selection rule yields another distinction: $P \cup \{G\}$ is *strongly terminating* if *every* SLD-tree for $P \cup \{G\}$ is finite; $P \cup \{G\}$ is *weakly terminating* if there *exists* a finite SLD-tree for $P \cup \{G\}$. Strong termination is for example studied in [Be] and [AB]. Between these two extremes, a view of practical interest is taken by Apt and Pedreschi [AP]: $P \cup \{G\}$ is *left terminating* if the SLD-tree for $P \cup \{G\}$ via the *leftmost* selection rule is finite.

Starting from Floyd [Fl] the classical proofs of program termination have been based on the use of well-founded orderings. Every 'situation' S in a computation is associated to an element f(S) of a well-founded set and for every 'step' leading from a situation S_1 to a situation S_2 it is shown that $f(S_2) < f(S_1)$.

In the area of logic programming, this approach is outlined in [Be]. There, a *level mapping* for a program P is defined as a function from the ground atoms in L_P to the natural numbers. A program is called *recurrent* if for some level mapping $|\ |$, for every *ground instance* $A \leftarrow B_1,\ldots,B_n$ of a clause in P:

$$|A| > |B_i| \text{ for } i = 1,\ldots,n.$$

An atom A is bounded w.r.t. a level mapping $|\ |$ if $|\ |$ takes a maximum on the set of ground instances [A] of A; if A is bounded then |[A]| denotes this maximum. A goal $G = \leftarrow A_1,\ldots,A_n$ is bounded if $A_1,\ldots,A_n$ are bounded; if G is bounded then |G| denotes the multiset $\{|[A_1]|,\ldots,|[A_n]|\}$. Notice that ground atoms, and hence ground goals, are always bounded.

A 'situation' in logic programming corresponds to a goal, a 'step' corresponds to a derivation step. A level mapping associates a multiset of natural numbers to a bounded goal. Now let P be a recurrent program and G a bounded goal w.r.t. some level mapping $|\ |$. Bezem [Be] proves: if $G \Rightarrow G'$ is a derivation step w.r.t. P, then G' is again a bounded goal and $|G'| < |G|$ in the multiset ordering (which is well-founded). Thus recurrent programs are strongly terminating for bounded goals, and especially for ground goals. A perhaps more surprising result of [Be] is that the converse of this theorem is also true: if a program P is strongly terminating for every ground goal, then P is recurrent.

In [AB] this approach is generalized to the case of general programs and SLDNF-resolution[1]. The corresponding programs are called *acyclic* programs; every acyclic program is locally stratified. As the distinction between SLDNF-resolution and SLS-resolution lies in their treatment of infinite failure, it is not surprising that the two coincide for acyclic programs.

Apt and Pedreschi [AP] have modified these ideas to characterize left-termination. Given a program P, a level mapping $|\ |$ and a model I of P, P is *acceptable* w.r.t. $|\ |$ and I if for every ground instance $A \leftarrow B_1,\ldots,B_n$ of a clause in P:

$$|A| > |B_i| \text{ for } i = 1,\ldots,\underline{n}, \text{ where } \underline{n} = \min(\{n\} \cup \{i \mid I \not\models B_i\}).$$

The idea is that $I \not\models B_i$ implies that selecting B_i results in (finite) failure. Thus due to the use of the leftmost selection rule $B_{i+1},\ldots,B_n$ are never selected, hence their levels are irrelevant. It is proved in [AP] that

---

[1] The treatment of floundering in original definitions of SLDNF-resolution (e.g. in [L]) is not very satisfactory: a 'floundering SLDNF-derivation' does not exist. In [AB] floundering is treated in SLDNF-resolution like in SLS-resolution (see Definition 5.2.4). See also [AD].

for every ground goal G: P∪{G} is left-terminating iff

for some level mapping | | and model I of P: P is acceptable w.r.t. | | and I.

Of course the question remains how a suitable level mapping and model are found, and how the acceptability condition is tested on infinitely many ground instances of clauses. Abstract interpretation techniques can help here.

Also related work by Bossi et al. [BCF] and by Wang & Shyamasundar [WS] could help at solving this problem. They transform a program P and a goal G into a directed graph (*U-graph* in [WS], *specific graph* in [BCF]). The nodes of these graphs are the atoms that occur in P and G; arcs represent dependency (via the clauses of P) and unifiability. Every cycle in the graph corresponds to a *possible* loop in the program. The problem of finding a suitable mapping can now be solved locally for every strongly connected component (SCC) of the graph: the mapping must not increase on the arcs of the SCC, and every loop in the SCC must contain an arc on which the mapping strictly decreases.

The method described in [BCF] is again related to a slightly different approach, introduced by Ullman & Van Gelder [UvG] and improved by Plümer [P]. First, they assume that the analyzed program is well-moded (as originally defined by [DM]), i.e., that the input and output positions of a predicate can be distinguished. Modes can often be inferred automatically, e.g. by abstract interpretation, and need not be supplied by the programmer. The size (level) of an atom A is now determined by sum of the sizes of the terms on (a selection of) the input positions of rel(A) (the size of a term can for example be defined as in Section 6.3).

Now their way to prove (left-)termination is the following. Consider an atom $p(t_1,\ldots t_n)$ with ground terms on its input positions. Suppose that constructing the SLD-tree of $P\cup\{\leftarrow p(t_1,\ldots t_n)\}$ via the leftmost selection rule leads recursively to the selection of another p-atom, say $p(s_1,\ldots,s_n)$. The termination of the program is ensured if in all such cases the size of $p(s_1,\ldots,s_n)$ is strictly smaller than the size of $p(t_1,\ldots,t_n)$.

In order to prove such a claim so-called *linear predicate inequalities* are inferred. In [UvG] these linear predicate inequalities have the form $p_1 + c \geq p_2$. In [P] this is generalized to $\sum_{i\in I} p_i + c \geq \sum_{j\in J} p_j$, where I is a selection of the input positions of the predicate p and J is a selection of its output positions. A linear predicate inequality is *valid* if for every ground atom $p(t_1,\ldots,t_n)$ in the least Herbrand model of P: $\sum_{i\in I} \text{size}(t_i) + c \geq \sum_{j\in J} \text{size}(t_j)$ holds.

With the aid of these linear predicate inequalities an upper bound for the sizes of the input terms s_i ($i \in I$) of a recursive call $p(s_1,\ldots,s_n)$ can be given in terms of the sizes of the input terms t_i ($i \in I$) of the 'parent' call. For example, consider the program P = {p(f(f((x)),x)←. q(0)←. q(x) ← p(x,y), q(y).} and take for $n \geq 0$: $size(f^n(0)) = n$. A possible mode for this program is p(+,–), q(+), i.e., the second position of p is an output position, the other positions are input positions. It is obvious that $p_1 - 2 \geq p_2$ is a valid linear predicate inequality for p in P. Thus evaluation of ←q(t) for some ground term t leads to a recursive call ←q(s) with size(t) – 2 ≥ size(s), hence size(s) < size(t).

Plümer describes how the analysis of the data flow within clauses enables the automatic derivation of valid linear predicate inequalities (thereby generalizing similar considerations restricted to terms being lists in [UvG]). The result is an efficient tool for testing left termination for a large class of logic programs.

Finally, M. Baudinet [Ba] presented a method for proving termination of PROLOG programs in which with each program a system of equations is associated whose least fixpoint is the meaning of the program. By analyzing this least fixpoint various termination properties can be proved. The main method of reasoning is fixpoint or structural induction.

Detecting loops in derivations

Loop detection techniques can be divided into two classes: those techniques that warn for *all possible* infinite loops, and those that signal only the *certainty* of a infinite loop. Due to the undecidability of the Halting Problem, the first class must give some false alarms, whereas the second class must miss some loops. For loop checks we introduced the terminology *complete* for the first class and (weakly) *sound* for the second class. A termination prover can be used as a loop detection technique of the first class: if the attempt to prove termination fails, a warning can be issued that the program *might* loop.

The aim of loop detection is controlling recursive inference. This can be implemented in two ways. The first way is to perform a static analysis of the program as a pre-processing step. The outcomes of such an analysis can be translated into modifications of the program, such as reordering the atoms in a clause or the addition of control primitives (e.g. wait declarations as described in [N]). The other method is the application of a loop detection mechanism at run-

time, where its results can influence control directly. The framework of loop checking we described falls in this category. Although run-time analysis is generally able to cope with larger classes of programs and loops therein, it has the disadvantage of introducing overhead at run-time.

As the two distinctions are more or less orthogonal, we can divide loop detection mechanisms in four categories. However, the boundaries between those categories are not always very sharp. For example, Schmücker [Sc] proposes two conditions: one for detecting potentially infinite loops and one for detecting definitely infinite loops; then she shows how the two conditions can be combined. Moreover, she explicitly mentions both run-time analysis and preprocessing as possible applications. Another example is that [SGG], a paper that is oriented towards run-time analysis, inspired the preprocessing-oriented work in [dSBV]. Finally, loop checks that are neither sound nor complete can be found in a number of papers.

Here we shall concentrate on the detection of loops in derivations at run-time. It seems that Gelernter [G] was the first one to apply this kind of loop checking (in a geometry theorem prover). Loveland and Reddy [LR] isolated his technique and elaborated on it. The problem was faced in the propositional (or more precisely: variable-free) case. In this case the solution is straightforward: a derivation is pruned as soon as an atom reproduces itself. In the absence of variables this check coincides with the Variant of Atom check (Definition 2.1.5) and the context checks (Section 3.4). Therefore it is sound and (for finite programs) complete.

Loveland and Reddy claim that 'the results presented here ... hold equally well in the first-order format ... since it is a propositional rule', but they do not further comment on what they mean by this. Poole and Goebel [PG] show that checking for syntactically identical atoms is not complete in the presence of variables, and that generalizations such as VA and IA are not weakly sound. On this basis they argue that it is better to forget about loop checking. Van Gelder [vG1] proposes an implementation of loop checks based on IA and on checking for identical atoms (see also Section 7.1: the tortoise-and-hare technique).

Brough and Walker [BW] describe two interpreters for logic programming that implement loop checking, and compare them with the standard interpreter (all using the leftmost selection rule). Their first interpreter, I_G, checks for identical atoms again. Their second interpreter, I_R, is more interesting, because

it uses one of the few loop checking mechanisms that are not based on comparing goals: it is based on comparing the clauses used. A derivation D = ($G_0 \Rightarrow_{C_1,\theta_1} G_1 \Rightarrow \ldots \Rightarrow G_{k-1} \Rightarrow_{C_k,\theta_k} G_k$) is pruned at G_k if for some i: $C_k\theta_k \leq C_i\theta_i$ and the selected atom in G_k is produced by the selected atom in G_i. It seems that this method satisfies Definition 2.1.2: it is a simple loop check. In general it is not a complete loop check, but it is complete for function-free programs.

Besnard [B] rephrases the loop checking criterion as 'allowing a clause to be used at most once'. Apart from the omission of the phrase 'on the same atom' (to capture that the selected atom in G_k is produced by the selected atom in G_i), this ignores the fact that actually the instances of the clauses used are compared. Still, his example, which shows that the loop check is not weakly sound, is correct. Brough and Walker themselves already show that the loop check is not sound (Example 3 of [BW]). We present here a small extension of this example, which also shows that it is not even weakly sound.

EXAMPLE 8.1.

Let P =

{ p(a,b,c) ←. (C1),
p(y,z,x) ← p(x,y,z). (C2),
q(c) ←. (C3) }

and let G = ←p(u,v,w),q(u).

Figure 8.1 shows the only SLD-refutation of P∪{G} (modulo variants). Obviously the instances of (C2) used in the first and second step are variants, so this refutation is not found by I_R. □

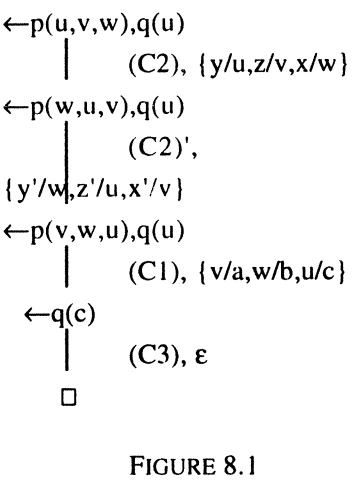


FIGURE 8.1

In spite of this, Brough and Walker claim that I_G and I_R are 'better' than the standard interpreter. They support this claim as follows. According to their definitions an interpreter does not enumerate answers as it finds them, but it collects them and produces them at once when it terminates. Consequently, if it does not terminate, it gives no answers at all. Furthermore, the more (correct) answers an interpreter gives, the better it is (for a given program and goal). That

nontermination can be preferable over the result 'no (more) answers' if the latter is not sound w.r.t. CWA is not taken into account. In this context their claim is justified, because they prove: 'given a program and a goal, if the standard interpreter stops, then I_G and I_R do not prune any derivations'. But this way of comparing interpreters is suspicious at least.

Covington [Co1] also proposes some kind of Variant of Atom check, or perhaps EVG, but his definitions are not very precise. He proposes to 'block the evaluation of subgoals that match higher goals' (in such a context 'goal' often means 'selected literal'), where 'in determining whether the arguments are the same, uninstantiated variables are considered to match each other even though they have different names'. Taking this literally, p(x,x) and p(y,z) match each other. Obviously the resulting method is unsound, but at least it can be defined as a simple loop check, because the matching criterion is closed under variants.

Covington also suggests to limit the number of comparisons. He suggests some special solutions for transitive and symmetrical relations. For other cases he suggests the use of a single selected loop check (Definition 7.1.6), with a selection of the form $\{b, b^2, b^3, \ldots\}$. He shows that this gives approximately $b/(b-1)$ comparisons per goal. He also suggests the use of hashing techniques.

In a second paper ([Co2]), Covington shows that his loop check is unsound, and that hence the resulting interpreter incomplete. He proposes to weaken the loop check by adding an extra requirement: the same program clause must be applied on the 'similar' atoms. This revision closely resembles I_R of [BW]. Indeed, although Covington says that this 'appears to solve the problem completely', Example 8.1 refutes this claim.

Finally we mention here the work of Besnard [B]. After giving some counterexamples regarding [BW] and [Co1], he proceeds by describing a weakly sound loop check (CIG). In Section 3.4 we elaborated on this work by defining some variations on this check (CVG, CIR, CVR); CIR and CVR are sound. These checks still test for similar atoms, but now the context of these atoms is taken into account. That is why we named them 'context checks'.

Tabulation

The fact that VA is not weakly sound had been observed already by Black ([Bl Section 6.4]. He suggests some improvements, such as taking into account which clause produced an atom, or allowing a fixed number of repetitions before

stopping a derivation. But because the resulting methods are still not weakly sound, he proposes a more subtle solution than simply pruning infinite derivations. His solution was re-invented several times and is known under different names, such as memoing or memo-ization ([D]), tabulation ([TS]) and lemma resolution ([V]). It can also be found in [SGG Algorithms 3.7 and 3.8].

These techniques are essentially the same, although the proposed implementations can differ in details. The main idea, which originates from functional programming, is to store intermediate results, and to look them up when they would normally be recomputed. Apart from the effect on termination, this can improve the efficiency of a program considerably. In the following, more precise description of this method, we shall adhere to the terminology of [V]. He calls his formalization of the method SLD-AL resolution: SLD-resolution augmented with an *Admissibility test* and *Lemma resolution*.

When an atom in a goal is selected, the admissibility test distinguishes two possibilities. The first possibility is that the atom is 'new': no results have been computed for it yet. In this case the goal is called *admissible* and the selected atom is resolved by SLD-resolution against a clause of the program, as usual. The other possibility is that the selected atom is an instance of an atom that was selected earlier. For this earlier atom (the *producer*) answers (*lemmas*) have been searched. In this case the goal is not admissible, and the selected atom is only resolved against those lemmas (*lemma resolution*).

It is necessary to use a local selection rule: once an atom is selected all answers for that *atom* are requested; restrictions imposed by another atom on the answers for the full *goal* are found when (an instance of) this other atom is selected. The part of a derivation between the selection of an atom and the point where the atom is completely resolved can be considered a 'local proof' for (an instance of) that atom. Its result is added as a lemma. In [TS] only the leftmost selection rule is considered, whereas [V] allows any local selection rule.

When an atom A is identified as being a producer for another atom B, two cases arise. If all possibilities for A are already exhausted, i.e., all answers for A are known, then these answers can simply be applied on B. The other possibility is that A is still being processed. This is certainly the case if B descends from A. In this case the answers for A that are already known can be applied on B. If A produces more answers later, then these answers must also be applied on B. If B descends from A, this can lead to still more answers for A and the process

repeats. As is shown in the following example, it is even possible that there are infinitely many answers for A; in this case B has infinitely many descendants.

EXAMPLE 8.2.

Let P = { p(0)←.
p(f(x) ← p(x). }
and let G = ←p(x).

In Figure 8.2 p(x') is found to be an instance of p(x), so ←p(x') is not admissible and p(x) becomes a producer. At that point only the answer {x/0} has been found for p(x). When this answer is used to resolve p(x'), a new answer for p(x), namely {x/f(0)}, is found. Using this on p(x') yields {x/f(f(0))} for p(x), et cetera □

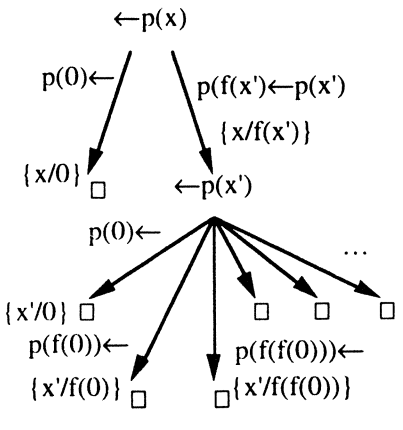


FIGURE 8.2

The order in which the nodes of an SLD-AL tree are constructed (visited) is described by the search rule. As for ordinary SLD-resolution, a 'good' search rule is required for finding all answers. An important advantage of SLD-AL resolution over SLD-resolution is that for function-free programs SLD-AL trees are always finite (hence any search rule is 'good'). But in the presence of function symbols, SLD-AL trees can be infinitely branching, in addition to the possibility of having infinite branches.

A further notable difference between SLD-trees and SLD-AL trees is that the shape of the SLD-AL tree depends on the search rule that is applied while constructing it. Until now we spoke about 'earlier' selected atoms, relying on the intuition of the reader. Whether one goal is 'earlier' than another depends on the search rule. Moreover, a inadmissible goal can be visited several times if its producer is still producing new answers (e.g. the goal ←p(x') in Figure 8.2 is visited infinitely many times; each visit results in a new solution for its producer and hence in a new visit).

In Definition 4.1 of [TS] and Appendix D of [V], a search strategy for SLD-AL trees is described: the multistage depth-first strategy. This strategy involves the creation of a *sequence* of *straight SLD-AL trees*. These trees differ from

SLD-AL trees in that the lemmas used for lemma resolution are not proved in the tree itself, but given in advance. At each stage of the search, a straight SLD-AL tree is constructed depth-first, using the lemmas proved in the straight SLD-AL trees constructed in the previous stages. Consequently, straight SLD-AL trees are again finitely branching. In particular, for the tree constructed in the first stage, no lemmas are available, thus inadmissible goals are just pruned.

It is obvious that every straight SLD-AL tree in a sequence thus constructed contains the preceding trees. This gives rise to two implementations: either each tree is constructed anew, or each tree is obtained by extending its predecessor. The first implementation (called QSQR in [V]) can take maximal advantage of depth-first techniques, but involves duplicate work. The second one (used in QoSaQ) requires more storage.

In order to achieve completeness, it is important that none of the straight SLD-AL trees in the constructed sequence is infinite. In the presence of function symbols this is not necessarily the case. (Consider the program {p(x)←p(f(x)).} and the goal ←p(0).) [TS] propose the use of *term-depth abstraction* (later called *subgoal generalization* in [V]) to obtain finite trees. In this technique the admissibility test is not applied on the atoms that actually occur, but on generalized versions of them: whenever a subterm of a term occurs below a certain depth, it is replaced by a fresh variable. The result is the same as if the bounded term-size property were met. SLD-ALG resolution (SLD-AL resolution incorporating this technique) is sound and complete.

In order to limit the number of inadmissible goals, some other suggestions for altered admissibility tests have been made, namely:

- applying the test only for designated predicates (called r-predicates in [V]; it is suggested that at least all recursively defined predicates should be r-predicates),
- making a goal inadmissible only if the selected atom is an instance of one of its *ancestors*.
- testing for *variants* instead of *instances*.

Kemp and Topor [KT] and Seki and Itoh [SI] generalized the tabulation technique to stratified programs (locally stratified programs are not considered, and the issue of floundering is avoided). [SI] is based on [TS], whereas [KT] is based on [V]. Both use negation as failure (i.e., SLS-resolution). Recently, Chen and Warren [CW] implemented an interpreter for their XOLDTNF-

resolution, which uses tabulated SLS-resolution to compute the well founded model [vGRS] of arbitrary programs with negation.

Finally a combination of pruning and SLD-AL resolution is proposed in [V]. The pruning mechanism he proposes is closely related to the SIR_M check. The only significant difference is that the SIR_M check compares a goal only with its ancestors, whereas in [V] a goal can be compared with every earlier constructed goal. The following definition formalizes his approach, and corrects some minor errors in it.

DEFINITION 8.3 (Redundancy).
Let P be a program, G_0 a goal and T an SLD-tree of $P \cup \{G_0\}$ via a local selection rule. Let G be a goal in T. A *relevant ancestor* of G in T is an ancestor H of G such that the selected atom A in H is not yet resolved in the *parent* of G (in other words: the path from H to G is fully contained in the local proof of A).

A goal G_1 is *redundant relatively to* its relevant ancestor H if there is a goal G_2 (other than G_1) such that H is also a relevant ancestor of G_2 and for some substitution σ the following holds.

Let A be the selected atom in H and let R be the rest of H. Let θ_1 be the composition of mgu's on the path from H to G_1, and θ_2 the composition of mgu's on the path from H to G_2. Then we can write $G_1 = \leftarrow(S_1, R\theta_1)$ and $G_2 = \leftarrow(S_2, R\theta_2)$. It must hold that $S_2\sigma \subseteq_M S_1$ and $A\theta_2\sigma = A\theta_1$.

A goal is *locally redundant* if it is redundant relatively to *all* its relevant ancestors, a goal is *globally redundant* if it is redundant to at least *one* of its relevant ancestors. □

It is easy to prove that in the given context $A\theta_2\sigma = A\theta_1$ implies $R\theta_2\sigma = R\theta_1$, and hence $S_2\sigma \subseteq_M S_1$ implies $G_2\sigma \subseteq_M G_1$. Furthermore, if a goal G is redundant relatively to H, then G is redundant relatively to all ancestors of H that are relevant ancestors of G. Thus in order to determine that G is *locally* redundant (in a partially constructed SLD-tree), it is sufficient to compare G to all previously constructed descendants of the *parent of G* (which is by definition a relevant ancestor of G). In order to determine that G is *globally* redundant, it is sufficient to compare G to all descendants of the *highest relative ancestor of G* in the SLD-tree constructed so far.

In ordinary SLD-trees, all globally redundant goals can be pruned without losing computed answers. The proof of this claim is similar to the proof of Theorem 3.3.7 (the soundness of SIR_M). See also Appendix A.3 in [V]. It is based on the observation that if a goal G that is redundant relatively to H is pruned, all solutions for H are still found, even though some solutions for some descendants of H may not be found.

For SLD-AL trees, the situation gets more complicated. In these trees, inadmissible goals occur that obtain the solutions for their selected atoms from the associated producers. This works properly only if the producer produces *all* solutions. This cannot be guaranteed if globally redundant descendants of the producer can be pruned: they can be pruned only if the are redundant *relatively to the producer.*

A simple solution is to prune only locally redundant goals, as they are redundant relatively to all their ancestors. A more complex solution is to prune in addition those globally redundant goals that are not needed by a producer. I.e., a goal is pruned if it is globally redundant and also redundant relatively to all relevant ancestors of it that are producers. This seems still quite simple, until one realizes that a goal is not created as a producer: it can become a producer when an other goal is created. Thus this solution requires a *waking* mechanism: a goal G is put asleep when it is globally redundant and not needed by a producer; when one of its relevant ancestors H becomes a producer and G is not redundant relatively to H, then G is woken up. The practical problem of implementing such a waking mechanism is addressed in the QoSaQ procedure (see [V]).

Bottom-up query processing

Although it was described as a top-down strategy, tabulation is closely related to bottom-up strategies. The widely recognized disadvantage of naive or semi-naive (bottom-up) evaluation is that it fails to focus on relevant data. Tabulation can be seen as top-down determination of relevant data, alternated with bottom-up computation of answers. The same effect can be obtained by a program transformation called Magic Sets. For studies relating tabulation to semi-naive bottom-up evaluation using magic sets, we refer to [Br], [Se] and [BD]. A number of other bottom-up strategies is described and compared in [BR].

Conclusion

One could get the impression that tabulation and sophisticated bottom-up evaluation strategies are by far superior over PROLOG-like interpreters (with or without loop checking). For certain applications, especially in the area of deductive databases, this is definitely the case. Although most of our completeness results are restricted to function-free programs, loop checking can be applied to all programs, because most soundness results are valid for all programs. A loop check does not need to detect *all* loops in order to be useful. We feel that as long as programmers alter logic programs to enforce termination, a general and easy-to-understand pruning mechanism is a welcome modification of interpreters, especially if it can be implemented at a reasonably low cost. Programmers who insist that all loops are bugs could use loop checking as a debugging tool.

The fact that the topic of implementing simple methods for loop detection is addressed frequently seems to prove these points. Unfortunately, the topic is often studied in an ad hoc manner. When the first idea, invariably closely related to the Instance of Atom check, appears to be wrong, the disappointed researcher usually abandons the issue. As a result no framework for the classification of simple loop checks had been set up until now. Moreover, a lot of similar and duplicate work has been done. Hopefully this book can serve as a theoretical foundation for further, more practically oriented research in this area.

References

[A] K.R. APT, *Logic Programming*, in: Handbook of Theoretical Computer Science (J. van Leeuwen ed.), vol. B, North Holland, 1990, 493-574.

[AB] K.R. APT and M. BEZEM, *Acyclic Programs*, New Generation Computing 29(3), 1991, 335–363.

[ABo] K.R. APT and R.N. BOL, *Logic Programming and Negation: A Survey*, J. Logic Programming, 1994.

[ABW] K.R. APT, H. BLAIR and A. WALKER, *Towards a Theory of Declarative Knowledge*, in [M], 89–148.

[AD] K.R. APT and H.C. DOETS, A new definition of SLDNF-resolution. ILLC Prepublication Series CT-92-03, Dept. of Math. and Comp. Sci., Univ. of Amsterdam, 1992. To appear in J. Logic Programming.

[AvE] K.R. APT and M.H. VAN EMDEN, *Contributions to the Theory of Logic Programming*, J. ACM 29(3), 1982, 841–862.

[AP] K.R. APT and D. PEDRESCHI, *Reasoning about Termination of Pure Prolog Programs*, Information and Computation 106(1), 1993, 109–157.

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

[Ba] M. BAUDINET, *Proving Termination Properties of PROLOG Programs: a Semantic Approach*, in: Proc. of the Third Annual IEEE Symposium on Logic in Computer Science (LICS), Edinburgh, 1988, 336–347.

[BCF] A. BOSSI, N. COCCO and M. FABRIS, *Proving Termination of Logic Programs by Exploiting Term Properties*, in: Proc. TAPSOFT'91, 1991, 153–180.

[BD] R.N. BOL and L. DEGERSTEDT, *The Underlying Search for Magic Templates and Tabulation*, in: Proc. of the Tenth Int. Conf. on Logic Programming (D.S. Warren ed.), MIT Press, Cambridge Massachusetts, 1993, 793–811.

[Be] M. BEZEM, *Characterizing Termination of Logic Programs with Level Mappings*, in: Proc. of the 1989 North American Conf. on Logic Programming (E.L. Lusk and R. Overbeek eds.), MIT Press, Cambridge Massachusetts, 1989, 69–80.

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

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

[Bl] F. BLACK, *A Deductive Question Answering Machine*, in: Semantic Information Processing (M. Minsky ed.), MIT Press, Cambridge Massachusetts, 1968, 354–402.

[BR] F. BANCILHON and R. RAMAKRISHNAN, *An Amateur's Introduction to Recursive Query Processing Strategies*, in: Proc. ACM-SIGMOD Int. Conf. on Management of Data, 1986, 16–52.

[Br] F. BRY, *Query Evaluation in Recursive Databases: Bottom-up and Top-down reconciled*, in: Proc. of the First Int. Conf. on Deductive and Object-Oriented Databases, 1989.

[BdSK] M. BRUYNOOGHE, D. DE SCHREYE and B. KREKELS, *Compiling Control*, J. Logic Programming 6, 1989, 135–162.

[BdSM] M. BRUYNOOGHE, D. DE SCHREYE and B. MARTENS, *A General Criterion for Avoiding Infinite Unfolding during Partial Evaluation*, in: Proc. of the 1991 Int. Logic Programming Symposium (V. Saraswat and K. Ueda eds.), MIT Press, Cambridge Massachusetts, 1991.

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

[Ca] L. CAVEDON, *Continuity, Consistency, and Completeness Properties for Logic Programs*, in: Proc. of the Sixth Int. Conf. on Logic Programming (G. Levi and M. Martelli eds.), MIT Press, Cambridge Massachusetts, 1989, 571–584.

[Ch] D. CHAN, *Constructive Negation based on the Completed Database*, in [KB], 111–125.

[CL] C.L. CHANG and R.C. LEE, *Symbolic Logic and Mechanical Theorem Proving*, Academic Press, New York, 1973.

[Cl1] K.L. CLARK, *Predicate Logic as a Computational Formalism*, Research Report DOC 79/59, Department of Computing, Imperial College, London, 1979.

[Cl2] K.L. CLARK, *Negation as Failure*, in: Logic and Data Bases (H. Gallaire and J. Minker eds.), Plenum Press, New York, 1978, 293–322.

[CM] W. CLOCKSIN and C. MELLISH, *Programming in PROLOG*, Springer Verlag, Berlin, 1981.

[Co1] M.A. COVINGTON, *Eliminating Unwanted Loops in PROLOG*, SIGPLAN Notices 20(1), 1985, 20–26.

[Co2] M.A. COVINGTON, *A Further Note on Looping in PROLOG*, SIGPLAN Notices 20(8), 1985, 28–31.

[D] S.W. DIETRICH, *Extension Tables: Memo Relations in Logic Programming*, in: Proc. of the Symposium on Logic Programming, 1987, 264–272.

[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.

[DM] P. DEMBINSKI and J. MAŁUSZYNSKI, *And-Parallelism with Intelligent Backtracking for Annotated Logic Programs*, in: Proc. of the Symposion on Logic Programming, 1985, 29–38.

[Dr] W. DRABENT, *What is Failure? An approach to Constructive Negation*, draft 1992. Provisionally accepted by Acta Informatica.

[F] M. FITTING, *First-order Logic and Automated Theorem Proving*, Springer Verlag, Berlin, 1990.

[Fl] R.W. FLOYD, *Assigning Meanings to Programs*, in: Mathematical Aspects of Computer Science (J.T. Schwartz ed.), Proc. Symposia in Applied Mathematics (1966), vol. XIX, 1967, 19–32.

[FPS] F. FERRUCCI, G. PACINI and M.I. SESSA, *Redundancy Elimination and Loop Checks for Logic Programs*, draft 1993. To appear in Information and Computation.

[G] H. GELERNTER, *Realization of a Geometry-theorem Proving Machine*, in: Computers and Thought (E. Feigenbaum and J. Feldman eds.), McGraw-Hill, New York, 1963, 134–152.

[vG1] A. VAN GELDER, *Efficient Loop Detection in Prolog using the Tortoise-and-Hare Technique*, J. Logic Programming 4, 1987, 23–31.

[vG2] A. VAN GELDER, *Negation as Failure Using Tight Derivations for General Logic Programs*, in [M], 149–176.

[GL] M. GELFOND and V. LIFSCHITZ, *The Stable Model Semantics for Logic Programs*, in [KB], 1070–1080.

[vGRS] A. VAN GELDER, K. ROSS and J. SCHLIPF, *The Well-founded Semantics for General Logic Programs*, J. ACM 38(3), 620–650, 1991.

[H] G. HIGMAN, *Ordering by divisibility in abstract algebra's*, in: Proc. of the London Mathematical Society (3) 2 (7), 1952, 215–221.

[He] J.P. HENRARD, *Implementations of Loop Checking*, Licenciate thesis, University of Namur, Belgium, 1991.

[Hö] S. HÖLLDOBLER, *Foundations of Equational Logic Programming*, LNCS 353, Springer Verlag, Berlin, 1989.

[K] H.J. KOMOROWSKI, *A Specification of an Abstract Prolog Machine and its Application to Partial Evaluation*, Ph.D. dissertation, LSST 69, Linköping University, 1981.

[KB] R. KOWALSKI and K. BOWEN, eds., *Proc. of the Fifth Int. Conf. on Logic Programming*, MIT Press, Cambridge Massachusetts, 1988.

[KM] J.W. KLOP and J.J.CH. MEYER, *Toegepaste Logica deel I: Resolutie-logica*, Course Notes, Free University of Amsterdam, 1988 (in Dutch).

[Ko] R.A. KOWALSKI, *Algorithm = Logic + Control*, Comm. of the ACM 22(7), 1979, 424–435.

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

[KT] D.B. KEMP and R.W. TOPOR, *Completeness of a Top-Down Query Evaluation Procedure for Stratified Databases*, in [KB], 178–194.

[Ku1] K. KUNEN, *Negation in Logic Programming*, J. Logic Programming 4, 289–308, 1987.

[Ku2] K. KUNEN, *Some Remarks on the Completed Database*, in [KB], 978–992.

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

[LR] D. LOVELAND and C. REDDY, *Deleting Repeated Goals in the Problem Reduction Format*, J. of the ACM 28(4), 1981, 646–661.

[LS] J.W. LLOYD and J.C. SHEPHERDSON, *Partial Evaluation in Logic Programming*, J. Logic Programming 11, 1991, 217–242.

[M] J. MINKER, ed., *Foundations of Deductive Databases and Logic Programming*, Morgan Kaufmann, Los Altos, 1988.

[MM] A. MARTELLI and U. MONTANARI, *An Efficient Unification Algorithm*, ACM Transactions on Programming Languages and Systems 4(1), 1982, 258–282.

[N] L. NAISH, *Automating Control for Logic Programs*, J. Logic Programming 2, 1985, 167–183.

[Na] J.F. NAUGHTON, *One-Sided Recursion*, in: Proc. of the Sixth ACM Symposium on Principles of Database Systems, ACM New York, 1987, pp. 340–348.

[P] L. PLÜMER, *Termination Proofs for Logic Programs*, LNCS 446, Springer Verlag, Berlin, 1990.

[P1] T.C. PRZYMUSINSKI, *On the Declarative Semantics of Deductive Databases and Logic Programs*, in [M], 193–216.

[P2] T.C. PRZYMUSINSKI, *On the Declarative and Procedural Semantics of Logic Programs*, J. Automated Reasoning 5, 1989, 167–205.

[P3] T.C. PRZYMUSINSKI, *On Constructive Negation in Logic Programming*, in: Proc. of the 1989 North American Conf. on Logic Programming (E.L. Lusk and R. Overbeek eds.), MIT Press, Cambridge Massachusetts, 1989.

[PG] D. POOLE and R. GOEBEL, *On Eliminating Loops in PROLOG*, SIGPLAN Notices 20(8), 1985, 38–40.

[PP] H. PRZYMUSINSKA and T.C. PRZYMUSINSKI, *Weakly Perfect Model Semantics for Logic Programs*, in [KB], 1106–1120.

[PW] T.C. PRZYMUSINSKI and D.S. WARREN, *Well Founded Semantics: Theory and Implementation*, draft 1992.

[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.

[Ro] J.A. ROBINSON, *A Machine-oriented Logic Based on the Resolution Principle*, J. of the ACM 12(1), 1965, 23–41.

[S1] D. SAHLIN, *The Mixtus Approach to Automatic Partial Evaluation of Full Prolog*, in: Proc. of the 1990 North American Conf. 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.

[dSBV] D. DE SCHREYE, M. BRUYNOOGHE and K. VERSCHAETSE, *On the Existence of Non-terminating Queries for a Restricted Class of PROLOG-clauses*, Artificial Intelligence 41, 1989, 237–248..

[Sc] A. SCHMÜCKER, *On the Detection of Infinite Loops in Logic Programs*, draft, 1990.

[Se] H. SEKI, *On the Power of Alexander Templates*, in: Proc. of the Eighth Symposium om Proinciples of Database Systems, ACM SIGACT-SIGMOD, 1989.

[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 [KB], 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.

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

[UvG] J.D. ULLMAN and A. VAN GELDER, *Efficient Tests for Top-down Termination of Logical Rules*, J. ACM 35(2), 1988, 345–373.

[V] L. VIEILLE, *Recursive Query Processing: The Power of Logic*, Theoretical Computer Science 69(1), 1989, 1–53.

[VP] T. VASAK and J. POTTER, *Characterization of Terminating Logic Programs*, in: Proc. of the 1986 IEEE Symposium on Logic Programming, 1986, 140–147.

[WS] B. WANG and R.K. SHYAMASUNDAR, *Methodology for Proving the Termination of Logic Programs*, in: Proc. of the Eighth Annual Symposium on Theoretical Aspects of Computer Science (STACS91), Hamburg. LNCS 480, Springer Verlag, Berlin, 1991, 214–227.

Index

List of Notations

CIG	Context checks based on Instances and Goals	70
CIR	Context checks based on Instances and Resultants	72
$cl_P(p)$	the class of p in P	53
$C_S(x)$	the chain of x in S	17
CVG	Context checks based on Variants and Goals	70
CVR	Context checks based on Variants and Resultants	72
CWA	Closed World Assumption	5, 22
dom(θ)	domain of θ	12
D_P, D_P^*	Dependency graph of P and its reflexive, transitive closure	53
EIG_L, EIG_M	Equals Instance of Goal check	47
EIR_L, EIR_M	Equals Instance of Resultant check	48
EVG_L, EVG_M	Equals Variant of Goal check	47
EVR_L, EVR_M	Equals Variant of Resultant check	48
f_L	effect of L on an SLD-tree	26
f_L^1	effect of L on an SLS-tree	107
f_L^*	effect of L on a justified SLS-tree	107, 109
ground(P)	set of ground instances of clauses in P	13
IA	Instance of Atom check	46
I_G	Goal terminating Interpreter	175
Initials(S)	Initial subderivations of S	26
I_R	Rule terminating Interpreter	175
L(φ)	Full-comparison loop check of criterion φ	146
$L^1(\varphi)$	Single selected loop check of criterion φ	150
$L^2(\varphi)$	Double selected loop check of criterion φ	150
$L^{th}(\varphi)$	Tortoise-and-hare loop check of criterion φ	148
M_P	perfect Herbrand model of P	100
nvi	non-variable introducing	65, 66
O_L	One level loop check derived from L	118
OSC(d,size)	OverSizeCheck based on d and size	136
PTR(P)	Proof Tree Redundancy check for P	38
ran(θ)	range of θ	12
rel(A)	predicate symbol of A	53

SIG_L, SIG_M	Subsumes Instance of Goal check	58
SIR_L, SIR_M	Subsumes Instance of Resultant check	58
stratum(L)	stratum of a literal (or goal)	98
STRONG(P)	STRONG check for P	42
Succ(P,G,σ)	set of refutations of P∪{G} with a computed answer more general than G~σ	39
SVG_L, SVG_M	Subsumes Variant of Goal check	58
svo	single variable occurrence property	65
SVR_L, SVR_M	Subsumes Variant of Resultant check	58
T_{top}	top level of T	103
$U,>_U$	strict partially ordered set	140
VA	Variant of Atom check	27
VAR	the set of variables	11
var(E)	set of variables in E	12
weight(G)	weight of G	54
X^+	result of removing all negative literals from X	117
$L_1 + L_2$	sum of L_1 and L_2	33
□	empty goal	11
ε	empty substitution	12
$\Rightarrow_{C,\theta}$	derivation step with clause C and mgu θ	14, 99
$\models$	semantical implication	22
$\leq$	more general than	13
$=_L$	equality of goals seen as lists	47
$=_M$	equality of goals seen as multisets	47
$\subseteq_L$	inclusion of goals seen as lists	58
$\subseteq_M$	inclusion of goals seen as multisets	58
| |	level mapping	172
|D|	length of a derivation D (number of steps)	14
|G|	length of a goal G (number of atoms)	12
G~	conjunction of atoms of G	14
$\sim_S$	immediately related atoms in S	16
$\approx_S$	reflexive, transitive closure of $\sim_S$	16
$\sim_X$	resultants that are variants modulo X	56
$\sim_{X,G,k}$	restriction of $\sim_X$ to certain resultants	56
$[R]_{X,G,k}$	equivalence class of R under $\sim_{X,G,k}$	56

CWI TRACTS

1 D.H.J. Epema. *Surfaces with canonical hyperplane sections.* 1984.

2 J.J. Dijkstra. *Fake topological Hilbert spaces and characterizations of dimension in terms of negligibility.* 1984.

3 A.J. van der Schaft. *System theoretic descriptions of physical systems.* 1984.

4 J. Koene. *Minimal cost flow in processing networks, a primal approach.* 1984.

5 B. Hoogenboom. *Intertwining functions on compact Lie groups.* 1984.

6 A.P.W. Böhm. *Dataflow computation.* 1984.

7 A. Blokhuis. *Few-distance sets.* 1984.

8 M.H. van Hoorn. *Algorithms and approximations for queueing systems.* 1984.

9 C.P.J. Koymans. *Models of the lambda calculus.* 1984.

10 C.G. van der Laan, N.M. Temme. *Calculation of special functions: the gamma function, the exponential integrals and error-like functions.* 1984.

11 N.M. van Dijk. *Controlled Markov processes; time-discretization.* 1984.

12 W.H. Hundsdorfer. *The numerical solution of nonlinear stiff initial value problems: an analysis of one step methods.* 1985.

13 D. Grune. *On the design of ALEPH.* 1985.

14 J.G.F. Thiemann. *Analytic spaces and dynamic programming: a measure theoretic approach.* 1985.

15 F.J. van der Linden. *Euclidean rings with two infinite primes.* 1985.

16 R.J.P. Groothuizen. *Mixed elliptic-hyperbolic partial differential operators: a case-study in Fourier integral operators.* 1985.

17 H.M.M. ten Eikelder. *Symmetries for dynamical and Hamiltonian systems.* 1985.

18 A.D.M. Kester. *Some large deviation results in statistics.* 1985.

19 T.M.V. Janssen. *Foundations and applications of Montague grammar, part 1: Philosophy, framework, computer science.* 1986.

20 B.F. Schriever. *Order dependence.* 1986.

21 D.P. van der Vecht. *Inequalities for stopped Brownian motion.* 1986.

22 J.C.S.P. van der Woude. *Topological dynamix.* 1986.

23 A.F. Monna. *Methods, concepts and ideas in mathematics: aspects of an evolution.* 1986.

24 J.C.M. Baeten. *Filters and ultrafilters over definable subsets of admissible ordinals.* 1986.

25 A.W.J. Kolen. *Tree network and planar rectilinear location theory.* 1986.

26 A.H. Veen. *The misconstrued semicolon: Reconciling imperative languages and dataflow machines.* 1986.

27 A.J.M. van Engelen. *Homogeneous zero-dimensional absolute Borel sets.* 1986.

28 T.M.V. Janssen. *Foundations and applications of Montague grammar, part 2: Applications to natural language.* 1986.

29 H.L. Trentelman. *Almost invariant subspaces and high gain feedback.* 1986.

30 A.G. de Kok. *Production-inventory control models: approximations and algorithms.* 1987.

31 E.E.M. van Berkum. *Optimal paired comparison designs for factorial experiments.* 1987.

32 J.H.J. Einmahl. *Multivariate empirical processes.* 1987.

33 O.J. Vrieze. *Stochastic games with finite state and action spaces.* 1987.

34 P.H.M. Kersten. *Infinitesimal symmetries: a computational approach.* 1987.

35 M.L. Eaton. *Lectures on topics in probability inequalities.* 1987.

36 A.H.P. van der Burgh, R.M.M. Mattheij (eds.). *Proceedings of the first international conference on industrial and applied mathematics (ICIAM 87).* 1987.

37 L. Stougie. *Design and analysis of algorithms for stochastic integer programming.* 1987.

38 J.B.G. Frenk. *On Banach algebras, renewal measures and regenerative processes.* 1987.

39 H.J.M. Peters, O.J. Vrieze (eds.). *Surveys in game theory and related topics.* 1987.

40 J.L. Geluk, L. de Haan. *Regular variation, extensions and Tauberian theorems.* 1987.

41 Sape J. Mullender (ed.). *The Amoeba distributed operating system: Selected papers 1984-1987.* 1987.

42 P.R.J. Asveld, A. Nijholt (eds.). *Essays on concepts, formalisms, and tools.* 1987.

43 H.L. Bodlaender. *Distributed computing: structure and complexity.* 1987.

44 A.W. van der Vaart. *Statistical estimation in large parameter spaces.* 1988.

45 S.A. van de Geer. *Regression analysis and empirical processes.* 1988.

46 S.P. Spekreijse. *Multigrid solution of the steady Euler equations.* 1988.

47 J.B. Dijkstra. *Analysis of means in some non-standard situations.* 1988.

48 F.C. Drost. *Asymptotics for generalized chi-square goodness-of-fit tests.* 1988.

49 F.W. Wubs. *Numerical solution of the shallow-water equations.* 1988.

50 F. de Kerf. *Asymptotic analysis of a class of perturbed Korteweg-de Vries initial value problems.* 1988.

51 P.J.M. van Laarhoven. *Theoretical and computational aspects of simulated annealing.* 1988.

52 P.M. van Loon. *Continuous decoupling transformations for linear boundary value problems.* 1988.

53 K.C.P. Machielsen. *Numerical solution of optimal control problems with state constraints by sequential quadratic programming in function space.* 1988.

54 L.C.R.J. Willenborg. *Computational aspects of survey data processing.* 1988.

55 G.J. van der Steen. *A program generator for recognition, parsing and transduction with syntactic patterns.* 1988.

56 J.C. Ebergen. *Translating programs into delay-insensitive circuits.* 1989.

57 S.M. Verduyn Lunel. *Exponential type calculus for linear delay equations.* 1989.

58 M.C.M. de Gunst. *A random model for plant cell population growth.* 1989.

59 D. van Dulst. *Characterizations of Banach spaces not containing l^1.* 1989.

60 H.E. de Swart. *Vacillation and predictability properties of low-order atmospheric spectral models.* 1989.

61 P. de Jong. *Central limit theorems for generalized multilinear forms.* 1989.

62 V.J. de Jong. *A specification system for statistical software.* 1989.

63 B. Hanzon. *Identifiability, recursive identification and spaces of linear dynamical systems, part I.* 1989.

64 B. Hanzon. *Identifiability, recursive identification and spaces of linear dynamical systems, part II.* 1989.

65 B.M.M. de Weger. *Algorithms for diophantine equations.* 1989.

66 A. Jung. *Cartesian closed categories of domains.* 1989.

67 J.W. Polderman. *Adaptive control & identification: Conflict or conflux?.* 1989.

68 H.J. Woerdeman. *Matrix and operator extensions.* 1989.

69 B.G. Hansen. *Monotonicity properties of infinitely divisible distributions.* 1989.

70 J.K. Lenstra, H.C. Tijms, A. Volgenant (eds.). *Twenty-five years of operations research in the Netherlands: Papers dedicated to Gijs de Leve.* 1990.

71 P.J.C. Spreij. *Counting process systems. Identification and stochastic realization.* 1990.

72 J.F. Kaashoek. *Modeling one dimensional pattern formation by anti-diffusion.* 1990.

73 A.M.H. Gerards. *Graphs and polyhedra. Binary spaces and cutting planes.* 1990.

74 B. Koren. *Multigrid and defect correction for the steady Navier-Stokes equations. Application to aerodynamics.* 1991.

75 M.W.P. Savelsbergh. *Computer aided routing.* 1992.

76 O.E. Flippo. *Stability, duality and decomposition in general mathematical programming.* 1991.

77 A.J. van Es. *Aspects of nonparametric density estimation.* 1991.

78 G.A.P. Kindervater. *Exercises in parallel combinatorial computing.* 1992.

79 J.J. Lodder. *Towards a symmetrical theory of generalized functions.* 1991.

80 S.A. Smulders. *Control of freeway traffic flow.* 1993.

81 P.H.M. America, J.J.M.M. Rutten. *A parallel object-oriented language: design and semantic foundations.* 1992.

82 F. Thuijsman. *Optimality and equilibria in stochastic games.* 1992.

83 R.J. Kooman. *Convergence properties of recurrence sequences.* 1992.

84 A.M. Cohen (ed.). *Computational aspects of Lie group representations and related topics. Proceedings of the 1990 Computational Algebra Seminar at CWI, Amsterdam.* 1991.

85 V. de Valk. *One-dependent processes.* 1994.

86 J.A. Baars, J.A.M. de Groot. *On topological and linear equivalence of certain function spaces.* 1992.

87 A.F. Monna. *The way of mathematics and mathematicians.* 1992.

88 E.D. de Goede. *Numerical methods for the three-dimensional shallow water equations.* 1993.

89 M. Zwaan. *Moment problems in Hilbert space with applications to magnetic resonance imaging.* 1993.

90 C. Vuik. *The solution of a one-dimensional Stefan problem.* 1993.

91 E.R. Verheul. *Multimedians in metric and normed spaces.* 1993.

92 J.L.M. Maubach. *Iterative methods for non-linear partial differential equations.* 1994.

93 A.W. Ambergen. *Statistical uncertainties in posterior probabilities.* 1993.

94 P.A. Zegeling. *Moving-grid methods for time-dependent partial differential equations.* 1993.

95 M.J.C. van Pul. *Statistical analysis of software reliability models.* 1993.

96 J.K. Scholma. *A Lie algebraic study of some integrable systems associated with root systems.* 1993.

97 J.L. van den Berg. *Sojourn times in feedback and processor sharing queues.* 1993.

98 A.J. Koning. *Stochastic integrals and goodness-of-fit tests.* 1993.

99 B.P. Sommeijer. *Parallelism in the numerical integration of initial value problems.* 1993.

100 J. Molenaar. *Multigrid methods for semiconductor device simulation.* 1993.

101 H.J.C. Huijberts. *Dynamic feedback in nonlinear synthesis problems.* 1994.

102 J.A.M. van der Weide. *Stochastic processes and point processes of excursions.* 1994.

103 P.W. Hemker, P. Wesseling (eds.). *Contributions to multigrid.* 1994.

104 I.J.B.F. Adan. *A compensation approach for queueing problems.* 1994.

105 O.J. Boxma, G.M. Koole (eds.). *Performance evaluation of parallel and distributed systems - solution methods. Part 1.* 1994.

106 O.J. Boxma, G.M. Koole (eds.). *Performance evaluation of parallel and distributed systems - solution methods. Part 2.* 1994.

107 R.A. Trompert. *Local uniform grid refinement for time-dependent partial differential equations.* 1995.

108 M.N.M. van Lieshout. *Stochastic geometry models in image analysis and spatial statistics.* 1995.

109 R.J. van Glabbeek. *Comparative concurrency semantics and refinement of actions.* 1995.

110 W. Vervaat (ed.). *Probability and lattices.* 1995.

111 I. Helsloot. *Covariant formal group theory and some applications.* 1995.

112 R.N. Bol. *Loop checking in logic programming.* 1995.

113 G.J.M. Koole. *Stochastic scheduling and dynamic programming.* 1995.

MATHEMATICAL CENTRE TRACTS

1 T. van der Walt. *Fixed and almost fixed points.* 1963.

2 A.R. Bloemena. *Sampling from a graph.* 1964.

3 G. de Leve. *Generalized Markovian decision processes, part I: model and method.* 1964.

4 G. de Leve. *Generalized Markovian decision processes, part II: probabilistic background.* 1964.

5 G. de Leve, H.C. Tijms, P.J. Weeda. *Generalized Markovian decision processes, applications.* 1970.

6 M.A. Maurice. *Compact ordered spaces.* 1964.

7 W.R. van Zwet. *Convex transformations of random variables.* 1964.

8 J.A. Zonneveld. *Automatic numerical integration.* 1964.

9 P.C. Baayen. *Universal morphisms.* 1964.

10 E.M. de Jager. *Applications of distributions in mathematical physics.* 1964.

11 A.B. Paalman-de Miranda. *Topological semigroups.* 1964.

12 J.A.Th.M. van Berckel, H. Brandt Corstius, R.J. Mokken, A. van Wijngaarden. *Formal properties of newspaper Dutch.* 1965.

13 H.A. Lauwerier. *Asymptotic expansions.* 1966, out of print: replaced by MCT 54.

14 H.A. Lauwerier. *Calculus of variations in mathematical physics.* 1966.

15 R. Doornbos. *Slippage tests.* 1966.

16 J.W. de Bakker. *Formal definition of programming languages with an application to the definition of ALGOL 60.* 1967.

17 R.P. van de Riet. *Formula manipulation in ALGOL 60, part 1.* 1968.

18 R.P. van de Riet. *Formula manipulation in ALGOL 60, part 2.* 1968.

19 J. van der Slot. *Some properties related to compactness.* 1968.

20 P.J. van der Houwen. *Finite difference methods for solving partial differential equations.* 1968.

21 E. Wattel. *The compactness operator in set theory and topology.* 1968.

22 T.J. Dekker. *ALGOL 60 procedures in numerical algebra, part 1.* 1968.

23 T.J. Dekker, W. Hoffmann. *ALGOL 60 procedures in numerical algebra, part 2.* 1968.

24 J.W. de Bakker. *Recursive procedures.* 1971.

25 E.R. Paërl. *Representations of the Lorentz group and projective geometry.* 1969.

26 European Meeting 1968. *Selected statistical papers, part I.* 1968.

27 European Meeting 1968. *Selected statistical papers, part II.* 1968.

28 J. Oosterhoff. *Combination of one-sided statistical tests.* 1969.

29 J. Verhoeff. *Error detecting decimal codes.* 1969.

30 H. Brandt Corstius. *Exercises in computational linguistics.* 1970.

31 W. Molenaar. *Approximations to the Poisson, binomial and hypergeometric distribution functions.* 1970.

32 L. de Haan. *On regular variation and its application to the weak convergence of sample extremes.* 1970.

33 F.W. Steutel. *Preservations of infinite divisibility under mixing and related topics.* 1970.

34 I. Juhász, A. Verbeek, N.S. Kroonenberg. *Cardinal functions in topology.* 1971.

35 M.H. van Emden. *An analysis of complexity.* 1971.

36 J. Grasman. *On the birth of boundary layers.* 1971.

37 J.W. de Bakker, G.A. Blaauw, A.J.W. Duijvestijn, E.W. Dijkstra, P.J. van der Houwen, G.A.M. Kamsteeg-Kemper, F.E.J. Kruseman Aretz, W.L. van der Poel, J.P. Schaap-Kruseman, M.V. Wilkes, G. Zoutendijk. *MC-25 Informatica Symposium.* 1971.

38 W.A. Verloren van Themaat. *Automatic analysis of Dutch compound words.* 1972.

39 H. Bavinck. *Jacobi series and approximation.* 1972.

40 H.C. Tijms. *Analysis of (s,S) inventory models.* 1972.

41 A. Verbeek. *Superextensions of topological spaces.* 1972.

42 W. Vervaat. *Success epochs in Bernoulli trials (with applications in number theory).* 1972.

43 F.H. Ruymgaart. *Asymptotic theory of rank tests for independence.* 1973.

44 H. Bart. *Meromorphic operator valued functions.* 1973.

45 A.A. Balkema. *Monotone transformations and limit laws.* 1973.

46 R.P. van de Riet. *ABC ALGOL, a portable language for formula manipulation systems, part 1: the language.* 1973.

47 R.P. van de Riet. *ABC ALGOL, a portable language for formula manipulation systems, part 2: the compiler.* 1973.

48 F.E.J. Kruseman Aretz, P.J.W. ten Hagen, H.L. Oudshoorn. *An ALGOL 60 compiler in ALGOL 60, text of the MC-compiler for the EL-X8.* 1973.

49 H. Kok. *Connected orderable spaces.* 1974.

50 A. van Wijngaarden, B.J. Mailloux, J.E.L. Peck, C.H.A. Koster, M. Sintzoff, C.H. Lindsey, L.G.L.T. Meertens, R.G. Fisker (eds.). *Revised report on the algorithmic language ALGOL 68.* 1976.

51 A. Hordijk. *Dynamic programming and Markov potential theory.* 1974.

52 P.C. Baayen (ed.). *Topological structures.* 1974.

53 M.J. Faber. *Metrizability in generalized ordered spaces.* 1974.

54 H.A. Lauwerier. *Asymptotic analysis, part 1.* 1974.

55 M. Hall, Jr., J.H. van Lint (eds.). *Combinatorics, part 1: theory of designs, finite geometry and coding theory.* 1974.

56 M. Hall, Jr., J.H. van Lint (eds.). *Combinatorics, part 2: graph theory, foundations, partitions and combinatorial geometry.* 1974.

57 M. Hall, Jr., J.H. van Lint (eds.). *Combinatorics, part 3: combinatorial group theory.* 1974.

58 W. Albers. *Asymptotic expansions and the deficiency concept in statistics.* 1975.

59 J.L. Mijnheer. *Sample path properties of stable processes.* 1975.

60 F. Göbel. *Queueing models involving buffers.* 1975.

63 J.W. de Bakker (ed.). *Foundations of computer science.* 1975.

64 W.J. de Schipper. *Symmetric closed categories.* 1975.

65 J. de Vries. *Topological transformation groups, 1: a categorical approach.* 1975.

66 H.G.J. Pijls. *Logically convex algebras in spectral theory and eigenfunction expansions.* 1976.

68 P.P.N. de Groen. *Singularly perturbed differential operators of second order.* 1976.

69 J.K. Lenstra. *Sequencing by enumerative methods.* 1977.

70 W.P. de Roever, Jr. *Recursive program schemes: semantics and proof theory.* 1976.

71 J.A.E.E. van Nunen. *Contracting Markov decision processes.* 1976.

72 J.K.M. Jansen. *Simple periodic and non-periodic Lamé functions and their applications in the theory of conical waveguides.* 1977.

73 D.M.R. Leivant. *Absoluteness of intuitionistic logic.* 1979.

74 H.J.J. te Riele. *A theoretical and computational study of generalized aliquot sequences.* 1976.

75 A.E. Brouwer. *Treelike spaces and related connected topological spaces.* 1977.

76 M. Rem. *Associons and the closure statements.* 1976.

77 W.C.M. Kallenberg. *Asymptotic optimality of likelihood ratio tests in exponential families.* 1978.

78 E. de Jonge, A.C.M. van Rooij. *Introduction to Riesz spaces.* 1977.

79 M.C.A. van Zuijlen. *Empirical distributions and rank statistics.* 1977.

80 P.W. Hemker. *A numerical study of stiff two-point boundary problems.* 1977.

81 K.R. Apt, J.W. de Bakker (eds.). *Foundations of computer science II, part 1.* 1976.

82 K.R. Apt, J.W. de Bakker (eds.). *Foundations of computer science II, part 2.* 1976.

83 L.S. van Benthem Jutting. *Checking Landau's "Grundlagen" in the AUTOMATH system.* 1979.

84 H.L.L. Busard. *The translation of the elements of Euclid from the Arabic into Latin by Hermann of Carinthia (?), books vii-xii.* 1977.

85 J. van Mill. *Supercompactness and Wallmann spaces.* 1977.

86 S.G. van der Meulen, M. Veldhorst. *Torrix I, a programming system for operations on vectors and matrices over arbitrary fields and of variable size.* 1978.

88 A. Schrijver. *Matroids and linking systems.* 1977.

89 J.W. de Roever. *Complex Fourier transformation and analytic functionals with unbounded carriers.* 1978.

90 L.P.J. Groenewegen. *Characterization of optimal strategies in dynamic games.* 1981.

91 J.M. Geysel. *Transcendence in fields of positive characteristic.* 1979.

92 P.J. Weeda. *Finite generalized Markov programming.* 1979.

93 H.C. Tijms, J. Wessels (eds.). *Markov decision theory.* 1977.

94 A. Bijlsma. *Simultaneous approximations in transcendental number theory.* 1978.

95 K.M. van Hee. *Bayesian control of Markov chains.* 1978.

96 P.M.B. Vitányi. *Lindenmayer systems: structure, languages, and growth functions.* 1980.

97 A. Federgruen. *Markovian control problems; functional equations and algorithms.* 1984.

98 R. Geel. *Singular perturbations of hyperbolic type.* 1978.

99 J.K. Lenstra, A.H.G. Rinnooy Kan, P. van Emde Boas (eds.). *Interfaces between computer science and operations research.* 1978.

100 P.C. Baayen, D. van Dulst, J. Oosterhoff (eds.). *Proceedings bicentennial congress of the Wiskundig Genootschap, part 1.* 1979.

101 P.C. Baayen, D. van Dulst, J. Oosterhoff (eds.). *Proceedings bicentennial congress of the Wiskundig Genootschap, part 2.* 1979.

102 D. van Dulst. *Reflexive and superreflexive Banach spaces.* 1978.

103 K. van Harn. *Classifying infinitely divisible distributions by functional equations.* 1978.

104 J.M. van Wouwe. *GO-spaces and generalizations of metrizability.* 1979.

105 R. Helmers. *Edgeworth expansions for linear combinations of order statistics.* 1982.

106 A. Schrijver (ed.). *Packing and covering in combinatorics.* 1979.

107 C. den Heijer. *The numerical solution of nonlinear operator equations by imbedding methods.* 1979.

108 J.W. de Bakker, J. van Leeuwen (eds.). *Foundations of computer science III, part 1.* 1979.

109 J.W. de Bakker, J. van Leeuwen (eds.). *Foundations of computer science III, part 2.* 1979.

110 J.C. van Vliet. *ALGOL 68 transput, part I: historical review and discussion of the implementation model.* 1979.

111 J.C. van Vliet. *ALGOL 68 transput, part II: an implementation model.* 1979.

112 H.C.P. Berbee. *Random walks with stationary increments and renewal theory.* 1979.

113 T.A.B. Snijders. *Asymptotic optimality theory for testing problems with restricted alternatives.* 1979.

114 A.J.E.M. Janssen. *Application of the Wigner distribution to harmonic analysis of generalized stochastic processes.* 1979.

115 P.C. Baayen, J. van Mill (eds.). *Topological structures II, part 1.* 1979.

116 P.C. Baayen, J. van Mill (eds.). *Topological structures II, part 2.* 1979.

117 P.J.M. Kallenberg. *Branching processes with continuous state space.* 1979.

118 P. Groeneboom. *Large deviations and asymptotic efficiencies.* 1980.

119 F.J. Peters. *Sparse matrices and substructures, with a novel implementation of finite element algorithms.* 1980.

120 W.P.M. de Ruyter. *On the asymptotic analysis of large-scale ocean circulation.* 1980.

121 W.H. Haemers. *Eigenvalue techniques in design and graph theory.* 1980.

122 J.C.P. Bus. *Numerical solution of systems of nonlinear equations.* 1980.

123 I. Yuhász. *Cardinal functions in topology - ten years later.* 1980.

124 R.D. Gill. *Censoring and stochastic integrals.* 1980.

125 R. Eising. *2-D systems, an algebraic approach.* 1980.

126 G. van der Hoek. *Reduction methods in nonlinear programming.* 1980.

127 J.W. Klop. *Combinatory reduction systems.* 1980.

128 A.J.J. Talman. *Variable dimension fixed point algorithms and triangulations.* 1980.

129 G. van der Laan. *Simplicial fixed point algorithms.* 1980.

130 P.J.W. ten Hagen, T. Hagen, P. Klint, H. Noot, H.J. Sint, A.H. Veen. *ILP: intermediate language for pictures.* 1980.

131 R.J.R. Back. *Correctness preserving program refinements: proof theory and applications.* 1980.

132 H.M. Mulder. *The interval function of a graph.* 1980.

133 C.A.J. Klaassen. *Statistical performance of location estimators.* 1981.

134 J.C. van Vliet, H. Wupper (eds.). *Proceedings international conference on ALGOL 68.* 1981.

135 J.A.G. Groenendijk, T.M.V. Janssen, M.J.B. Stokhof (eds.). *Formal methods in the study of language, part I.* 1981.

136 J.A.G. Groenendijk, T.M.V. Janssen, M.J.B. Stokhof (eds.). *Formal methods in the study of language, part II.* 1981.

137 J. Telgen. *Redundancy and linear programs.* 1981.

138 H.A. Lauwerier. *Mathematical models of epidemics.* 1981.

139 J. van der Wal. *Stochastic dynamic programming, successive approximations and nearly optimal strategies for Markov decision processes and Markov games.* 1981.

140 J.H. van Geldrop. *A mathematical theory of pure exchange economies without the no-critical-point hypothesis.* 1981.

141 G.E. Welters. *Abel-Jacobi isogenies for certain types of Fano threefolds.* 1981.

142 H.R. Bennett, D.J. Lutzer (eds.). *Topology and order structures, part 1.* 1981.

143 J.M. Schumacher. *Dynamic feedback in finite- and infinite-dimensional linear systems.* 1981.

144 P. Eijgenraam. *The solution of initial value problems using interval arithmetic; formulation and analysis of an algorithm.* 1981.

145 A.J. Brentjes. *Multi-dimensional continued fraction algorithms.* 1981.

146 C.V.M. van der Mee. *Semigroup and factorization methods in transport theory.* 1981.

147 H.H. Tigelaar. *Identification and informative sample size.* 1982.

148 L.C.M. Kallenberg. *Linear programming and finite Markovian control problems.* 1983.

149 C.B. Huijsmans, M.A. Kaashoek, W.A.J. Luxemburg, W.K. Vietsch (eds.). *From A to Z, proceedings of a symposium in honour of A.C. Zaanen.* 1982.

150 M. Veldhorst. *An analysis of sparse matrix storage schemes.* 1982.

151 R.J.M.M. Does. *Higher order asymptotics for simple linear rank statistics.* 1982.

152 G.F. van der Hoeven. *Projections of lawless sequencies.* 1982.

153 J.P.C. Blanc. *Application of the theory of boundary value problems in the analysis of a queueing model with paired services.* 1982.

154 H.W. Lenstra, Jr., R. Tijdeman (eds.). *Computational methods in number theory, part I.* 1982.

155 H.W. Lenstra, Jr., R. Tijdeman (eds.). *Computational methods in number theory, part II.* 1982.

156 P.M.G. Apers. *Query processing and data allocation in distributed database systems.* 1983.

157 H.A.W.M. Kneppers. *The covariant classification of two-dimensional smooth commutative formal groups over an algebraically closed field of positive characteristic.* 1983.

158 J.W. de Bakker, J. van Leeuwen (eds.). *Foundations of computer science IV, distributed systems, part 1.* 1983.

159 J.W. de Bakker, J. van Leeuwen (eds.). *Foundations of computer science IV, distributed systems, part 2.* 1983.

160 A. Rezus. *Abstract AUTOMATH.* 1983.

161 G.F. Helminck. *Eisenstein series on the metaplectic group, an algebraic approach.* 1983.

162 J.J. Dik. *Tests for preference.* 1983.

163 H. Schippers. *Multiple grid methods for equations of the second kind with applications in fluid mechanics.* 1983.

164 F.A. van der Duyn Schouten. *Markov decision processes with continuous time parameter.* 1983.

165 P.C.T. van der Hoeven. *On point processes.* 1983.

166 H.B.M. Jonkers. *Abstraction, specification and implementation techniques, with an application to garbage collection.* 1983.

167 W.H.M. Zijm. *Nonnegative matrices in dynamic programming.* 1983.

168 J.H. Evertse. *Upper bounds for the numbers of solutions of diophantine equations.* 1983.

169 H.R. Bennett, D.J. Lutzer (eds.). *Topology and order structures, part 2.* 1983.