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 \( A \).
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 \textit{SLD-resolution}: the construction of an \textit{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 \textit{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:

\[
P = \{ \text{tc}(x,y) \leftarrow r(x,y). \\
                  \text{tc}(x,y) \leftarrow r(x,z), \text{tc}(z,y). \}
\]

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 \textit{logically} P is the right program for computing the transitive closure of \(r\), \textit{operationally} it is not. One solution is to write a different program, which is not straightforward – see for example the program in [CM, Section7.2]. 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.

\textit{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 \textit{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 \textit{sound} if it does not prune an SLD-tree to such an extent that solutions are lost. We call it \textit{weakly sound} if its application results in the loss of \textit{some} solutions, but not in the loss of \textit{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 \textit{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

\[
P = \{ q \leftarrow p(x). \\
p(a) \leftarrow p(b). \\
p(b) \leftarrow p(c). \}.
\]

The SLD-derivation \( q \Rightarrow p(x) \Rightarrow \{x/a\} \Rightarrow p(b) \Rightarrow \Leftarrow p(c) \) fails finitely, but it is pruned at \( \Leftarrow p(b) \) 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):

\[
\text{if } DB \not\models \varphi \text{ then } DB \models \neg \varphi,
\]

that is: deduce \( \neg \varphi \) if \( \varphi \) 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 \( A \theta \) such that \( DB \models A \theta \). In other words: compute all answers to the query \( A \). Indeed, when \( A \) is ground and \( DB \not\models A \), 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 \textit{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 \textit{equality} between
goals. They are complete for function-free \textit{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 \textit{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 \textit{context} of the atom) into account. The context checks are complete for the
three classes of programs mentioned.

\textit{Another example}

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

\[
EAX = \{ \begin{align*}
eq(x,x) & \leftarrow. \quad \text{(reflexivity)}, \\
eq(x,y) & \leftarrow \eq(y,x). \quad \text{(symmetry)}, \\
eq(x,y) & \leftarrow \eq(x,z),\eq(z,y). \quad \text{(transitivity)}, \\
eq(f(x_1,\ldots,x_n),f(y_1,\ldots,y_n)) & \leftarrow \eq(x_1,y_1),\ldots,\eq(x_n,y_n). \quad \text{(substitutivity)} \}. \\
\end{align*}
\]
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:

\[
\begin{align*}
\text{eq}(t,u) & \quad \text{(symmetry)} & \quad \text{eq}(t,u) & \quad \text{(transitivity)} \\
\text{eq}(u,t) & \quad \text{(symmetry)} & \quad \text{eq}(t,z) & \quad \text{eq}(z,u) & \quad \text{(reflexivity)} \\
\text{eq}(t,u) & \quad \text{(symmetry)} & \quad \text{eq}(t,u) & \quad \text{(reflexivity)} \\
\end{align*}
\]

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.