Program Verification and Prolog

Krzysztof R. Apt

Abstract

We show here that verification of Prolog programs can be systematically carried out within a simple framework which comprises syntactic analysis, declarative semantics, modes and types. We apply these techniques to study termination, partial correctness, occur-check freedom, absence of errors and absence of floundering. Finally, we discuss which aspects of these techniques can be automated.
Notes. This research was partly supported by the ESPRIT Basic Research Action 6810 (Compulog 2). A preliminary, shorter, version of this paper appeared as Apt [3].

1 Introduction

1.1 Motivation

Prolog is 20 years old and so is logic programming. However, they were developed separately and these two developments never really merged. The first track is best exemplified by Sterling and Shapiro [36], which puts emphasis on programming style and techniques, and the second by Lloyd [25], which concentrates on the theoretical foundations. As a result of these separate developments, until recently little work was done on verification and development of Prolog programs.

It is natural and almost self-evident to base verification of Prolog programs on the theory of logic programming. However, the choices made in logic programming theory do not necessarily coincide with those made in Prolog (like the choice of a selection rule) and its extensions and modifications. Some new issues (like the occur-check problem) need to be addressed and additional results (like those dealing with termination) need to be established.

The aim of this chapter is to provide an overview of our recent work on verification of Prolog programs. We show that many relevant properties of Prolog programs can be established by means of simple arguments. In particular, we explain how termination and partial correctness can be dealt with by studing declarative interpretation of logic programs. Termination is handled by techniques developed in Apt and Pedreschi [8] and Apt and Pedreschi [9].

We also study here run-time properties. These are properties which refer to the program execution. Examples of such properties include the absence of the occur-check problem, which states that the omission of the occur-check in the unification algorithm does not result in incorrect use of unification, and the absence of run-time errors in the presence of arithmetic operations.

To prove run-time properties of Prolog programs we introduce increasingly more powerful tools. When dealing with the occur-check problem and with the absence of floundering in presence of negation we use syntactic analysis and modes. We follow here the approach of Apt and Pellegrini [10]. Then, when dealing with the absence of run-time errors for Prolog programs with arithmetic, we use directional types, proposed recently by Bronsard, Lakshman and Reddy [14].

1.2 Terminology and Notation

We work here with *queries*, that is sequences of atoms, instead of *goals*, that is constructs of the form $\leftarrow Q$, where Q is a query. We denote by $\square$ the empty query. Throughout the chapter we restrict attention to one selection rule, namely Prolog's leftmost selection rule. We refer to SLD-resolution with the leftmost selection rule as *LD-resolution*. All proof-theoretic notions, such as the computed answer substitution, refer to LD-resolution.

Given two syntactic expressions E and F, we say that E is *more general than* F, and write $E \leq F$, if $E\theta = F$ for some substitution θ. We denote the set of variables occurring in an expression E by $Var(E)$. Given a list `t` we write `a` $\in$ `t` when `a` is a member of `t` and `a` $\notin$ `t` when `a` is not a member of `t`. Also, we identify here constants with 0-ary function symbols.

Apart from this we use the standard notation of Lloyd [25] and Apt [2]. In particular, for a program P, B_P stands for its Herband base, M_P stands for its least Herbrand model, $ground(P)$ for the set of all ground instances of clauses of P, and $[A]$ for the set of all ground instances of the atom A.

2 Setting the Stage

2.1 Syntax

We shall deal here with three subsets of Prolog.

2.1.1 *Pure Prolog*

The syntax of programs written in this subset coincides with the customary syntax of logic programs, though the *ambivalent syntax* and *anonymous variables* are allowed.

Let us explain both concepts. In first-order logic, and consequently in logic programming, it is assumed that function symbols and relation symbols of different arity form mutually disjoint classes of symbols. While this assumption is rarely stated explicitly, it is a folklore postulate in mathematical logic which can be easily tested by exposing a logician to Prolog syntax and waiting for his protests. Namely, in contrast to first-order logic, Prolog allows ambivalent syntax. Thus we can use a binary relation symbol `member`, unary function symbol `member` and a binary function symbolmember, and build syntactically legal facts like `member(member(a,b), [c, member(a)])`. Such expressions can be uniquely parsed once the context is given in which they occur.

The ambivalent syntax at this level is not an issue and it is safe to assume it when studying formally pure Prolog programs. The ambivalent syntax becomes an interesting subject at the moment of considering meta-interpreters which use the `clause` relation – see Kalsbeek [21] and Martens and De Schreye [28] for recent work on this topic. All in all, it is a minor point in this article but still worth mentioning.

Prolog also allows so-called *anonymous variables*, written as "_" (underscore). These variables have a special interpretation, because each occurrence of "_" in a query or in a clause is interpreted as a *different* variable. Thus by definition each anonymous variable occurs in a query or a clause only once. Anonymous variables form a simple and elegant device which sometimes increases the readability of programs in a remarkable way.

2.1.2 *Pure Prolog with Arithmetic*

This subset extends the previous one by allowing in the bodies of the program clauses the arithmetic comparison operators $<, \leq, =:=, \neq, \geq, >$ and the binary "`is`" relation of Prolog.

2.1.3 *Pure Prolog with Negation*

This subset extends the first one by allowing negative literals in the bodies of the program clauses. Thus it coincides with the syntax of general logic programs.

The methods discussed in this chapter can be readily used to deal with the "union" of the last two subsets, that is pure Prolog with arithmetic and negation.

When considering a specific logic program one has to fix a first-order language w.r.t. which it is analyzed. Usually, one associates with the program the language determined by it – its function and relation symbols are the ones occurring in the program (see, e.g., Lloyd [25] and Apt [2]). Another choice was made by Kunen [23] who assumed a universal first-order language with infinitely many function and relation symbols in each arity, in which all programs and queries are written. One can think of this language as the language defined by a Prolog manual.

In this chapter we follow Kunen's choice. In contrast to the other alternative it imposes no syntactic restriction on the queries which may be used for a given program. This better reflects the reality of programming. In Section 2.3 we shall indicate another advantage of this choice. Of course, the sets $ground(P)$ and $[A]$ refer to the ground instances in this universal language. All considered interpretations are interpretations of this universal language.

2.2 Proof Theory

Let us now explain the proof theory for the three subsets introduced above.

2.2.1 *Pure Prolog*

We use, as expected, the LD-resolution. However, in most implementations of Prolog, unification without the occur-check is used. Hence we have to deal with this issue.

Moreover, we assume that, as in Prolog, the clauses of the program are ordered. This ordering will be reflected in the considered LD-trees. It should be added, however, that in our approach to correctness the ordering of the clauses will *never* play any role. In other words, our approach will not be able to distinguish between programs which differ only by the clause ordering. We shall return to this point in Section 3.1, when studying termination.

2.2.2 *Pure Prolog with Arithmetic*

Consider the program `QUICKSORT`:

```
qs(Xs, Ys) ← Ys is an ordered permutation of the list Xs.
qs([X | Xs], Ys) ←
   part(X, Xs, Littles, Bigs),
   qs(Littles, Ls),
   qs(Bigs, Bs),
   app(Ls, [X | Bs], Ys).
qs([], []).

part(X, Xs, Ls, Bs) ← Ls is a list of elements of Xs which are < X,
                      Bs is a list of elements of Xs which are ≥ X.
part(X, [Y|Xs], [Y|Ls], Bs) ← X > Y, part(X, Xs, Ls, Bs).
part(X, [Y|Xs], Ls, [Y|Bs]) ← X ≤ Y, part(X, Xs, Ls, Bs).
part(_, [], [], []).
```

augmented by the `APPEND` program `defined by:`

```
   app(Xs, Ys, Zs) ← Zs is the concatenation of the lists Xs and Ys.
   app([X | Xs], Ys, [X | Zs]) ← app(Xs, Ys, Zs).
   app([], Ys, Ys).
```

When studying it formally as a Prolog program we have to decide the status of the built-in's $>$ and $\leq$. Are they some further unspecified relation symbols whose definitions we can ignore? Well, with this choice we face the following problem. In Prolog the relations $>$ and $\leq$ are built-in's whose evaluation results in an error when its arguments are not ground arithmetic expressions (in short, gae's). Consequently, the query `qs([3,4,X,7], [3,4,7,8])` results in an error at the moment the variable `X` becomes an argument of $>$.

Now, logic programming does not have any facilities to deal with run-time errors, but at least one could consider trading them for failure. Unfortunately, this is not possible. Otherwise, for some terms `s` and `t` the query `s>t` would succeed, and then by the Lifting Lemma the query `X>Y` would succeed as well. So what is the conclusion? The standard theory of logic programming *cannot* be used to capture properly the behaviour of the built-in's $>$ and $\leq$, and it is not possible to model the fact that the query `qs([3,4,X,7], [3,4,7,8])` results in an error.

To model Prolog's interpretation of arithmetic relations within logic programming we follow Kunen [22]. First, we extend the LD-resolution by stipulating that an LD-derivation *ends in an error* when at the moment of evaluation the arguments of the comparison relations are not gae's. In the case of the assignment `s is t`, an error results when at the moment of evaluation `t` is not a gae.

Next, we add to each program infinitely many clauses which define the ground instances of the used arithmetic relations. Given a gae `n` we denote by `val(n)` its value. For example, `val(3+4)` equals 7. So for $<$ we add the following set of unit clauses:

$$M_{<} = \{\texttt{m} < \texttt{n} \mid \texttt{m, n} \text{ are gae's and } \texttt{val(m)<val(n)}\},$$

for "`is`" we add the set

$$M_{\texttt{is}} = \{\texttt{val(n) is n} \mid \texttt{n} \text{ is a gae}\},$$

etc. So, for example, $\texttt{7 is } 3+4 \in M_{\texttt{is}}$. We also assume that, conforming to the status of built-in's, in the original program arithmetical relations are not used in clause heads.

These added clauses allow us to compute resolvents when the selected atom involves an arithmetic relation. For example, the query `X is 3+4, X < 2+3` resolves to only one query, namely `7 < 2+3` (using the clause `7 is 3+4`) and the query `7 < 2+3` fails. Thus all LD-derivations of the query `X is 3+4, X < 2+3` fail, which agrees with Prolog's interpretation.

Note that thanks to the "ending in an error" provision every query with a selected atom involving an arithmetic relation has at most one descendant in every LD-tree. Consequently, in spite of the fact that the considered programs contain now infinitely many clauses, the resulting LD-trees remain finitely branching.

2.2.3 *Pure Prolog with Negation*

As expected, to interpret these programs we use the SLDNF-resolution with the leftmost selection rule, further referred to as LDNF-resolution. Less expected is the fact that the usual definition of the SLDNF-resolution given in Lloyd [25] needs to be modified.

We leave to the reader the task of checking that according to the definition of SLDNF-resolution given in Clark [16] and reproduced in Lloyd [24] it is not clear what is the SLDNF-derivation for the program $P = \{p \leftarrow p\}$, and the query $\neg p$, whereas according to the definition given in Lloyd [25] no SLDNF-derivations exist for the program $P = \{p \leftarrow \neg p\}$ and query p. The problem with the first definition is that it is circular and not all cases for forming a resolvent are defined, whereas the latter definition is mathematically correct, but more restrictive than the first one.

It should be pointed out here that the latter definition is *sufficient* for proving soundness and various forms of completeness of SLDNF-resolution. However, when reasoning about termination of Prolog programs we need to have at our disposal a definition of SLDNF-resolution (with the leftmost selection rule) which properly formalizes the computation process and not only correctly predicts the computed results.

Such a definition was proposed by Martelli and Tricomi [27]. In their revision the subsidiary trees used to resolve negative literals are built "inside" the main tree. Another solution was suggested later in Apt and Doets [5] where, as in the original definition, the subsidiary trees are kept "aside" of the "main" tree but their construction is no longer viewed as an atomic step in the resolution process.

Additionally, when studying the LDNF-resolution we need to modify the definition of floundering. It occurs when a negative non-ground literal is selected. We say that $P \cup \{Q\}$ *does not flounder* if no LDNF-derivation of $P \cup \{Q\}$ flounders.

It is perhaps useful to recall here that Prolog ignores floundering. This leads to a number of well-known complications and explains why it is natural to seek conditions which ensure absence of floundering. In fact, our methods for proving termination and partial correctness of general programs do rely on the absence of floundering.

2.3 Semantics

There is no universal agreement as to what is the declarative semantics of a logic program. In this chapter we advocate for a program without negation the use of its least Herband model as its declarative semantics. However, we have to be careful when making this seemingly unique choice.

Consider the proverbial `APPEND` program. With the first choice of Subsection 2.1 the underlying first-order language has only one constant, viz. `[]`, and one, binary, function symbol `[.|.]`. Thus the Herbrand universe consists of all ground lists whose flattened form is a list with all elements equal to `[]`. Call such lists *trivial*. It is easy to see that then

$$M_{\texttt{APPEND}} = \{\texttt{app}(\texttt{s}, \texttt{t}, \texttt{u}) \mid \texttt{s}, \texttt{t}, \texttt{u} \text{ are trivial lists and } \texttt{s} * \texttt{t} = \texttt{u}\},$$

where "* " denotes the operation of concatenating two lists. This is the semantics of the `APPEND` program given in Sterling and Shapiro [36]. Clearly, it cannot be used to render the meaning of queries in which function symbols other than [] and [.|.] are used.

As soon as the underlying first-order language has another constant than [], and so in particular in our case, the Herbrand universe contains elements which are not lists. Consequently, on the account of the second clause of APPEND, $M_{\texttt{APPEND}}$ contains elements of the form app(s,t,u) where neither t nor u is a list. (On the other hand, it is still the case that whenever app(s,t,u) $\in M_{\texttt{APPEND}}$, then s is a list.) So the choice of the first-order language affects the structure of the least Herbrand models of the considered programs.

The fact that APPEND and various other well-known programs do admit "ill-typed" atoms in their least Herbrand models complicates matters somewhat. To simplify our presentation we therefore continue our discussion with the "correctly typed" version of APPEND, which we denote by APPEND-T:

```
app([X | Xs], Ys, [X | Zs]) ← app(Xs, Ys, Zs).
app([], Ys, Ys) ← list(Ys).
```

augmented by the LIST program defined by:

```
list(Xs) ← Xs is a list.
list([_ | Ts]) ← list(Ts).
list([]).
```

Note that

$$\begin{aligned} M_{\texttt{APPEND-T}} &= \{\texttt{app}(\texttt{s},\texttt{t},\texttt{u}) \mid \texttt{s},\texttt{t},\texttt{u} \text{ are g. lists,} \texttt{s} * \texttt{t} = \texttt{u}\} \\ &\cup\ M_{\texttt{LIST}}, \end{aligned}$$

where

$$M_{\texttt{LIST}} = \{\texttt{list}(\texttt{s}) \mid \texttt{s} \text{ is a g. list}\}.$$

Here and elsewhere "g. list(s)" stands for "ground list(s)".

We shall return to the original program APPEND in Section 6.1. Discussion of the semantics of the other two fragments of Prolog is postponed until Sections 4.2 and 5.3.

3 Pure Prolog

We now discuss correctness of programs written in the three defined subsets of Prolog. We start with pure Prolog.

3.1 Termination

First we consider termination. We present here the approach of Apt and Pedreschi [8]. It is a modification of a method of Bezem [12] which deals with termination w.r.t. all selection rules. For simplicity we restrict our attention here to one atom queries. We recall the relevant concepts.

Definition 3.1 *A program is called left terminating if all its LD-derivations starting with a ground query are finite.* □

To prove that a program is left terminating, and to characterize the queries that terminate w.r.t. such a program, the following notions are introduced.

Definition 3.2

- *A level mapping for a program P is a function $|\ |: B_P \rightarrow N$ from ground atoms to natural numbers. For $A \in B_P$, $|A|$ is the level of A.*
- *An atom A is called bounded with respect to a level mapping $|\ |$, if $|\ |$ is bounded on the set $[A]$ of ground instances of A. For A bounded w.r.t. $|\ |$, we define $|A|$, the level of A w.r.t. $|\ |$, as the maximum $|\ |$ takes on $[A]$.*
- *A clause is called acceptable with respect to $|\ |$ and an interpretation I, if I is its model and for every ground instance $A \leftarrow \mathbf{A}, B, \mathbf{B}$ of it such that $I \models \mathbf{A}$*
$$|A| > |B|.$$
- *A program P is called acceptable with respect to $|\ |$ and I, if all its clauses are. P is called acceptable if it is acceptable with respect to some level mapping and an interpretation.* □

The following results link the introduced notions.

Theorem 3.3 *Let P be acceptable w.r.t. $|\ |$ and I. Then, for every atom A bounded w.r.t. $|\ |$, all LD-derivations of $P \cup \{A\}$ are finite. In particular, P is left terminating.* □

Theorem 3.4 *Let P be a left terminating program. Then, for some level mapping $|\ |$ and a Herbrand interpretation I,*

(i) *P is acceptable w.r.t. $|\ |$ and I,*

(ii) *for every atom A, all LD-derivations of $P \cup \{A\}$ are finite iff A is bounded w.r.t. $|\ |$.* □

The model I represents the limited declarative knowledge needed to prove termination. Note that using Theorem 3.3 we deal can only establish termination of a query w.r.t. a left terminating program and we use here the notion of so-called "universal" termination, according to which the query terminates irrespectively of the clause ordering. We found that this strong form of termination is satisfied by most pure Prolog programs and queries considered in standard books on Prolog.

To see how this method of proving termination can be applied to specific programs we now consider a couple of examples. When dealing with them we use the following function $|\ |$ from ground terms to natural numbers:

$$\begin{aligned} |[x|xs]| &= |xs| + 1, \\ |f(x_1, \ldots, x_n)| &= 0 \text{ if } f \neq [\,.\,|\,.\,]. \end{aligned}$$

Then for a list xs, $|xs|$ equals its length.

Palindrome

First, let us consider a program whose proof of termination does not involve the choice of the model I. In the following program PALINDROME-T:

```
palindrome(Xs) ← the list Xs equals to its reverse.
palindrome(Xs) ← reverse(Xs, Xs).

reverse(Xs, Ys) ←  Ys is the reverse of the list Xs.
reverse(X1s, X2s) ← reverse(X1s, [], X2s).

reverse(Xs, Ys, Zs) ← Zs is the result of concatenating
                      the reverse of the list Xs and the list Ys.
reverse([X | X1s], X2s, Ys) ← reverse(X1s, [X | X2s], Ys).
reverse([], Xs, Xs) ← list(Xs).
```

augmented by the LIST program,

the body of each clause has at most one atom. In this case the reduction of the level mapping required in the definition of acceptability has to be achieved irrespective of the choice of the model of the program. The following level mapping $|\ |$ does the job:

$$
\begin{aligned}
|\texttt{palindrome}(\texttt{xs})| &= 2\cdot|\texttt{xs}|+3,\\
|\texttt{reverse}(\texttt{xs},\texttt{ys})| &= 2\cdot|\texttt{xs}|+2,\\
|\texttt{reverse}(\texttt{xs},\texttt{ys},\texttt{zs})| &= 2\cdot|\texttt{xs}|+|\texttt{ys}|+1,\\
|\texttt{list}(\texttt{xs})| &= |\texttt{xs}|.
\end{aligned}
$$

We leave it to the reader to check that PALINDROME-T is indeed acceptable w.r.t. the level mapping $|\ |$ and the Herbrand model $B_{\texttt{PALINDROME-T}}$ (or any other model) of PALINDROME-T. Moreover, for a list xs, the query palindrome(xs) is bounded w.r.t. $|\ |$ and consequently, by Theorem 3.3, all LD-derivations of PALINDROME-T $\cup$ {palindrome(xs)} are finite.

Sequence

The choice of the level mapping and of the model can affect the class of queries whose termination can be established. To see this consider the following problem from Coelho and Cotta [17] (see page 193) and its formalization in Prolog: arrange three 1's, three 2's, ..., three 9's in sequence so that for all $i \in [1,9]$ there are exactly i numbers between successive occurrences of i.

```
sublist(Xs, Ys) ← Xs is a sublist of the list Ys.
sublist(Xs, Ys) ← app(_, Zs, Ys), app(Xs, _, Zs).

sequence(Xs) ← Xs is a list of 27 elements.
sequence([_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_]).
```

```
question(Ss) ← Ss is the desired list of 27 elements.
question(Ss) ←
   sequence(Ss),
   sublist([1,_,1,_,1], Ss),
   sublist([2,_,_,2,_,_,2], Ss),
   sublist([3,_,_,_,3,_,_,_,3], Ss),
   sublist([4,_,_,_,_,4,_,_,_,_,4], Ss),
   sublist([5,_,_,_,_,_,5,_,_,_,_,_,5], Ss),
   sublist([6,_,_,_,_,_,_,6,_,_,_,_,_,_,6], Ss),
   sublist([7,_,_,_,_,_,_,_,7,_,_,_,_,_,_,_,7], Ss),
   sublist([8,_,_,_,_,_,_,_,_,8,_,_,_,_,_,_,_,_,8], Ss),
   sublist([9,_,_,_,_,_,_,_,_,_,9,_,_,_,_,_,_,_,_,_,9], Ss).
```

augmented by the `APPEND-T` program.

Call the above program `SEQUENCE-T`. For those curious to know, there are 6 solutions to this problem, generated by the above program:

```
| ?- question(Ss).

Ss = [7,5,3,8,6,9,3,5,7,4,3,6,8,5,4,9,7,2,6,4,2,8,1,2,1,9,1];

Ss = [3,4,7,9,3,6,4,8,3,5,7,4,6,9,2,5,8,2,7,6,2,5,1,9,1,8,1];

Ss = [3,4,7,8,3,9,4,5,3,6,7,4,8,5,2,9,6,2,7,5,2,8,1,6,1,9,1];

Ss = [1,9,1,6,1,8,2,5,7,2,6,9,2,5,8,4,7,6,3,5,4,9,3,8,7,4,3];

Ss = [1,8,1,9,1,5,2,6,7,2,8,5,2,9,6,4,7,5,3,8,4,6,3,9,7,4,3];

Ss = [1,9,1,2,1,8,2,4,6,2,7,9,4,5,8,6,3,4,7,5,3,9,6,8,3,5,7];

no
```

It is straightforward to verify that `SEQUENCE-T` is acceptable w.r.t. the level mapping $|\ |$ defined by:

$$
\begin{aligned}
|\texttt{question(xs)}| &= |\texttt{xs}| + 30,\\
|\texttt{sequence(xs)}| &= 0,\\
|\texttt{sublist(xs,ys)}| &= |\texttt{xs}| + |\texttt{ys}| + 2,\\
|\texttt{app(xs,ys,zs)}| &= \min(|\texttt{xs}|, |\texttt{zs}|) + 1,\\
|\texttt{list(xs)}| &= |\texttt{xs}|,
\end{aligned}
$$

and the model $B_{\texttt{SEQUENCE-T}}$. However, with this choice of the level mapping we face the problem that the atom `question(Ss)` is not bounded. Consequently, we cannot use Theorem 3.3 to prove termination of this query. In fact, using this level mapping we can only prove that for `s` ground, all LD-derivations of `SEQUENCE-T` $\cup$ {`question(s)`} are finite.

To prove the stronger termination property we change the above level mapping by putting

$$|\texttt{question(xs)}| \quad = \quad 57,$$

and choose any model I of `SEQUENCE-T` such that for a ground `s`

$$I \models \texttt{sequence(s)} \text{ iff } \texttt{s} \text{ is a list of 27 elements.}$$

Then `SEQUENCE-T` is acceptable w.r.t. $|\ |$ and I. Moreover, the query `question(Ss)` is now bounded w.r.t. $|\ |$ and consequently, by Theorem 3.3, all LD-derivations of `SEQUENCE-T` $\cup$ {`question(Ss)`} are finite.

3.1.1 *An Improvement*

The definition of acceptability requires a strict decrease of the level mapping from the clause head to the atoms of the clause body. Apt and Pedreschi [9] observed that this requirement can be relaxed in the case of non-recursive calls. This leads to an alternative definition of acceptability, that we qualify with the prefix *semi*. This notion is actually equivalent to the original one, but it gives rise to a more flexible proof method.

To describe this modification we need to define first when two relation symbols occurring in a program are mutually recursive.

Definition 3.5 *Let P be a program and p, q relation symbols occurring in it.*

- *We say that p refers to q in P if there is a clause in P that uses p in its head and q in its body.*
- *We say that p depends on q in P, and write $p \sqsupseteq q$, if (p, q) is in the reflexive, transitive closure of the relation refers to.*
- *We say that p and q are mutually recursive, and write $p \simeq q$, if $p \sqsupseteq q$ and $q \sqsupseteq p$. In particular, p and p are mutually recursive.* □

We also write $p \sqsupset q$ when $p \sqsupseteq q$ and $q \not\sqsupseteq p$. The following definition of *semi-acceptability* exploits the introduced orderings over the relation symbols. We denote here by $rel(A)$ the relation symbol occurring in atom A.

Definition 3.6 *Let P be a program, $|\ |$ a level mapping for P and I an interpretation.*

- *A clause of P is called semi-acceptable with respect to $|\ |$ and I, if I is its model and for every ground instance $A \leftarrow \mathbf{A}, B, \mathbf{B}$ of it such that $I \models \mathbf{A}$*
  - $|A| > |B|$ *if* $rel(A) \simeq rel(B)$,
  - $|A| \geq |B|$ *if* $rel(A) \sqsupset rel(B)$.

- *A program P is called semi-acceptable with respect to $|\ |$ and I, if all its clauses are. P is called semi-acceptable if it is semi-acceptable with respect to some level mapping and an interpretation.* □

Thus the level mapping is required to decrease from an atom A in the head of a clause to an atom B in the body of that clause only if the relations of A and B are mutually recursive. Additionally, the level mapping is required not to increase from A to B if the relations of A and B are not mutually recursive.

The following observations are immediate.

Note 3.7 *If a program is acceptable w.r.t. $|\ |$ and I, then it is semi-acceptable w.r.t. $|\ |$ and I.* □

Note 3.8 *If a program is semi-acceptable w.r.t. $|\ |$ and I, then it is acceptable w.r.t. a level mapping $\|\ \|$ and the same interpretation I. Moreover, for each atom A, if A is bounded w.r.t. $|\ |$, then A is bounded w.r.t. $\|\ \|$.* □

This brings us to the following conclusion.

Corollary 3.9 *A program is acceptable iff it is semi-acceptable.* □

To see how the notion of semi-acceptability leads to more natural level mappings reconsider the programs studied before.

Palindrome

When proving that `PALINDROME-T` is acceptable, we had to repeatedly use "+1" to ensure the decrease of the level mapping. Now a simpler level mapping $|\ |$ suffices:

$$
\begin{aligned}
|\mathtt{palindrome(xs)}| &= 2 \cdot |\mathtt{xs}|, \\
|\mathtt{reverse(xs, ys)}| &= 2 \cdot |\mathtt{xs}|, \\
|\mathtt{reverse(xs, ys, zs)}| &= 2 \cdot |\mathtt{xs}| + |\mathtt{ys}|, \\
|\mathtt{list(xs)}| &= |\mathtt{xs}|.
\end{aligned}
$$

It is straightforward to check that `PALINDROME-T` is semi-acceptable w.r.t. the level mapping $|\ |$ and $B_{\mathtt{PALINDROME-T}}$.

Sequence

It is easy to see that `SEQUENCE-T` is semi-acceptable w.r.t. the level mapping $|\ |$ defined by:

$$
\begin{aligned}
|\mathtt{question(xs)}| &= 54, \\
|\mathtt{sequence(xs)}| &= 0, \\
|\mathtt{sublist(xs, ys)}| &= |\mathtt{xs}| + |\mathtt{ys}|, \\
|\mathtt{app(xs, ys, zs)}| &= \min(|\mathtt{xs}|, |\mathtt{zs}|), \\
|\mathtt{list(xs)}| &= |\mathtt{xs}|
\end{aligned}
$$

and (as before) any model I of `SEQUENCE-T` such that for a ground `s`

$$I \models \texttt{sequence(s)} \text{ iff } \texttt{s} \text{ is a list of 27 elements.}$$

Again, in the above level mapping it was possible to disregard the accumulated use of "+1" 's.

This approach was further generalized in Apt and Pedreschi [9] to a yield a modular method of proving left termination. It was applied there to a number of non-trivial examples including the `MAP_COLOR` program from Sterling and Shapiro [36] (see page 212) which generates a colouring of a map in such a way that no two neighbours have the same colour.

It should be made clear here that due to Theorems 3.3 and 3.4 it is undecidable whether a program is acceptable. Starting with Ullman and Van Gelder [38] a lot of attention has been devoted to a study of sufficient, decidable conditions for proving left termination, or more generally, left termination of a given query and a program. An interested reader is referred to the recent survey article of De Schreye and Decorte [32] and the last section of this chapter.

3.2 Partial Correctness

Our approach to partial correctness is based on the use of the least Herbrand model M_P. We restrict our attention to left terminating programs. This explains why we treated termination first. The following observation of Apt and Pedreschi [8] explains why for a left terminating program it is easier to verify that a Herbrand interpretation is its least Herbrand model.

Definition 3.10 *We say that a model I of a program P is supported if for every ground atom A such that $I \models A$ there exists $\mathbf{B}$ such that $A \leftarrow \mathbf{B} \in ground(P)$ and $I \models \mathbf{B}$.* □

Intuitively, $\mathbf{B}$ is an explanation (or support) for the truth of A in I.

Lemma 3.11 *For a left terminating program P, M_P is the unique supported Herbrand model of P.* □

Now, for all programs considered here, and for plenty of other "correctly typed" programs, checking that a given Herbrand interpretation is a supported model is straightforward. Consequently, by virtue of the above lemma, for a left terminating program, we omit the proof that a given Herbrand interpretation is its least Herbrand model.

Of course, it is legimitate to ask how one finds a candidate for the least Herbrand model. According to our experience it is usually the "specification" of the program limited to ground queries. We do not consider here the problem of in what language it is most convenient to write this specification.

In the sequel it will be more convenient to work with the instances of the queries instead of with the substitutions. More precisely, we introduce the following definition.

Definition 3.12 *Consider a program P.*

- *We say that Q' is a correct instance of the query Q, if for some correct answer substitution θ for Q, $Q' = Q\theta$; that is , if Q' is an instance of Q and $P \models Q'$.*
- *We say that Q' is a computed instance of the query Q if for some computed answer substitution θ for Q, $Q' = Q\theta$.* □

Clearly, a unique correct (resp. computed) answer substitution can be computed from a query and its correct (resp. computed) instance in a straightforward way. So considering instances instead of substitutions is just a matter of convenience. Using this terminology the usual soundness and strong completeness properties of logic programs, now restricted to the leftmost selection rule, can be formulated as follows.

Theorem 3.13 (Soundness of LD-resolution) *Consider a program P and a query Q. Every computed instance of Q is a correct instance of Q.* □

Theorem 3.14 (Strong Completeness of LD-resolution) *Consider a program P and a query Q. For every correct instance Q' of Q there exists a computed instance Q'' of Q such that $Q'' \leq Q'$.* □

Let us now introduce the following notation. For a program P, a query Q and a set of queries $\mathcal{Q}$, we write

$$\{Q\}\ P\ \mathcal{Q}$$

to denote the fact that $\mathcal{Q}$ is the set of computed instances of Q. $\{Q\}\ P\ \mathcal{Q}$ should be read as: "the program P transforms Q into the set of its computed instances $\mathcal{Q}$". In particular, when $\mathcal{Q}$ is a singleton, say $\mathcal{Q} = \{Q'\}$, we have $\{Q\}\ P\ \{Q'\}$ which not accidentally coincides with the syntax of correctness formulas in Hoare style approach to verification of imperative programs (see, e.g., Apt and Olderog [11]). We now present an easy method of establishing constructs of the form $\{Q\}\ P\ \mathcal{Q}$.

Theorem 3.15 *Consider a program P and a query Q. Suppose that the set $\mathcal{Q}$ of ground correct instances of Q is finite. Then*

$$\{Q\}\ P\ \mathcal{Q}.$$

Proof. First note that

$$\text{every correct instance } Q' \text{ of } Q \text{ is ground.} \tag{3.1}$$

Indeed, otherwise, by the fact that the Herbrand universe is infinite, the set $\mathcal{Q}$ would be infinite.

Consider now $Q_1 \in \mathcal{Q}$. By the Strong Completeness Theorem 3.14, there exists a computed instance Q_2 of Q such that $Q_2 \leq Q_1$. By the

Soundness Theorem 3.13, Q_2 is a correct instance of Q, so by (3.1) Q_2 is ground. Consequently $Q_2 = Q_1$, that is Q_1 is a computed instance of Q.

Conversely, take a computed instance Q_1 of Q. By the Soundness Theorem 3.13, Q_1 is a correct instance of Q. By (3.1) Q_1 is ground, so $Q_1 \in \mathcal{Q}$. $\square$

For a query consisting of just one atom A the set of its ground correct instances equals $[A] \cap M_P$, so the assumption of the above theorem can be rephrased as "the set $[A] \cap M_P$ is finite". This simplifies checking its validity and explains the relevance of M_P in our approach. As the examples below indicate, the above theorem is quite useful.

Append

First consider the `APPEND-T` program and three of its uses.

(i) Given ground lists `s,t,u` we have

$$\texttt{app(s,t,u)} \in M_{\texttt{APPEND-T}} \text{ iff } \texttt{s} * \texttt{t} = \texttt{u}.$$

Consequently

- when `s*t = u`,

$$\{\texttt{app(s,t,u)}\}\ \texttt{APPEND} - \texttt{T}\ \{\texttt{app(s,t,u)}\};$$

- when `s*t` $\neq$ `u`,

$$\{\texttt{app(s,t,u)}\}\ \texttt{APPEND} - \texttt{T}\ \emptyset.$$

(ii) Given ground lists `s,t`, the set $[\texttt{app(s,t,Zs)}] \cap M_{\texttt{APPEND-T}}$ consists of just one element: `app(s,t,s*t)`. Thus

$$\{\texttt{app(s,t,Zs)}\}\ \texttt{APPEND} - \texttt{T}\ \{\texttt{app}(\texttt{s},\texttt{t},\texttt{s} * \texttt{t})\}.$$

(iii) Finally, given a ground list `u`, we have

$$[\texttt{app(Xs,Ys,u)}] \cap M_{\texttt{APPEND-T}} = \{\texttt{app(s,t,u)} \mid \texttt{s},\texttt{t} \text{ are g. lists, } \texttt{s} * \texttt{t} = \texttt{u}\}.$$

But each list can be split only in finitely many ways, so the set

$$[\texttt{app(Xs,Ys,u)}] \cap M_{\texttt{APPEND-T}}$$

is finite. Thus

$$\{\texttt{app(Xs,Ys,u)}\}\ \texttt{APPEND} - \texttt{T}\ \{\texttt{app(s,t,u)} \mid \texttt{s},\texttt{t} \text{ are g. lists, } \texttt{s} * \texttt{t} = \texttt{u}\}.$$

Palindrome

A slightly less trivial example is the PALINDROME-T program. Given a list s, let rev(s) denote its reverse. It is easy to check that

$$\begin{aligned} M_{\texttt{PALINDROME-T}} &= \{\texttt{palindrome(s)} \mid \texttt{s} \text{ is a g. list}, \texttt{rev(s) = s}\} \\ &\cup \{\texttt{reverse(s,t)} \mid \texttt{s,t} \text{ are g. lists}, \texttt{rev(s) = t}\} \\ &\cup \{\texttt{reverse(s,t,u)} \mid \texttt{s,t,u} \text{ are g.lists}, \texttt{rev(s)*t = u}\} \\ &\cup M_{\texttt{LIST}}, \end{aligned}$$

by noting that for lists x1s, x2s

$$\texttt{rev}([\texttt{x}|\texttt{x1s}]) * \texttt{x2s} = \texttt{rev}(\texttt{x1s}) * [\texttt{x}|\texttt{x2s}].$$

Thus for a ground list s

- when rev(s) = s,

$$\{\texttt{palindrome(s)}\}\ \texttt{PALINDROME}-\texttt{T}\ \{\texttt{palindrome(s)}\};$$

- when rev(s) $\neq$ s,

$$\{\texttt{palindrome(s)}\}\ \texttt{PALINDROME}-\texttt{T}\ \emptyset.$$

Moreover, for a ground list s, $[\texttt{reverse(s,Ys)}] \cap M_{\texttt{PALINDROME-T}} = \{\texttt{reverse(s,rev(s))}\}$, so

$$\{\texttt{reverse(s,Ys)}\}\ \texttt{PALINDROME}-\texttt{T}\ \{\texttt{reverse(s,rev(s))}\}.$$

Sequence

Finally, consider the SEQUENCE-T program. Call a list of 27 numbers satisfying the description of the sequence a *desired list*. We leave it to the reader to check that

$$\begin{aligned} M_{\texttt{SEQUENCE-T}} &= M_{\texttt{APPEND-T}} \\ &\cup \{\texttt{sublist(s,t)} \mid \texttt{s,t} \text{ are g. lists}, \texttt{s} \text{ is a sublist of } \texttt{t}\} \\ &\cup \{\texttt{sequence(s)} \mid \texttt{s} \text{ is a g. list of length } 27\} \\ &\cup \{\texttt{question(s)} \mid \texttt{s} \text{ is a desired list}\}. \end{aligned}$$

Thus $[\texttt{question(Ss)}] \cap M_{\texttt{SEQUENCE-T}} = \{\texttt{question(s)} \mid \texttt{s} \text{ is a desired list}\}$. But the number of desired lists is obviously finite (in fact, as we noted, there are 6 of them). Consequently,

$$\{\texttt{question(Ss)}\}\ \texttt{SEQUENCE}-\texttt{T}\ \{\texttt{question(s)} \mid \texttt{s} \text{ is a desired list}\}.$$

Clearly, the above approach to partial correctness cannot be used to reason about queries with "non-ground inputs" (or more precisely about queries with non-ground computed instances), like `app(s,t,Zs)` where `s,t` are non-ground lists, since $[\texttt{app(s,t,Zs)}] \cap M_{\texttt{APPEND\_T}}$ is infinite. Recently, Apt and Gabbrielli [6] proposed a modification of the above method which allows us to deal properly with such queries.

3.3 Occur-check Freedom

In this section we study the occur-check problem.

3.3.1 *Occur-check Free Programs*

To define this problem we need to recall the unification algorithm due to Martelli and Montanari [26]. Two atoms can unify only if they have the same relation symbol. With two atoms $p(s_1, ..., s_n)$ and $p(t_1, ..., t_n)$ to be unified we associate the set of equations

$$\{s_1 = t_1, ..., s_n = t_n\}.$$

In the sequel we often refer to this set as $p(s_1, ..., s_n) = p(t_1, ..., t_n)$. The algorithm operates on such finite sets of equations. We use below the notions of sets and of systems of equations interchangeably. A substitution θ such that $s_1\theta = t_1\theta, \ldots, s_n\theta = t_n\theta$ is called a *unifier* of $\{s_1 = t_1, ..., s_n = t_n\}$. Thus the set of equations $\{s_1 = t_1, ..., s_n = t_n\}$ has the same unifiers as the atoms $p(s_1, ..., s_n)$ and $p(t_1, ..., t_n)$.

Two sets of equations are called *equivalent* if they have the same set of unifiers, and a set of equations is called *solved* if it is of the form $\{x_1 = t_1, ..., x_n = t_n\}$, where the x_i's are distinct variables and none of them occurs in a term t_j. If $E = \{x_1 = t_1, ..., x_n = t_n\}$ is solved, then we call $\{x_1/t_1, ..., x_n/t_n\}$ the *unifier determined by* E.

To find a most general unifier (in short, *mgu*) of two atoms it suffices to transform the associated set of equations into an equivalent one which is solved. The following algorithm does it if this is possible and otherwise halts with failure.

MARTELLI-MONTANARI ALGORITHM

Nondeterministically choose from the set of equations an equation of a form below and perform the associated action:

(1) $f(s_1, ..., s_n) = f(t_1, ..., t_n)$	*replace it by the equations* $s_1 = t_1, ..., s_n = t_n$,
(2) $f(s_1, ..., s_n) = g(t_1, ..., t_m)$ where $f \not\equiv g$	*halt with failure*,
(3) $x = x$	*delete it*,
(4) $t = x$ where t is not a variable	*replace it by* $x = t$,

(5) $x = t$ where $x \not\equiv t$, x does not occur in t and x occurs elsewhere	*perform the substitution* $\{x/t\}$ *in every other equation,*
(6) $x = t$ where $x \not\equiv t$ and x occurs in t	*halt with failure.*

The algorithm terminates when no action can be performed or when failure arises. The following theorem holds (see Martelli and Montanari [26]).

Theorem 3.16 (Unification) *The Martelli-Montanari algorithm always terminates. If the original set of equations E has a unifier, then the algorithm successfully terminates and produces a solved set of equations determining an mgu of E, and otherwise it terminates with failure.* □

The test "x does not occur in t" in action (5) is called the *occur-check* and in most Prolog implementations omitted for reasons of efficiency. By omitting the occur-check in (5) and deleting action (6) from the Martelli-Montanari algorithm we are still left with two options depending on whether the substitution $\{x/t\}$ is performed in t itself. If it is, then divergence can result, because x occurs in t implies that x occurs in $t\{x/t\}$. If it is not, then an incorrect result can be produced, as in the case of the single equation $x = f(x)$ which yields the substitution $\{x/f(x)\}$. So in both cases the omission of the occur-check leads to complications. They are usually termed as the *occur-check problem.*

To deal with the occur-check problem we propose simple syntactic conditions which allow us to prove that for a given pure Prolog program and a query the occur-check can be safely omitted. To formally define this property we introduce the following notions.

Definition 3.17

- *A set of equations E is called not subject to occur-check (NSTO in short) if in no execution of the Martelli-Montanari algorithm started with E action (6) can be performed.*
- *Let ξ be an LD-derivation. Let A be an atom selected in ξ and H the head of the input clause selected to resolve A in ξ. Suppose that A and H have the same relation symbol. Then we say that the system $A = H$ is considered in ξ.*
- *Suppose that all systems of equations considered in the LD-derivations of $P \cup \{Q\}$ are NSTO. Then we say that $P \cup \{Q\}$ is occur-check free.* □

The concept of an NSTO set of equations is due to Deransart, Ferrand and Téguia [19] who studied the conditions under which the occur-check can be safely omitted independently of the selection rule and of the chosen resolution strategy . Note that for an NSTO set of equations it is irrelevant for the purposes of unification whether the occur-check is omitted from the Martelli-Montanari algorithm.

The above definition assumes a specific unification algorithm but allows us to derive precise results. Moreover, the nondeterminism built into the Martelli-Montanari algorithm allows us to model executions of various other unification algorithms. In contrast, no specific unification algorithm in the definition of the LD-derivation is assumed.

Since in the definition of the occur-check freedom *all* LD-derivations of $P \cup \{Q\}$ are considered, all systems of equations that can be considered in a possibly backtracking Prolog execution of a query Q w.r.t. the program P are taken into account.

We now present the approach of Apt and Pellegrini [10] for proving occur-check freedom. To this end we need some preparatory definitions. One of them is the notion of a mode.

3.3.2 *Well-moded Queries and Programs*

Intuitively, modes indicate how the arguments of a relation should be used. They were first considered in Mellish [29], and more extensively studied in Reddy [31] and Dembinski and Maluszynski [18].

Definition 3.18 *Consider an n-ary relation symbol p. By a mode for p we mean a function m_p from $\{1, \ldots, n\}$ to the set $\{+, -\}$. If $m_p(i) =$ "$+$", we call i an input position of p, and if $m_p(i) =$ "$-$", we call i an output position of p (both w.r.t. m_p). By a moding we mean a collection of modes, each for a different relation symbol.* □

We write m_p in a more suggestive form $p(m_p(1), \ldots, m_p(n))$. For example, `member(-,+)` denotes a binary relation symbol `member` with the first position moded as output and the second position moded as input.

The definition of moding assumes one mode per relation symbol in a program. Multiple modes may be obtained by simply renaming the relations. In the remainder of this section we assume that *every considered relation symbol* has a fixed mode associated with it. This assumption will allow us to talk about input positions and output positions of an atom.

We now introduce a restriction which constrains the "flow of data" through the query and through the clauses of the programs. To simplify the notation, when writing an atom as $p(\mathbf{u}, \mathbf{v})$, we now assume that $\mathbf{u}$ is a sequence of terms filling in the input positions of p and $\mathbf{v}$ is a sequence of terms filling in the output positions of p.

Definition 3.19

- *A query $p_1(\mathbf{s_1}, \mathbf{t_1}), \ldots, p_n(\mathbf{s_n}, \mathbf{t_n})$ is called well-moded if for $i \in [1, n]$*

$$Var(\mathbf{s_i}) \subseteq \bigcup_{j=1}^{i-1} Var(\mathbf{t_j}).$$

- *A clause*

$$p_0(\mathbf{t_0}, \mathbf{s_{n+1}}) \leftarrow p_1(\mathbf{s_1}, \mathbf{t_1}), \ldots, p_n(\mathbf{s_n}, \mathbf{t_n})$$

is called well-moded if for $i \in [1, n+1]$

$$Var(\mathbf{s_i}) \subseteq \bigcup_{j=0}^{i-1} Var(\mathbf{t_j}).$$

- *A program is called well-moded if every clause of it is.* □

In particular, an atomic query $p(\mathbf{s}, \mathbf{t})$ is well-moded if $Var(\mathbf{s}) = \emptyset$, and a unit clause $p(\mathbf{s}, \mathbf{t}) \leftarrow$ is well-moded if $Var(\mathbf{t}) \subseteq Var(\mathbf{s})$.

Thus, a query is well-moded if

- every variable occurring in an input position of an atom ($i \in [1, n]$) occurs in an output position of an earlier ($j \in [1, i-1]$) atom.

And a clause is well-moded if

- ($i \in [1, n]$) every variable occurring in an input position of a body atom occurs either in an input position of the head ($j = 0$), or in an output position of an earlier ($j \in [1, i-1]$) body atom,
- ($i = n+1$) every variable occurring in an output position of the head occurs in an input position of the head ($j = 0$), or in an output position of a body atom ($j \in [1, n]$).

Finally, we introduce the notion of linearity.

Definition 3.20

- *A family of terms is called linear if every variable occurs at most once in it.*
- *An atom is called input (resp. output) linear if the family of terms occurring in its input (resp. output) positions is linear.* □

Thus a family of terms is linear iff no variable has two distinct occurrences in any term and no two terms have a variable in common.

We now state a result allowing us to conclude that $P \cup \{Q\}$ is occur-check free. As we shall see, it can be easily applied to various pure Prolog programs.

Theorem 3.21 *Let P and Q be well-moded. Suppose that*

- *the head of every clause of P is output linear.*

Then $P \cup \{Q\}$ is occur-check free. □

Let us see now how this theorem can be applied to the programs considered in the previous sections.

Append

First, consider the program `APPEND` with the mode `app(+,+,-)`. It is easy to check that in this mode `APPEND` is well-moded and the head of every clause is output linear. By Theorem 3.21 we conclude that for `s` and `t` ground, `APPEND` $\cup$ {`app(s, t, u)`} is occur-check free.

Append, again

Also in the mode `app(-,-,+)` `APPEND` is well-moded and the head of every clause is output linear. Theorem 3.21 applies and yields that for `u` ground, `APPEND` $\cup$ {`app(s, t, u)`} is occur-check free.

Palindrome

Finally, consider the program `PALINDROME-T`. We mode it as follows: `palindrome(+)`, `reverse(+,-)`, `reverse(+,+,-)`, `list(+)`. Clearly, the program `PALINDROME-T` is then well-moded and the heads of all clauses are output linear, so by Theorem 3.21 for `s` ground, `PALINDROME-T` $\cup$ {`palindrome(s)`} is occur-check free.

3.3.3 *Nicely Moded Programs*

The above conclusions are still of a restrictive kind, because in each case we had to assume that the input positions of the one atom queries are ground. Moreover, Theorem 3.21 cannot be used to establish that `SEQUENCE-T` $\cup$ {`question(Ss)`} is occur-check free. Indeed, there is no way to mode this program and query so that both of them are well-moded.

To see this, first note that to get the query `question(Ss)` well-moded we have to use the mode `question(-)`. This implies that to get the clause defining the `question` relation well-moded, we have to use the mode `sequence(-)`. But then we cannot satisfy the requirement of well-modedness for the unit clause defining the `sequence` relation.

To deal with these difficulties we introduce the following notion due to Chadha and Plaisted [15] (and independently, though later, rediscovered in Apt and Pellegrini [10]).

Definition 3.22

- *A query* $p_1(\mathbf{s_1}, \mathbf{t_1}), \ldots, p_n(\mathbf{s_n}, \mathbf{t_n})$ *is called nicely moded if* $\mathbf{t_1}, \ldots, \mathbf{t_n}$ *is a linear family of terms and for* $i \in [1, n]$

$$Var(\mathbf{s_i}) \cap (\bigcup_{j=i}^{n} Var(\mathbf{t_j})) = \emptyset.$$

- *A clause*

$$p_0(\mathbf{s_0}, \mathbf{t_0}) \leftarrow p_1(\mathbf{s_1}, \mathbf{t_1}), \ldots, p_n(\mathbf{s_n}, \mathbf{t_n})$$

  is called nicely moded if $p_1(\mathbf{s_1}, \mathbf{t_1}), \ldots, p_n(\mathbf{s_n}, \mathbf{t_n})$ *is nicely moded and*

$$Var(\mathbf{s_0}) \cap (\bigcup_{j=1}^{n} Var(\mathbf{t_j})) = \emptyset.$$

  In particular, every unit clause is nicely moded.

- *A program is called nicely moded if every clause of it is.* □

Thus, assuming that in every atom the input positions occur first, a query is nicely moded if

- every variable occurring in an output position of an atom does not occur earlier in the query.

And a clause is nicely moded if

- every variable occurring in an output position of a body atom occurs neither earlier in the body nor in an input position of the head.

So, intuitively, the concept of being nicely moded prevents a "speculative binding" of the variables which occur in output positions — these variables are required to be "fresh". The following result of Apt and Pellegrini [10] and Chadha and Plaisted [15] clarifies the importance of this notion.

Theorem 3.23 *Let P and Q be nicely moded. Suppose that*

- *the head of every clause of P is input linear.*

Then $P \cup \{Q\}$ is occur-check free. □

Let us see now how this theorem can be applied to the previously studied programs.

Append

Consider again the program `APPEND` with the moding `app(+,+,-)`. Then `APPEND` is nicely moded and the head of every clause is input linear. By Theorem 3.23 we conclude that when `u` is linear and $Var(\mathtt{s},\mathtt{t}) \cap Var(\mathtt{u}) = \emptyset$, `APPEND` $\cup$ { `app(s, t, u)`} is occur-check free.

Append, again

With the moding `app(-,-,+)` `APPEND` is nicely moded as well, and the head of every clause is input linear. Again, by Theorem 3.23 we conclude that when `s,t` is a linear family of terms and $Var(\mathtt{s},\mathtt{t}) \cap Var(\mathtt{u}) = \emptyset$, `APPEND` $\cup$ { `app(s, t, u)`} is occur-check free.

Sequence

Reconsider now the program `SEQUENCE-T`. To be able to apply Theorem 3.23 we mode it as follows: `sublist(-,+)`, `sequence(+)`, `question(+)`, `app(-,-,+)`, `list(+)`. Thanks to the use of anonymous variables it is easy to check that then `SEQUENCE-T` is indeed nicely moded and that the heads of all clauses are input linear. By Theorem 3.23 we now get that when `t` is linear (and so, for example, a variable), `SEQUENCE-T` $\cup$ {`question(t)`} is occur-check free.

Palindrome

So far it seems that Theorem 3.23 allows us to draw more useful conclusions than Theorem 3.21. However, reconsider the program `PALINDROME-T`. In Chadha and Plaisted [15] it is shown that no moding exists such that `PALINDROME-T` is nicely moded and the heads of all clauses are input linear.

Thus Theorem 3.23 cannot be applied to this program whereas Theorem 3.21 was applicable.

The last two examples thus show that each of these theorems is applicable to different classes of programs.

4 Pure Prolog with Arithmetic

We now move on to the study of the second subset of Prolog, pure Prolog with arithmetic. The previous approach to termination can be readily applied to this subset – it suffices to use level mappings which assign to ground atoms with arithmetic relations the value 0.

However, some caution has to be exercised. While the base for our approach to termination, Theorem 3.3, remains valid for pure Prolog programs with arithmetic (in fact, the same proof carries through), Theorem 3.4 does not hold anymore. Indeed, consider the program with only one clause: $p \leftarrow\ x < y, p$. Because the LD-derivations which end in an error are finite, the above program is left terminating. However, it is easy to see that it is not acceptable – just consider the ground instance $p \leftarrow\ 1 < 2, p$ and recall from Section 2.2 that the clause $1 < 2$ is added to the program, so it is true in every model of it. (In contrast, the program consisting of the clause $p \leftarrow\ x < x, p$ *is* acceptable.) This shows that the proposed method of proving termination is somewhat less general in the case of programs with arithmetic.

We refer to Apt and Pedreschi [8] for a proof that `QUICKSORT` is left terminating and that for a list `s` all LD-derivations of `QUICKSORT` $\cup$ {`qs(s,Ys)`} are finite.

The subject of partial correctness is considered after studying the issue of errors.

4.1 Absence of Run-time Errors

To prove absence of errors we use types. We found it convenient to use here an approach recently proposed by Bronsard, Lakshman and Reddy [14] which from the semantic point of view coincides with the method of Bossi and Cocco [13] for proving partial correctness. In our presentation we abstract from the concrete syntax introduced in these papers. Bossi and Cocco [13] use first-order language and concentrate on proofs of partial correctness, whereas Bronsard, Lakshman and Reddy [14] introduce a language which allows us to express in a concise way recursive and polymorphic types which involve incomplete data structures. The idea is to associate with each relation symbol two types: a pre-type and a post-type.

We call an atom a *p-atom* if its relation symbol is p. Recall from Section 3.1 that we denoted by $rel(A)$ the relation symbol occurring in atom A. So an atom A is a $rel(A)$-atom.

The following very general definition of a type is sufficient for our purposes.

Definition 4.1 *Consider a relation symbol p.*

- *A type for p is a set of p-atoms closed under substitution.*
- *A type is a type for a relation symbol p.*
- *A directional type for p is a pair pre_p, $post_p$ of types for p. We call pre_p (resp. $post_p$) a pre-type (resp. a post-type) associated with p.* □

Below we shall often use certain sets of terms in the consider universal language:
$\mathcal{T}$ — the set of all terms,
List — the set of lists,
Gae — the set of of gae's,
ListGae — the set of lists of gae's.

In what follows we write a directional type for a relation symbol p in a more suggestive form used in Pedreschi [30], another recent work on directional types:

$$p : S \rightarrow T,$$

where $pre_p = \{p(\mathbf{s}) \mid \mathbf{s} \in S\}$ and $post_p = \{p(\mathbf{t}) \mid \mathbf{t} \in T\}$. For example ,

$$\texttt{app} : (List \times List \times \mathcal{T}) \cup (\mathcal{T} \times \mathcal{T} \times List) \rightarrow List \times List \times List$$

is a directional type for a ternary relation symbol `app`.

In the remainder of this section we assume that *every considered relation symbol* has a fixed directional type associated with it. This assumption will allow us to talk about pre- and post-types of a relation symbol.

Definition 4.2 *Given atoms $A_1, \ldots, A_n, A_{n+1}$ and types $T_1, \ldots, T_n, T_{n+1}$, where $n \geq 0$, we write*

$$\models A_1 \in T_1, \ldots, A_n \in T_n \;\Rightarrow\; A_{n+1} \in T_{n+1}$$

to denote the fact that for all substitutions θ, if $A_1\theta \in T_1, \ldots, A_n\theta \in T_n$, then $A_{n+1}\theta \in T_{n+1}$. □

We now abbreviate $A \in pre_{rel(A)}$ to $pre(A)$ and analogously for *post*.

Definition 4.3

- *A query $A_1, \ldots, A_n$ is called well-typed if for $j \in [1, n]$*

$$\models post(A_1), \ldots, post(A_{j-1}) \;\Rightarrow\; pre(A_j).$$

- *A clause $H \leftarrow B_1, \ldots, B_n$ is called well-typed if*

  for $j \in [1, n+1]$

$$\models pre(H), post(B_1), \ldots, post(B_{j-1}) \;\Rightarrow\; pre(B_j),$$

  where $pre(B_{n+1}) := post(H)$.

- *A program is called well-typed if every clause of it is.* □

In particular, an atomic query A is well-typed if $\models pre(A)$, and a unit clause $A \leftarrow$ is well-typed if $\models pre(A) \Rightarrow post(A)$.

The following property of the notion of being well-typed holds (essentially, see Bossi and Cocco [13] or an account of it in Apt and Marchiori [7]).

Lemma 4.4 (Persistence) *An LD-resolvent of a well-typed query, and a well-typed clause that is variable disjoint with it, is well-typed.* □

This brings us to the following important conclusion.

Corollary 4.5 *Let P and Q be well-typed, and let ξ be an LD-derivation of $P \cup \{Q\}$. Then $\models pre(A)$ for every atom A selected in ξ.*

Proof. A variant of a well-typed clause is well-typed and for a well-typed query $A_1, \ldots, A_n$ we have $\models pre(A_1)$. □

In the sequel, we say that an atom A *satisfies its precondition* if $\models pre(A)$.

Quicksort

To see the usefulness of this corollary let us return to the QUICKSORT program. To prove absence of run-time errors we start by typing the relation qs in a way reflecting the following statement: when the first argument is a list of gae's, upon successful termination the second argument is a list a gea's, so:

$$\texttt{qs} : ListGae \times \mathcal{T} \rightarrow \mathcal{T} \times ListGae,$$

and the built-in's $>$ and $\leq$ in such a way that the above corollary can be applied, so:

$$>: Gae \times Gae \rightarrow \mathcal{T} \times \mathcal{T},$$

and

$$\leq: Gae \times Gae \rightarrow \mathcal{T} \times \mathcal{T}.$$

We now complete the typing in such a way that QUICKSORT is well-typed:

$$\texttt{part} : Gae \times ListGae \times \mathcal{T} \times \mathcal{T} \rightarrow \mathcal{T} \times \mathcal{T} \times ListGae \times ListGae,$$

$$\texttt{app} : \mathcal{T} \times ListGae \times \mathcal{T} \rightarrow \mathcal{T} \times ListGae \times \mathcal{T}.$$

It is worthwhile to note that a trivial directional type, namely

$$\texttt{app} : \mathcal{T} \times \mathcal{T} \times \mathcal{T} \rightarrow \mathcal{T} \times \mathcal{T} \times \mathcal{T}$$

is sufficient here. The reason for using the above directional type will become clear in Section 6.1.

Assume now that s is a list of gae's. By Corollary 4.5 we conclude that all atoms selected in the LD-derivations of QUICKSORT $\cup$ $\{$qs(s,t)$\}$ satisfy their preconditions. In particular, when these atoms are of the form `u > v` or u $\leq$ v, both u and u are gae's. Thus the LD-derivations of QUICKSORT $\cup$ $\{$qs(s,t)$\}$ do not end in an error.

Length

The following program LENGTH uses another arithmetic relation, is:

```
length(Xs, N) ← N is the length of the list Xs.
length([_ | Ts], N) ← length(Ts, M), N is M+1.
length([], 0).
```

To prove absence of run-time errors we use the following types:

$$\texttt{length} : \mathcal{T} \times \mathcal{T} \rightarrow \mathcal{T} \times Gae,$$

$$\texttt{is} : \mathcal{T} \times Gae \rightarrow Gae \times \mathcal{T}.$$

It is easy to check that LENGTH is then well-typed. Corollary 4.5 now yields that for arbitrary terms s, t, all atoms selected in the LD-derivations of LENGTH $\cup$ $\{$length(s,t)$\}$ satisfy their preconditions. In particular, when these atoms are of the form `u is v`, v is a gae. So the LD-derivations of LENGTH $\cup$ $\{$length(s,t)$\}$ do not end in an error.

4.2 Partial Correctness

When dealing with partial correctness of programs that use arithmetic relations we need to remember (see Section 2.2) that to each program we added infinitely many clauses which define the used arithmetic relations. Both the Soundess Theorem 3.13 and the Strong Completeness Theorem 3.14 remain valid for programs with infinitely many clauses; however, completeness does not hold any more in the presence of arithmetic relations. Indeed, we have $P \models \texttt{X} < \texttt{Y}\{\texttt{X}/1, \texttt{Y}/2\}$ for any program P that uses $<$, whereas the LD-derivations of $P \cup \{\texttt{X} < \texttt{Y}\}$ end in an error. Also Theorem 3.15 does not hold then, as the query `X < 2` shows. Still, the following version of this theorem can be used for proofs of partial correctness.

Theorem 4.6 *Consider a program P and a query Q. Assume that the LD-derivations of $P \cup \{Q\}$ do not end in error. Suppose that the set $\mathcal{Q}$ of ground correct instances of Q is finite. Then*

$$\{Q\}\ P\ \mathcal{Q}.$$

Proof. Under the assumptions of the theorem both the Soundness Theorem 3.13 and the Strong Completeness Theorem 3.14 remain valid. For the completeness theorem this is not obvious, since it usually relies on the

Lifting Lemma which does not hold now. Indeed, the query `1 < 2` admits a successful LD-derivation, whereas all the LD-derivations of its more general version `X < Y` end in an error. However, the admirably short and elegant proof of Stärk [35] does not use the Lifting Lemma and carries through. Consequently, the proof of Theorem 3.15 carries through as well. □

Quicksort

To apply this theorem reconsider the `QUICKSORT` program. We deal here with its "correctly typed" version `QUICKSORT-T`, obtained by using `APPEND-T` instead of `APPEND` and in which the last clause defining the `part` relation is replaced by

$$\texttt{part(X, [], [], [])} \leftarrow \texttt{X} \leq \texttt{X.}$$

This forces the first argument of `part` to be a gae. (Without this change the query `qs([s],Ys)` would succeed for any `s`.)

Below we use the following terminology. An element `a` *partitions a list of gae's* `s` *into* `ls, bs` if `a` is a gae, `ls` is a list of elements of `s` which are $<$ `a` and `bs` is a list of elements of `s` which are $\geq$ `a`.

By extending the previously considered typing with

$$\texttt{list} : ListGae \rightarrow ListGae$$

we conclude that for a list of gae's `s` the LD-derivations of $\texttt{QUICKSORT-T} \cup \{\texttt{qs}(\texttt{s}, \texttt{Ys})\}$ do not end in an error. Moreover, the above-mentioned proof of termination of $\texttt{QUICKSORT} \cup \{\texttt{qs}(\texttt{s}, \texttt{Ys})\}$ can be modified in a straightforward way to the program `QUICKSORT-T`.

We leave it to the reader to check that

$$\begin{array}{rcl} M_{\texttt{QUICKSORT-T}} & = & M_{\texttt{APPEND-T}} \cup M_{>} \cup M_{\leq} \\ & \cup & \{\texttt{part}(\texttt{a}, \texttt{s}, \texttt{ls}, \texttt{bs}) \mid \texttt{s}, \texttt{ls}, \texttt{bs} \text{ are lists of gae's,} \\ & & \qquad \texttt{a} \text{ partitions } \texttt{s} \text{ into } \texttt{ls, bs}\} \\ & \cup & \{\texttt{qs}(\texttt{s}, \texttt{t}) \mid \texttt{s}, \texttt{t} \text{ are lists of gae's and} \\ & & \qquad \texttt{t} \text{ is a sorted permutation of } \texttt{s}\}. \end{array}$$

So for a list of gae's `s` the set $[\texttt{qs}(\texttt{s}, \texttt{Ys})] \cap M_{\texttt{QUICKSORT-T}}$ consists of just one element: `qs(s,t)`, where `t` is a sorted permutation of `s`. Consequently, by Theorem 4.6,

$$\{\texttt{qs}(\texttt{s}, \texttt{Ys})\}\ \texttt{QUICKSORT} - \texttt{T}\ \{\texttt{qs}(\texttt{s}, \texttt{t})\}.$$

Length

In contrast, the LENGTH program can be directly handled without any modification. It is easy to check that

$$\begin{array}{rcl} M_{\texttt{LENGTH}} & = & M_{\texttt{is}} \\ & \cup & \{\texttt{length}(\texttt{s}, |\texttt{s}|) \mid \texttt{s} \text{ is a g. list}\}. \end{array}$$

(Recall, that for a list s, |s| is its length.) Such a check involves the use of Lemma 3.11 which is applicable here, since the program LENGTH is easily seen to be acceptable, and so left terminating. So for a ground list s the set $[\texttt{length}(\texttt{s}, \texttt{N})] \cap M_{\texttt{LENGTH}}$ consists of just one element: length(s,|s|). By Theorem 4.6,

$$\{\texttt{length}(\texttt{s}, \texttt{N})\}\ \texttt{LENGTH}\ \{\texttt{length}(\texttt{s}, |\texttt{s}|)\}.$$

Note that the proof of the above claim for a non-ground list s breaks down because the set $[\texttt{length}(\texttt{s}, \texttt{N})] \cap M_{\texttt{LENGTH}}$ is then infinite.

4.3 Occur-check Freedom

Finally, we deal with the issue of the occur-check. The approach of Section 3.3 is applicable to pure Prolog programs with arithmetic without any modification. The reason is that the unit clauses which define the arithmetic relations are all ground, so they automatically satisfy the conditions of Theorems 3.21 and 3.23. To see how these results apply here reconsider the two running examples of this section.

Quicksort

Consider QUICKSORT with the moding qs(+,-), partition(+,+,-,-), app(+,+,-), >(+, +), ≤(+, +). QUICKSORT is then well-moded and the heads of all clauses are output linear. Theorem 3.21 applies and yields that for s ground, QUICKSORT ∪ {qs(s, t)} is occur-check free.

Moreover, in this moding QUICKSORT is also nicely moded and the head of every clause is input linear. Thus Theorem 3.23 applies as well, and yields that when t is linear and $Var(\texttt{s}) \cap Var(\texttt{t}) = \emptyset$, QUICKSORT ∪ { qs(s, t)} is occur-check free.

Length

Next, consider the LENGTH program with the moding length(+,-), is(-,+). Then LENGTH is well-moded and the heads of all clauses are output linear. By Theorem 3.21 for s ground, LENGTH ∪ {length(s, t)} is occur-check free.

Moreover, in this moding LENGTH is also nicely moded and the head of every clause is input linear. Thus Theorem 3.23 applies here as well, and yields that when t is linear and $Var(\texttt{s}) \cap Var(\texttt{t}) = \emptyset$, LENGTH ∪ { length(s, t)} is occur-check free. In particular, this conclusion holds for any list s and a variable N not appearing in s.

It is well-known that programs with difference-lists easily lead to complications in absence of the occur-check. For example, the program `empty`

```
empty(L \ L).
```

when executed with the goal $\leftarrow$ `empty([a | X] \ X)` leads to the consideration of the system { `[a | X] = L, X = L` } which is subject to the occur-check. It is worthwhile to note that programs which use difference-lists can be handled by the methods proposed. For example, Theorem 3.23 immediately implies that for `s` and `t` linear and variable disjoint, `empty` $\cup$ {`empty(s, t)`} is occur-check free.

However, more complex programs with difference lists like `quicksort_dl` (program 15.4 on page 244 in Sterling and Shapiro [36]) cannot be handled by the approach discussed here. In Apt and Pellegrini [10] a refinement of this approach is proposed which can be used to deal with such programs.

Of course, there exist programs whose executions for a natural class of queries do result in the occur-check problem. An example is the program that formalizes Curry's system of type assignment for the typed lambda calculus. For such a program and queries a transformation is proposed in Apt and Pellegrini [10] which transforms a program and a query into a program and a query for which only the calls to the built-in unification relation need to be resolved by a unification algorithm with the occur-check.

5 Pure Prolog with Negation

Finally, we deal with the third subset of Prolog, pure Prolog with negation. We call programs written in this subset general programs. Our approach to proving termination and partial correctness of general programs is applicable only under the assumption that floundering does not arise. So we have to deal with this issue first.

5.1 Absence of Floundering

To prove absence of floundering we generalize the notion of a well-moded program (Definition 3.19) to general programs. To this end we simply allow the negation symbol $\neg$ to occur in front of atoms in queries and clause bodies. More precisely, we introduce the following definition, where $\odot$ stands for $\neg$ or for the empty string.

Definition 5.1

- *A general query* $\odot p_1(\mathbf{s_1}, \mathbf{t_1}), \ldots, \odot p_n(\mathbf{s_n}, \mathbf{t_n})$ *is called well-moded if for* $i \in [1, n]$

$$Var(\mathbf{s_i}) \subseteq \bigcup_{j=1}^{i-1} Var(\mathbf{t_j}).$$

- *A general clause*

$$p_0(\mathbf{t_0}, \mathbf{s_{n+1}}) \leftarrow \odot\, p_1(\mathbf{s_1}, \mathbf{t_1}), \ldots, \odot p_n(\mathbf{s_n}, \mathbf{t_n})$$

is called well-moded if for $i \in [1, n+1]$

$$Var(\mathbf{s_i}) \subseteq \bigcup_{j=0}^{i-1} Var(\mathbf{t_j}).$$

- *A general program is called well-moded if every general clause of it is.* □

This definition will be useful later.

Definition 5.2 *A general program is called non-floundering if no LDNF-derivation starting in a ground general query flounders.* □

The following result is due to Apt and Pellegrini [10] and, independently, Stroetman [37].

Theorem 5.3 *Consider a well-moded general program* P *and a well-moded general query* Q*. Suppose that all relations used in negative literals of* P *and* Q *are moded completely input. Then* $P \cup \{Q\}$ *does not flounder. In particular,* P *is non-floundering.* □

To see the use of this theorem we now consider two general programs which deal with directed graphs. A directed graph is represented here as a (ground) list of its edges. In turn, an edge from node a to node b is represented by the list $[a, b]$.

Transitive Closure

The first general program, called `TRANS-T`, computes the transitive closure of a directed graph:

```
trans(X, Y, E, Avoids) ← list(Avoids), member([X, Y], E).
trans(X, Z, E, Avoids) ←
   member([X, Y], E),
   ¬ member(Y, Avoids),
   trans(Y, Z, E, [Y | Avoids]).

member(Element, List) ←Element is an element of the list List.
member(X, [Y | Xs]) ← member(X, Xs).
member(X, [X | Xs]) ← list(Xs).
```

augmented by the `LIST` program.

In a typical use of this program in order to check that `[x,y]` is in the transitive closure of the directed graph `e`, one evaluates the query `trans(x, y, e, [x])`.

With the moding `trans(-,-,+,+)`, `list(+)`, `member(+,+)` for the occurrence of `member` in the negative literal ¬ `member(Y, Avoids)`, and `member(-,+)` for the other occurrences of `member`, `TRANS-T` is well-moded. By Theorem 5.3, for `e,s` ground, `TRANS-T` ∪ {`trans(a, b, e, s)`} does not flounder. Moreover, `TRANS-T` is non-floundering.

Dag

Consider now the problem of testing whether a graph is a dag. Recall that *dag* is the abbreviation for "directed acyclic graph" and that a directed graph is *acyclic* if no path in it exists which forms a cycle. The solution is exceptionally simple, though not very efficient – we add to the general program TRANS-T the general clauses

```
acyclic(E) ← ¬ cyclic(E).
cyclic(E) ← trans(X, X, E, []).
```

Call the resulting general program DAG-T.

We now extend the above moding by `cyclic(+), acyclic(+)`. It is straightforward to check that DAG-T is then well-moded. Thus, by Theorem 5.3, for `e` ground, DAG-T $\cup$ {`acyclic(e)`} does not flounder. Moreover, DAG-T is non-floundering.

5.2 Termination

To deal with termination we use the approach of Apt and Pedreschi [8] which generalizes the method of Section 3.1 to general programs.

Definition 5.4 *A general program is called left terminating if all its LDNF-derivations starting with a ground query are finite.* □

Given a general program P, we now define its "negative part" P^-.

Definition 5.5 *Let P be a general program and p, q relations.*

- *p refers to q iff a general clause in P uses p in its head and q in its body.*
- *p depends on q is the reflexive, transitive closure of refers to.*
- *Neg_P is the set of relations which are used in a negative literal in P.*
- *Neg^*_P is the set of relations on which the relations in Neg_P depend.*
- *P^- is the set of general clauses in P in whose head a relation from Neg^*_P is used.* □

Recall now from Lloyd [25] and Apt [2] that $comp(P)$ stands for Clark's completion of a general program P.

Definition 5.6

- *Given a level mapping $|\ |$, we extend it to ground negative literals by putting $|\neg A| = |A|$. $\neg A$ is bounded with respect to $|\ |$ if A is.*
- *A general clause is called acceptable with respect to $|\,|$ and an interpretation I, if I is its model and for every ground instance $A \leftarrow \mathbf{K}, L, \mathbf{M}$ of it such that $I \models \mathbf{K}$*
$$|A| > |L|.$$
- *A general program P is called acceptable with respect to $|\ |$ and I, if every general clause of it is and if the restriction of I to the relation symbols from Neg^*_P is a model of $comp(P^-)$.* □

The following result relates these notions.

Theorem 5.7 *Let P be a general program acceptable w.r.t. $|\ |$ and I. Then for every literal L bounded w.r.t. $|\ |$ all LDNF-derivations of $P \cup \{L\}$ are finite. In particular, P is left terminating.* $\square$

So to apply the notion of acceptability we need a method for proving that an interpretation I is a model of $comp(P^-)$. For Herbrand interpretations the following observation due to Apt, Blair and Walker [4] comes to our rescue. The notion of a supported model is now extended to general programs in an obvious way.

Note 5.8 *A Herbrand interpretation I is a model of $comp(P)$ iff it is a supported model of P.* $\square$

The following result shows that the restriction to Herbrand models does not result in a limitation of the method.

Theorem 5.9 *Let P be a left terminating, non-floundering general program. Then, for some level mapping $|\ |$ and a Herbrand interpretation I,*

(i) P is acceptable w.r.t. $|\ |$ and I,
(ii) for every literal L all LDNF-derivations of $P \cup \{L\}$ are finite iff L is bounded w.r.t. $|\ |$. $\square$

Apt and Pedreschi [8] showed that `TRANS-T` is acceptable w.r.t. a level mapping $|\ |$ such that $|\mathtt{trans(a,b,e,s)}|$ is a function of `e` and `s`, and a Herbrand interpretation I. Thus for `e,s` ground all LDNF-derivations of `TRANS-T` $\cup$ {`trans(a, b, e, s)`} are finite. In particular, `TRANS-T` is left terminating.

By extending this level mapping to `DAG-T` with

$$
\begin{aligned}
|\mathtt{acyclic(e)}| &= |\mathtt{cyclic(e)}| + 1, \\
|\mathtt{cyclic(e)}| &= |\mathtt{trans(a,a,e,[])}| + 1,
\end{aligned}
$$

where `a` is a constant, and modifying I appropriately, we also conclude that for `e` ground all LDNF-derivations of `DAG-T` $\cup$ {`acyclic(e)`} are finite and that `DAG-T` is left terminating.

5.3 Partial Correctness

Our approach to partial correctness of general programs is applicable only to general programs which are left terminating and non-floundering. The following result of Apt and Pedreschi [8] is crucial.

Theorem 5.10 *Consider a left terminating, non-floundering general program P. Then,*

(i) P has a unique supported Herbrand model, M_P,

(ii) M_P *is a model of* $comp(P)$,

(iii) *for a ground general query* Q *such that* $P \cup \{Q\}$ *does not flounder,* $M_P \models Q$ *iff there exists a successful LDNF-derivation of* $P \cup \{Q\}$.

□

We now need to revise Definition 3.12.

Definition 5.11 *Consider a general program* P *and a general query* Q. *We say that* Q' *is a correct instance of* Q, *if* Q' *is an instance of* Q *and* $comp(P) \models Q'$. □

The definition of a computed instance refers now to the LDNF-resolution. The following soundness and completeness results are of help.

Theorem 5.12 (Soundness of LDNF-resolution) *Consider a general program* P *and a general query* Q. *Every computed instance of* Q *is a correct instance of* Q. □

Theorem 5.13 (Limited Completeness of LDNF-resolution) *Consider a left terminating, non-floundering general program* P *and a general query* Q *such that* $P \cup \{Q\}$ *does not flounder. For every ground correct instance* Q' *of* Q *there exists a computed instance* Q'' *of* Q *such that* $Q'' \leq Q'$.

Proof. $P \cup \{Q'\}$ does not flounder since $P \cup \{Q\}$ does not flounder. By Theorem 5.10(ii), (iii) there exists a successful LDNF-derivation of $P \cup \{Q'\}$. $P \cup \{Q\}$ does not flounder, so we can lift this derivation to a successful LDNF-derivation of $P \cup \{Q\}$ which yields a computed instance Q'' of Q such that $Q'' \leq Q'$. □

These theorems are needed to establish the following result.

Theorem 5.14 *Consider a left terminating, non-floundering general program* P *and a general query* Q *such that* $P \cup \{Q\}$ *does not flounder. Suppose that the set* $\mathcal{Q}$ *of ground correct instances of* Q *is finite. Then*

$$\{Q\}\ P\ \mathcal{Q}.$$

Proof. The proof is analogous to the proof of Theorem 3.15. So first we note that

$$\text{every correct instance } Q' \text{ of } Q \text{ is ground.} \tag{5.1}$$

Consider now $Q_1 \in \mathcal{Q}$. By the Limited Completeness Theorem 5.13, there exists a computed instance Q_2 of Q such that $Q_2 \leq Q_1$. By the Soundness Theorem 5.12, Q_2 is a correct instance of Q, so by (5.1) Q_2 is ground. Consequently, $Q_2 = Q_1$; that is, Q_1 is a computed instance of Q.

Conversely, take a computed instance Q_1 of Q. By the Soundness Theorem 5.12, Q_1 is a correct instance of Q. By (5.1) Q_1 is ground, so $Q_1 \in \mathcal{Q}$.

□

To apply this theorem we need a method to establish the premise "the set $\mathcal{Q}$ of ground correct instances of Q is finite". As in the case of pure Prolog programs, we solve this problem by restricting our attention to the model M_P. Indeed, for an atomic query A the above premise can be rephrased (thanks to Theorems 5.10 and 5.12) as "the set $[A] \cap M_P$ is finite".

As in the case of pure Prolog programs, it is usually straightforward to check that a Herbrand interpretation is a supported model of a general program. So in the examples below we omit the proofs of these facts.

Transitive Closure

We now show how to apply this theorem to the program `TRANS-T`. In the previous two subsections we proved that `TRANS-T` is left terminating and non-floundering. Adopt the following terminology. Given a list e, a *path in e from a to b* is a sequence $a_1, \ldots, a_n$ $(n > 1)$ such that

- $[a_i, a_{i+1}] \in e$ for $i \in [1, n-1]$,
- $a_1 = a$,
- $a_n = b$.

An *interior* of a path $a_1, \ldots, a_n$ $(n > 1)$ is the set $\{a_2, \ldots, a_{n-1}\}$. A path $a_1, \ldots, a_n$ $(n > 1)$ is called *acyclic* if the elements of its interior are pairwise different. A path $a_1, \ldots, a_n$ $(n > 1)$ *avoids* a list `s` if no element of its interior is a member of `s`. In particular, a path consisting of two elements has an empty interior and consequently is acyclic and avoids every `s`.

It is routine to check that

$$\begin{aligned} M_{\mathtt{TRANS-T}} &= M_{\mathtt{LIST}} \\ &\cup \{\mathtt{trans(a, b, e, s)} \mid \mathtt{e, s} \text{ are g. lists, an acyclic path in } \mathtt{e} \\ &\qquad\qquad \text{from } \mathtt{a} \text{ to } \mathtt{b} \text{ exists which avoids } \mathtt{s}\} \\ &\cup \{\mathtt{member(a, t)} \mid \mathtt{t} \text{ is a g. list and } \mathtt{a} \in \mathtt{t}\}. \end{aligned}$$

Consider now a directed graph `e`. We denote its transitive closure by $\mathtt{e}^*$. Then $\mathtt{[a,b]} \in \mathtt{e}^*$ iff there exists in `e` an acyclic path from `a` to `b` which avoids `[a]`. By Theorem 5.14 we conclude that

- when $\mathtt{[a,b]} \in \mathtt{e}^*$,

$$\{\mathtt{trans(a, b, e, [a])}\}\ \mathtt{TRANS-T}\ \{\mathtt{trans(a, b, e, [a])}\};$$

- when $\mathtt{[a,b]} \notin \mathtt{e}^*$,

$$\{\mathtt{trans(a, b, e, [a])}\}\ \mathtt{TRANS-T}\ \emptyset.$$

Note that `[a]` can be replaced here by `[]` or by `[a,b]`.

Moreover,

$$[\mathtt{trans(X, Y, e, [\,])}] \cap M_{\mathtt{TRANS-T}} = \{\mathtt{trans(a, b, e, [\,])} \mid \mathtt{[a, b]} \in \mathtt{e}^*\},$$

so

$$\{\texttt{trans(X,Y,e,[])}\}\ \texttt{TRANS}-\texttt{T}\ \{\texttt{trans(a,b,e,[])} \mid \texttt{[a,b]} \in \texttt{e}^*\},$$

since TRANS-T $\cup$ {trans(X,Y,e,[])} does not flounder. This, in conjunction with the fact that all LDNF-derivations of TRANS-T $\cup$ {trans(X,Y,e,[])} are finite, implies that the query trans(X, Y, e, []) generates all pairs of elements which form the nodes of the transitive closure e*.

Dag

To deal with the general program DAG-T we extend the above terminology. Given a list e, we call e *cyclic* if for some a a path in e from a to a exists, and we call e *acyclic* if it is not cyclic. We leave it to the reader to check that

$$\begin{array}{rcl} M_{\texttt{DAG-T}} & = & M_{\texttt{TRANS-T}} \\ & \cup & \{\texttt{acyclic(e)} \mid \texttt{e} \text{ is a ground acyclic list}\} \\ & \cup & \{\texttt{cyclic(e)} \mid \texttt{e} \text{ is a ground cyclic list}\}. \end{array}$$

Now take a directed graph e. By Theorem 5.14 we conclude that:

- when e is acyclic, {acyclic(e)} DAG $-$ T {acyclic(e)};
- when e is cyclic, {acyclic(e)} DAG $-$ T $\emptyset$.

5.4 Occur-check Freedom

When considering the notion of the occur-check freedom for general programs and general queries, we simply reuse the original Definition 3.17 but now apply it to the LDNF-derivations. In this way, we ignore the selection of negative literals, but this does not matter as the choice of a negative literal $\neg A$ either leads to floundering or to the consideration of the query A whose selected literal is positive. In both cases no unification is performed.

Further, we reuse the notion of well-moded general programs and general queries (Definition 5.1) introduced in Section 5.1. Theorem 3.21 easily generalizes to general programs and general queries. More precisely, we have the following result (see Apt and Pellegrini [10]).

Theorem 5.15 *Let P be a general well-moded program and Q a general well-moded query. Suppose that*

- *the head of every general clause of P is output linear.*

Then $P \cup \{Q\}$ is occur-check free. □

Transitive Closure

Let us see now how this result can be applied to TRANS-T. In Section 5.1 we had to introduce two modes for the member relation. Here a simpler moding suffices, namely trans(-,-,+,+), list(+), member(-,+). Then trans is well-moded and the heads of all general clauses are output linear.

So we conclude by Theorem 5.15 that for `e,v` ground, `TRANS-T` $\cup$ {`trans(s, t, e, v)`} is occur-check free.

Dag

Extending the above moding by `cyclic(+), acyclic(+)` we can also draw appropriate conclusions for the general program `DAG-T`: by Theorem 5.15 for `e` ground, `DAG-T` $\cup$ {`acyclic(e)`} is occur-check free.

It is also possible to generalize the result on nicely moded programs (viz. Theorem 3.23) to the case of general programs. However, the concept of a nicely moded general program does not prevent the use of non-ground input positions in the queries. As a result general programs to which the results on nicely moded general programs can be applied usually flounder. So — in the framework of LDNF-resolution — this generalization is of limited interest and consequently is omitted.

6 Conclusions

6.1 Dealing with "Ill-typed" Programs

In our analysis we only dealt with the "correctly typed" programs, i.e. programs named `XXX-T`. These programs are easier to handle than their corresponding "ill-typed" `XXX` versions, but they are much more inefficient due to the added "type checks".

It is possible to deal directly with the "ill-typed" programs, but the study of their partial correctness is quite a nuisance, because it is awkward to describe their unique supported Herbrand models in simple and intuitive terms.

Therefore we propose the following alternative, which we illustrate on the program `QUICKSORT`. Consider the typing of `QUICKSORT` defined in Section 4.1. Let `qs(s,t)` be a well-typed query and let ξ be an LD-derivation of `QUICKSORT` $\cup$ $\{\mathtt{qs(s,t)}\}$. By Corollary 4.5, if the selected atom is of the form $\mathtt{part}(\mathtt{s}_1, \mathtt{s}_2, \mathtt{s}_3, \mathtt{s}_4)$ then $\mathtt{s}_1 \in Gae$, and if the selected atom is of the form $\mathtt{app}(\mathtt{s}_1, \mathtt{s}_2, \mathtt{s}_3)$ then $\mathtt{s}_2 \in List$.

Thus in both cases in the corresponding LD-derivation of `QUICKSORT-T` $\cup$ $\{\mathtt{qs(s,t)}\}$ the inserted "type checks", namely `X` $\leq$ `X` and `list(Y)`, succeed with the empty computed answer substitution. Consequently, the computed instances of the query `qs(s,t)` are the same w.r.t. both programs. In particular, for a list of gae's `s` we have

$$\{\mathtt{qs(s,Ys)}\}\ \mathtt{QUICKSORT}\ \{\mathtt{qs(s,t)}\}.$$

The same approach can be applied to other pure Prolog programs and programs with arithmetic.

For general programs we need to extend Definition 4.3. This can be done by simply identifying $pre(\neg A)$ with $pre(A)$ and $post(\neg A)$ with $post(A)$. Then the generalization of Corollary 4.5 to LDNF-derivations holds, so the above technique is also applicable to general programs, in particular to `TRANS-T` and `DAG-T`.

6.2 Final Remarks

The aim of this chapter was to show that it is possible to reason about correctness of various Prolog programs by means of simple arguments based on syntactic analysis, declarative semantics, modes and types. We hope that this work can form a basis for a similar study of other languages based on the logic programming paradigm. In particular, it would be interesting to carry out such a study for logic programs executed with a dynamic selection rule defined by means of delay declarations. Such dynamic selection rules are for example present in Gödel, a declarative language designed by Hill and Lloyd [20].

In general, all correctness properties studied in this chapter are undecidable. However, certain aspects of the approach discussed here can be automated. We conclude this chapter by discussing this point in some detail and pointing out which issues require further investigation.

6.2.1 *Termination*

The approach to termination discussed here is based on the use of the notion of acceptability. Apt and Pedreschi [8] noted that some fragments of the proof of accceptability can be automated. In fact, they indicated that in many cases the task of checking the guesses for both the level mapping $|\ |$ and the model I can be reduced to checking the validity of universal formulas in an extension of Presburger arithmetic by the *min* and *max* operators. The validity problem for such formulas is decidable. In fact, Shostak [34] presented for this class a decision algorithm which is exponential. Cinzia Pieramico of the University of Pisa implemented this procedure for checking left termination w.r.t. a level mapping and a Herbrand interpretation which are expressible in the above language and verified mechanically that the quicksort program `QS` is left terminating.

De Schreye, Verschaetse and Bruynooghe [33] studied the problem of automatic generation of level mappings and Herbrand interpretations w.r.t. which the program is left terminating.

6.2.2 *Partial Correctness*

The approach to partial correctness reported in this chapter is to our knowledge new and its (partial) automation needs to be further studied. It is worthwhile to point out here that Theorem 5.10 implies that for left terminating (non-floundering general) programs the membership problem for the model M_P is decidable. So given such a (general) program, it is decidable whether a ground (general) query successfully terminates.

However, the complexity of this decision problem is in general forbiddingly high because the results of Bezem [12] imply that every total recursive function can be encoded in a model M_P.

6.2.3 *Occur-check Freedom*

The methods proposed here can be trivially implemented because they are based on syntactic analysis. However, to use Theorem 3.21 it is necessary to generate modings for which this theorem can be applied. To this end efficient algorithms are needed for generating modings for which a program is well-moded. A test as to whether a query or clause is well-moded w.r.t. a given moding can be efficiently performed by noting that:

- a query Q is well-moded iff every first from the left occurrence of a variable in Q is within an output position;
- a clause $p(\mathbf{s}, \mathbf{t}) \leftarrow \mathbf{B}$ is well-moded iff every first from the left occurrence of a variable in the sequence $\mathbf{s}, \mathbf{B}, \mathbf{t}$ is within the input position of $p(\mathbf{s}, \mathbf{t})$ or within an output position in $\mathbf{B}$.

(We assumed in this description that in every atom the input positions occur first.)

As already mentioned, the concepts of nicely moded program and query and Theorem 3.23 were also introduced in Chadha and Plaisted [15]. They proposed two efficient algorithms for generating modings with the minimal number of input positions, for which the program is nicely moded. These algorithms were implemented and applied to a number of well-known Prolog programs.

6.2.4 *Absence of Errors*

Our approach to proving absence of errors is based on Corollary 4.5. To apply it one needs to generate typings which include $>: Gae \times Gae \rightarrow \mathcal{T} \times \mathcal{T}$ for which a given program is well-typed. Aiken and Lakshman [1] showed that the problem of whether a program or query is well-typed w.r.t. a given typing is decidable for a large class of types which includes the ones studied here.

6.2.5 *Absence of Floundering*

Our method of proving absence of floundering is based on the use of the notion of well-modedness, already discussed in the context of the occur-check freedom.

Acknowledgements

Joint research and discussions with Dino Pedreschi on the subject of verification of logic programs helped us to clarify the opinions expressed in this chapter. Also, we thank the referee for helpful comments.

Bibliography

1. A. Aiken and T. K. Lakshman. Automatic type checking for logic programs. Technical report, Department of Computer Science, University of Illinois at Urbana Champaign, 1993.

2. K. R. Apt. Logic programming. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, pages 493–574. Elsevier, 1990. Vol. B.
3. K. R. Apt. Declarative programming in Prolog. In D. Miller, editor, *Proc. International Symposium on Logic Programming*, pages 11–35. MIT Press, 1993.
4. K. R. Apt, H. A. Blair, and A. Walker. Towards a theory of declarative knowledge. In J. Minker, editor, *Foundations of Deductive Databases and Logic Programming*, pages 89–148. Morgan Kaufmann, 1988.
5. K. R. Apt and H. C. Doets. A new definition of SLDNF-resolution. *Journal of Logic Programming*, 18(2):177–190, 1994.
6. K. R. Apt and M. Gabbrielli. Declarative interpretations reconsidered. In P. van Hentenryck, editor, *Proceedings of the 1994 International Conference on Logic Programming*, pages 74–89. MIT Press, 1994.
7. K. R. Apt and E. Marchiori. Reasoning about Prolog programs: from modes through types to assertions. Technical Report CS-R9358, CWI, Amsterdam, The Netherlands, 1993. To appear in Formal Aspects of Computing (FACS).
8. K. R. Apt and D. Pedreschi. Reasoning about termination of pure Prolog programs. *Information and Computation*, 106(1):109–157, 1993.
9. K. R. Apt and D. Pedreschi. Modular termination proofs for logic and pure Prolog programs. In G. Levi, editor, *Advances in logic programming theory*. Oxford University Press, 1994. To appear.
10. K. R. Apt and A. Pellegrini. On the occur-check free Prolog programs. *ACM Toplas*, 16(3):687–726, 1994.
11. K.R. Apt and E.-R. Olderog. *Verification of Sequential and Concurrent Programs.* Texts and Monographs in Computer Science, Springer-Verlag, New York, 1991.
12. M. A. Bezem. Strong termination of logic programs. *Journal of Logic Programming*, 15(1 & 2):79–98, 1993.
13. A. Bossi and N. Cocco. Verifying correctness of logic programs. In *Proceedings of TAPSOFT '89*, Lecture Notes in Computer Science, pages 96–110. Springer-Verlag, 1989.
14. F. Bronsard, T. K. Lakshman, and U. S. Reddy. A directional type system for Prolog: unifying the notions of types and directionality. Technical report, Department of Computer Science, University of Illinois at Urbana Champaign, 1993.
15. R. Chadha and D. A. Plaisted. Correctness of unification without occur check in Prolog. *Journal of Logic Programming*, 18(2):99–122, 1994.
16. K. L. Clark. Predicate logic as a computational formalism. Res. Report

DOC 79/59, Imperial College, Dept. of Computing, London, 1979.

17. H. Coelho and J. C. Cotta. *Prolog by Example.* Springer-Verlag, Berlin, 1988.
18. P. Dembinski and J. Maluszynski. AND-parallelism with intelligent backtracking for annotated logic programs. In *Proceedings of the International Symposium on Logic Programming*, pages 29–38, Boston, 1985.
19. P. Deransart, G. Ferrand, and M. Téguia. NSTO programs (not subject to occur-check). In V. Saraswat and K. Ueda, editors, *Proceedings of the International Logic Symposium*, pages 533–547. The MIT Press, 1991.
20. P. M. Hill and J. W. Lloyd. *The Gödel Programming Language.* The MIT Press, 1994.
21. M. Kalsbeek. The vanilla meta-interpreter for definite logic programs and ambivalent syntax. Technical Report CT-93-01, Department of Mathematics and Computer Science, University of Amsterdam, The Netherlands, 1993.
22. K. Kunen. Some remarks on the completed database. In R. A. Kowalski and K. A. Bowen, editors, *Proceedings of the Fifth International Conference on Logic Programming*, pages 978–992. The MIT Press, 1988.
23. K. Kunen. Signed data depedencies in logic programs. *Journal of Logic Programming*, 7:231–246, 1989.
24. J. W. Lloyd. *Foundations of Logic Programming.* Springer-Verlag, Berlin, 1984.
25. J. W. Lloyd. *Foundations of Logic Programming.* Springer-Verlag, Berlin, second edition, 1987.
26. A. Martelli and U. Montanari. An efficient unification algorithm. *ACM Transactions on Programming Languages and Systems*, 4:258–282, 1982.
27. M. Martelli and C. Tricomi. A new SLDNF-tree. *Information Processing Letters*, 43(2):57–62, 1992.
28. B. Martens and D. De Schreye. Why untyped non-ground meta-programming is not much of a problem. Technical Report CW 159 (Revised November 1993), Department of Computing Science, Katholieke Universiteit Leuven, Belgium, 1993. To appear in Journal of Logic Programming.
29. C. S. Mellish. The Automatic Generation of Mode Declarations for Prolog Programs. DAI Research Paper 163, Department of Artificial Intelligence, Univ. of Edinburgh, August 1981.
30. D. Pedreschi. A proof method for run-time properties of Prolog pro-

grams. In P. van Hentenryck, editor, *Proceedings of the 1994 International Conference on Logic Programming.* MIT Press, 1994. To appear.

31. U. S. Reddy. Transformation of logic programs into functional programs. In *International Symposium on Logic Programming*, pages 187–198, Silver Spring, MD, February 1984. Atlantic City, IEEE Computer Society.
32. D. De Schreye and S. Decorte. Termination of logic programs: the never-ending story. *Journal of Logic Programming*, 19-20:199–260, 1994.
33. D. De Schreye, K. Verschaetse, and M. Bruynooghe. A framework for analyzing the termination of definite logic programs with respect to call patterns. In *Proceedings of the International Conference on Fifth Generation Computer Systems 1992*, pages 481–488. Institute for New Generation Computer Technology, 1992.
34. R. E. Shostak. On the SUP-INF method for proving Presburger formulas. *J. ACM*, 24(4):529–543, 1977.
35. R. Stärk. A direct proof for the completeness of SLD-resolution. In Börger, H. Kleine Büning, and M.M. Richter, editors, *Computer Science Logic 89*, Lecture Notes in Computer Science 440, pages 382–383. Springer-Verlag, 1990.
36. L. Sterling and E. Shapiro. *The Art of Prolog.* MIT Press, 1986.
37. K. Stroetman. A completeness result for SLDNF resolution. *The Journal of Logic Programming*, 15:337–357, 1993.
38. J. D. Ullman and A. van Gelder. Efficient tests for top-down termination of logical rules. *J. ACM*, 35(2):345–373, 1988.