Declarative Programming in Prolog

Krzysztof R. Apt
CWI
P.O. Box 94079, 1090 GB Amsterdam, The Netherlands
and
Faculty of Mathematics and Computer Science
University of Amsterdam, Plantage Muidergracht 24
1018 TV Amsterdam, The Netherlands

Abstract

We try to assess to what extent declarative programming can be realized in Prolog and which aspects of correctness of Prolog programs can be dealt with by means of declarative interpretation.

More specifically, we discuss termination of Prolog programs, partial correctness, absence of errors and the safe use of negation.

1 Introduction

1.1 Motivation

Verification of Prolog programs has been an ongoing research endeavour since the beginning of logic programming. Already Clark and Tarnlund [CT77], and more extensively, Clark [Cla79] addressed this issue. Hogger [Hog84] dealt with this subject in his book, Deransart [Der90] compared various approaches to partial correctness, and Deville [Dev90] studied systematic development of logic and Prolog programs from specifications.

In the case of other styles of programming analogous research resulted in clearly isolated and widely recognized proof principles and design methods, which can be readily used when dealing with specific programs or programming problems (see e.g. Dijkstra [Dij76] and Gries [Gri81] for sequential imperative programming, Chandy and Misra [CM88], Apt and Olderog [AO91], and Manna and Pnueli [MP92] for concurrent imperative programming; and Burstall and Darlington [BD77], Meertens [Mee86] and Bird and Wadler [BW88] for functional programming).

Regrettably, such development did not take place in the case of logic programming. Among the reasons might be two often repeated claims. According to one of them, a well-written Prolog program is "obviously correct" because it can be viewed as a self-evident specification of the problem under consideration. According to another, any correctness proof of a program

will be so obscure that its validity will be less convincing than that of the program itself.

We strongly disagree with these statements and find that their widespread popularity is one of the causes why programming in logic programming is not yet considered as a viable and attractive alternative to programming in other styles.

Of course, from the programming point of view, the main interest in logic programming is due to its capability to support declarative programming. Loosely speaking, declarative programming can be described as follows. *Specifications*, when written in an appropriate format, can be used as a *program*. Then the desired conclusions should *logically* follow from the program. To compute these conclusions some *computation mechanism* should be used.

Clearly, logic programming comes close to this description. The soundness and completeness results relate the declarative and procedural interpretations and consequently the concepts of correct answer substitutions and computed answer substitutions. However, these substitutions do not need to coincide, so a mismatch may arise. Additional complications result from adding negation.

When moving from logic programming to Prolog new difficulties arise due to the use of depth-first search strategy combined with the ordering of the clauses, the fixed selection rule, the omission of the occur-check in the unification, and the use of built-in's and various "non-logical" features.

If we wish to consider declarative programming in Prolog seriously, we should identify the programs whose correctness can be established by means of *simple* methods based on declarative semantics. This is the aim of this paper.

1.2 Terminology and Notation

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. 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 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. Throughout the paper 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.

Apart from this we use the standard notation of Lloyd [Llo87] and Apt [Apt90]. 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.

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.

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.

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 paper 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 constants, function and relation symbols are the ones occurring in the program (see e.g. Lloyd [Llo87] and Apt [Apt90]). Another choice was made by Kunen [Kun89] who assumed a first-order language with infinitely many constants, function and relation symbols in which all programs and queries are written. In this paper 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.

2.2 Proof Theory

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

Pure Prolog

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

with this issue. Due to the lack of space we refer the reader to Apt and Pellegrini [AP92] whose work builds upon Deransart, Ferrand and Téguia [DFT91] and whose methods based on *syntactic* analysis can be applied to all programs here considered.

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.

Pure Prolog with Arithmetic

Consider the program QUICKSORT:

```
qs([X | Xs], Ys) ←
   part(X, Xs, Littles, Bigs),
   qs(Littles, Ls),
   qs(Bigs, Bs),
   app(Ls, [X | Bs], Ys).
qs([], []).

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(X, [], [], []).
```

augmented by the APPEND program defined by:

```
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.

Consequently, when interpreting the arithmetic relations we follow Prolog's interpretation, according to which, as just stated, when at the moment

of evaluation the arguments of the comparison relations are not gae's, the computation *ends in an error*. Also, the assignment `s is t` ends in an error when at the moment of evaluation `t` is not a gae.

To model this interpretation of arithmetic relations we follow Kunen [Kun88]. First we extend the LD-resolution by stipulating that an LD-derivation *ends in an error* precisely in the cases stated above. 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 $<$ we add the following set of unit clauses:

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

for "`is`" we add the set

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

etc. Note that thanks to the "ending in an error" provision the resulting LD-trees remain finitely branching. In fact, every query with a selected atom the relation of which is an arithmetic one has at most one descendant in every LD-tree.

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 [Llo87] needs to be modified.

We leave to the reader the task of checking that according to the definition of SLDNF-resolution given in Clark [Cla79] and reproduced in Lloyd [Llo84] 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 [Llo87] 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 [MT92]. 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 [AD92], 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.

2.3 Semantics

There is no universal agreement what is the declarative semantics of a logic program. In this paper 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 ground lists whose all elements are equal to []. Call such lists *trivial*. It is easy to see that then

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

where "* " denotes the operation of concatenating two lists. This is the semantics of the APPEND program given in Sterling and Shapiro [SS86]. Clearly it cannot be used to render the meaning of queries in which other constants and functions than [] and [. | .] are used.

As soon as the underlying first-order language has another constant than [], 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([H | Ts]) ← list(Ts).
list([]).
```

Note that

$$M_{\texttt{APPEND-T}} = \{\texttt{app(s,t,u)} \mid \texttt{s,t,u are ground lists and s} * \texttt{t} = \texttt{u}\} \cup M_{\texttt{LIST}},$$

where

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

We shall return to the original program APPEND in Subsection 6.1. Discussion of the semantics of the other two fragments of Prolog is postponed till Subsections 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 deal with termination. We present here the approach of Apt and Pedreschi [AP93] which makes use of the declarative semantics. For simplicity we restrict out attention to queries which consist of single atom. 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 $|\ |\ : B_P \to N$ 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 $[A]$. 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* 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 every clause of it is. □

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 an interpretation I of P*

(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 we can only handle 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 irrelevant 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.

Example

To see how this method can be applied considered the following problem from Coelho and Cotta [CC88, 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(Ys, XsYsZs) ← app(Xs,YsZs,XsYsZs), app(Ys,Zs,YsZs).

sequence([_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_]).

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. Consider the following function $|\ |$ from ground terms to natural numbers:

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

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

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

$$\begin{aligned} |\mathtt{question(xs)}| &= 57, \\ |\mathtt{sequence(xs)}| &= 0, \\ |\mathtt{sublist(xs,ys)}| &= |\mathtt{xs}| + |\mathtt{ys}| + 2, \\ |\mathtt{app(xs,ys,zs)}| &= \min(|\mathtt{xs}|, |\mathtt{zs}|) + 1, \\ |\mathtt{list(xs)}| &= 0, \end{aligned}$$

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

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

Also, the query question(Ss) is bounded w.r.t. $|\ |$ and consequently all LD-derivations of SEQUENCE-T $\cup$ {question(Ss)} are finite.

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 [AP93] explains why for a left terminating program it is easier to verify that a Herbrand interpretation is its least Herbrand model.

Definition 3.5 An interpretation I for a program P is called *supported* if for every ground atom A such that $I \models A$ there exist $B_1, \ldots, B_n$ such that $A \leftarrow B_1, \ldots, B_n \in ground(P)$ and $I \models B_1 \wedge \ldots \wedge B_n$. □

Lemma 3.6 *For a left terminating program P, M_P is the unique supported Herbrand model of P.* □

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, we omit the proofs that a given Herbrand interpretation is the least Herbrand model of a given left terminating program. 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 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 with the substitutions. More precisely, we introduce the following definition.

Definition 3.7 Consider a program P.

(i) 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 $P \models Q'$.

(ii) 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 3.8 (Soundness of LD-resolution) *Consider a program P and a query Q. Every computed instance of Q is a correct instance of Q.* □

Theorem 3.9 (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 introduce now 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 [AO91]). We now present an easy method of establishing constructs of the form $\{Q\}\ P\ \mathcal{Q}$.

Theorem 3.10 *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

$$\text{every correct instance } Q' \text{ of } Q \text{ is ground.} \tag{1}$$

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 3.9 there exists a computed instance Q_2 of Q such that $Q_2 \leq Q_1$. By the Soundness Theorem 3.8 Q_2 is a correct instance of Q, so by (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_2 of Q. By the Soundness Theorem 3.8 Q_2 is a correct instance of Q. By (1) Q_2 is ground, so $Q_2 \in \mathcal{Q}$. □

Examples

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

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

(i) Given ground lists s,t,u we have $\mathtt{app(s,t,u)} \in M_{\mathtt{APPEND-T}}$ iff $\mathtt{s} * \mathtt{t} = \mathtt{u}$. Consequently

- when s*t = u, $\{\mathtt{app(s,t,u)}\}\ \mathtt{APPEND} - \mathtt{T}\ \{\mathtt{app(s,t,u)}\}$,
- when s*t $\neq$ u, $\{\mathtt{app(s,t,u)}\}\ \mathtt{APPEND} - \mathtt{T}\ \emptyset$.

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

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

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

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

But each list can be split only in finitely many ways, so the set $[\mathtt{app(Xs,Ys,u)}] \cap M_{\mathtt{APPEND-T}}$ is finite. Thus

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

A more interesting example is the SEQUENCE-T program. Call a list of 27 numbers satisfying the description of the sequence a *desired list*. We leave to the reader checking that

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

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

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

Exercise 1 Consider the following REVERSE-T program:

```
reverse(Xs, Ys) ← reverse_dl(Xs, Ys-[]).
reverse_dl([X | Xs], Ys-Zs) ← reverse_dl(Xs, Ys-[X | Zs]).
reverse_dl([], Xs-Xs) ← list(Xs).
```

augmented by the LIST program.

Given a list s let rev(s) denote its reverse. Prove that for a ground list s

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

by checking that reverse_dl(s,t-u) $\in M_{\texttt{REVERSE\_T}}$ iff s,t,u are ground lists and rev(s)*u = t. □

Clearly, the above approach to partial correctness cannot be used to reason about queries with "non-ground inputs", like app(s,t,Zs) where s,t are non-ground lists, since $[\texttt{app(s,t,Zs)}] \cap M_{\texttt{APPEND\_T}}$ is then infinite. The treatment of such queries needs to await another paper.

4 Pure Prolog with Arithmetic

We now move on to the study of the second subset of Prolog, namely 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. We refer to Apt and Pedreschi [AP93] for a proof that QUICKSORT is left terminating and that for a list t all LD-derivations of $\texttt{QUICKSORT} \cup \{\texttt{qs(t,Ys)}\}$ are finite.

4.1 Absence of Errors

To deal with errors we provide some proof theoretic means to prove absence of runtime errors for desired queries. We found it convenient to use the notion of a well-typed program recently proposed by Bronsard, Lakshman and Reddy [BLR92] (where, unfortunately, it is called a well-moded program). It allows us to ensure that the input positions of the selected atoms remain correctly typed during the program execution. We recall here the definitions. We follow here the presentation of Apt and Etalle [AE93].

We start with the notion of a mode used to define input and output positions of a relation.

Definition 4.1 A *mode* for an n-ary relation symbol p is a function m_p from $[1,n]$ to the set $\{+,-\}$. If $m_p(i) = \text{'+'}$, we call i an *input position* of p and if $m_p(i) = \text{'-'}$, we call i an *output position* of p (both w.r.t. m_p). A *moding* is a collection of modes, each for a different relation symbol. □

The definition of moding assumes one mode per relation in a program. Multiple modes may be obtained by simply renaming the relations. When every considered relation has a mode associated with it, we can talk about input positions and output positions of an atom.

Next, we introduce types. The following very general definition is sufficient for our purposes.

Definition 4.2 A *type* is a decidable set of terms closed under substitution. □

By a *typed term* we mean a construct of the form $s : S$ where s is a term and S is a type. Given a sequence $\mathbf{s} : \mathbf{S} = s_1 : S_1, \ldots, s_n : S_n$ of typed terms we write $\mathbf{s} \in \mathbf{S}$ if for $i \in [1, n]$ we have $s_i \in S_i$.

Certain types will be of special interest below:

U — 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.

From now on we fix a specific set of types, denoted by *Types*, which includes the above ones. We also associate types with relation symbols.

Definition 4.3 A *type* for an n-ary relation symbol p is a function t_p from $[1, n]$ to the set *Types*. If $t_p(i) = T$, we call T *the type associated with the position* i *of* p. Assuming a type t_p for the relation p, we say that an atom $p(s_1, \ldots, s_n)$ is *correctly typed in position* i if $s_i \in t_p(i)$. □

We now assume that every considered relation has a mode and a type associated with it, so we can talk about types of input positions and of output positions of an atom. An n-ary relation p with a mode m_p and type t_p will be denoted by

$$p(m_p(1) : t_p(1), \ldots, m_p(n) : t_p(n)).$$

For example, $\mathtt{part}(+ : Gae, + : ListGae, - : ListGae, - : ListGae)$ denotes a relation `part` with four arguments: the first position is moded as input and typed *Gae*, the second position is moded as input and typed *ListGae*, and the third and fourth positions are moded as output and typed *ListGae*.

Well-Typed Programs

The notion of well-typed queries and programs relies on the concept of a type judgement.

Definition 4.4

- A *type judgement* is a statement of the form $\mathbf{s} : \mathbf{S} \Rightarrow \mathbf{t} : \mathbf{T}$.
- A type judgement $\mathbf{s} : \mathbf{S} \Rightarrow \mathbf{t} : \mathbf{T}$ *is true*, notation: $\models \mathbf{s} : \mathbf{S} \Rightarrow \mathbf{t} : \mathbf{T}$, if for all substitutions θ, $\mathbf{s}\theta \in \mathbf{S}$ implies $\mathbf{t}\theta \in \mathbf{T}$. □

For example, the type judgement $x : Gae,\ l : ListGae \Rightarrow [x \mid l] : ListGae$ is true.

To simplify the notation, when writing an atom as $p(\mathbf{u} : \mathbf{S}, \mathbf{v} : \mathbf{T})$ we now assume that $\mathbf{u} : \mathbf{S}$ is a sequence of typed terms filling in the input positions of p and $\mathbf{v} : \mathbf{T}$ is a sequence of typed terms filling in the output positions of p.

The following notion is due to Bronsard, Lakshman and Reddy [BLR92].

Definition 4.5

- A query $p_1(\mathbf{i_1} : \mathbf{I_1}, \mathbf{o_1} : \mathbf{O_1}), \ldots, p_n(\mathbf{i_n} : \mathbf{I_n}, \mathbf{o_n} : \mathbf{O_n})$ is called *well-typed* if for $j \in [1, n]$

$$\models \mathbf{o_1} : \mathbf{O_1}, \ldots, \mathbf{o_{j-1}} : \mathbf{O_{j-1}} \Rightarrow \mathbf{i_j} : \mathbf{I_j}.$$

- A clause

$$p_0(\mathbf{o_0} : \mathbf{O_0}, \mathbf{i_{n+1}} : \mathbf{I_{n+1}}) \leftarrow p_1(\mathbf{i_1} : \mathbf{I_1}, \mathbf{o_1} : \mathbf{O_1}), \ldots, p_n(\mathbf{i_n} : \mathbf{I_n}, \mathbf{o_n} : \mathbf{O_n})$$

is called *well-typed* if for $j \in [1, n+1]$

$$\models \mathbf{o_0} : \mathbf{O_0}, \ldots, \mathbf{o_{j-1}} : \mathbf{O_{j-1}} \Rightarrow \mathbf{i_j} : \mathbf{I_j}.$$

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

In general it is undecidable whether a program is well-typed. However, recently Aiken and Lakshman [AL93] showed that this problem is decidable for a large class of types which includes the ones studied here.

Bronsard, Lakshman and Reddy [BLR92] noticed the following persistence property of the notion of being well-typed.

Lemma 4.6 *An LD-resolvent of a well-typed query and a well-typed clause that is variable disjoint with it, is well-typed.* □

This allows us to draw the following important conclusion.

Corollary 4.7 *Let P and Q be well-typed, and let ξ be an LD-derivation of $P \cup \{Q\}$. All atoms selected in ξ are correctly typed in their input positions.*

Proof. A variant of a well-typed clause is well-typed and the first atom of a well-typed query is correctly typed in its input positions. □

To see the usefulness of this corollary let us return to the QUICKSORT program. To prove absence of errors we start by typing the relation qs in the way reflecting its use, so $\mathtt{qs}(+ : \mathit{ListGae}, - : \mathit{ListGae})$, and the built-in's $>$ and $\leq$ in such a way that the above corollary can be applied so $>(+ : \mathit{Gae}, + : \mathit{Gae})$, $\leq(+ : \mathit{Gae}, + : \mathit{Gae})$.

We now complete the typing in such a way that QUICKSORT is well-typed:
$\mathtt{part}(+ : \mathit{Gae}, + : \mathit{ListGae}, - : \mathit{ListGae}, - : \mathit{ListGae})$,
$\mathtt{app}(+ : \mathit{ListGae}, + : \mathit{ListGae}, - : \mathit{ListGae})$.

Assume now that s is a list of gae's. By Corollary 4.7 we conclude that all atoms selected in the LD-derivations of $\mathtt{QUICKSORT} \cup \{\mathtt{qs(s,t)}\}$ are correctly typed in their input positions. In particular, when these atoms are of the form $u > v$ or $u \leq v$, both u and v are gae's. Thus the LD-derivations of $\mathtt{QUICKSORT} \cup \{\mathtt{qs(s,t)}\}$ do not end in an error.

Exercise 2 Consider the LENGTH program:

```
length([H | Ts], N) ← length(Ts, M), N is M+1.
length([], 0).
```

Prove that for a ground list t

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

□

4.2 Partial Correctness

When dealing with partial correctness of programs that use arithmetic relations we have to remember that to each program we added infinitely many clauses which define the used arithmetic relations. Both the Soundess Theorem 3.8 and the Strong Completeness Theorem 3.9 remain valid for programs with infinitely many clauses, however completeness does not hold anymore in 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 3.10 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 4.8 *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 Soundess Theorem 3.8 and the Strong Completeness Theorem 3.9 remain valid. For the completeness theorem this is not obvious, since it usually relies on the Lifting Lemma which not does not hold now. However, the admirably short and elegant proof of Stärk [Stä90] does not use the Lifting Lemma and carries through. Consequently, the proof of Theorem 3.10 carries through, as well. □

To apply this theorem let us return to 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

```
part(X, [], [], []) ← X ≥ 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 *and* 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 by list(+:ListGae) we can conclude that for a list of gae's s the LD-derivations of QUICKSORT-T $\cup$ {qs(s, Ys)} do not end in error. Moreover, the previously given argument about the termination of QUICKSORT is also valid for QUICKSORT-T .

It is easy to check that

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

For a list of gae's s the set $[\texttt{qs(s, Ys)}] \cap M_{\texttt{QUICKSORT-T}}$ consists of just one element: qs(s,t), where t is a sorted permutation of s. Consequently, by Theorem 4.8

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

5 Pure Prolog with Negation

Finally, we deal with the third subset of Prolog, namely pure Prolog with negation. We call programs written in this subset general programs.

5.1 Absence of Floundering

To prove absence of floundering w.r.t. leftmost selection rule we use the notion of a well-moded program, is essentially due to Dembinski and Maluszynski [DM85]. We generalize it here to general programs. Assume that every considered relation has a mode associated with it. 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 that $\mathbf{v}$ is a sequence of terms filling in the output positions of p. Below $\odot$ 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 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 [AP92] and, independently, Stroetman [Str93].

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.* □

Example

To see the use of this theorem consider the general program TRANS-T which computes the transitive closure of a binary relation. Such a relation is represented below as a ground list of edges. In turn, an edge from a to b is represented by a list $[a, b]$.

```
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(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 binary relation `e`, one evaluates the query `trans(x, y, e, [x])`.

With the moding `trans(-,-,+,+)`, `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. Thus for `e,s` ground, TRANS-T ∪ {`trans(a,b,e,s)`} does not flounder. In particular, TRANS-T is non-floundering.

5.2 Termination

To deal with termination we use the approach Apt and Pedreschi [AP93] which generalizes the method of Subsection 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. □

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* 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 I is a model of $comp(P^-)$. □

The following results relate 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.* □

Theorem 5.8 *Let P be a left terminating, non-floundering general program. Then for some level mapping $|\ |$ and an interpretation I of P*

(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. $|\ |$. □

Apt and Pedreschi [AP93] 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 an 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.

5.3 Partial Correctness

When reasoning about partial correctness of general programs we face the obvious problem of determining their declarative semantics. We solve this problem by restricting our attention to a specific class of general programs. The notion of a supported interpretation extends to general programs in an obvious way. The following result of Apt and Pedreschi [AP93] is crucial.

Theorem 5.9 *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 $P \cup \{Q\}$. □

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.

We now need to revise Definition 3.7.

Definition 5.10 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 remains the same. The following soundness and completeness theorems are of help.

Theorem 5.11 (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 5.12 (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. By Theorem 5.9 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 5.13 *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. Analogous to the proof of Theorem 3.10. □

As in the case of pure Prolog programs, for a query consisting of just one atom A the assumption of the theorem can be rephrased (thanks to Theorem 5.9) as "the set $[A] \cap M_P$ is finite".

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

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

Given a binary relation e denote its transitive closure by e*. Then [a,b] $\in$ e* iff there exists in e an acyclic path from a to b which avoids [a]. By Theorem 5.13 we conclude that

- when [a,b] $\in$ e*, {trans(a,b,e,[a])} TRANS − T {trans(a,b,e,[a])},
- when [a,b] $\notin$ e*, {trans(a,b,e,[a])} TRANS − T $\emptyset$.

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

Exercise 3 Prove that for a binary relation e

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

□

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 at the end of Subsection 4.2. Let qs(s,t) be a well-typed query and let ξ be an LD-derivation of QUICKSORT $\cup$ {qs(s,t)}. By Corollary 4.7, if the selected atom is of the form part(s_1, s_2, s_3, s_4) then $s_1 \in Gae$, and if the selected atom is of the form app(s_1, s_2, s_3) then $s_2 \in List$.

Thus in both cases in the corresponding LD-derivation of QUICKSORT-T $\cup$ {qs(s,t)} the inserted "type checks", namely X $\geq$ 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

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

The same approach can be applied to other programs, including TRANS-T for which Corollary 4.7 needs to be extended to general programs in the obvious way.

6.2 Final Remarks

The aim of this paper was to show that it is possible to reason about correctness of various Prolog programs by means of simple arguments based on declarative semantics. We hope that this work can form a basis for a similar study of other languages based on the logic programming paradigm. It is quite possible that the proposed methods are in some instances special cases of approaches proposed earlier. Our point is that unless the verification method is easy and amenable to informal use, it will be ignored. So searching for simplicity is worth the effort.

We conclude by stating a number of, perhaps controversial, opinions.

1. A Prolog program written in one of the considered subsets is *declarative* if its correctness for the class of queries "of interest" can be established by means of static analysis and using first-order semantics. In this paper we showed how to reduce the latter to a simple study of supported Herbrand models.

2. From this viewpoint some pure Prolog programs are not declarative.

3. The following view of (general) left terminating programs can be helpful. The supported Herbrand model uniquely determines ground queries which succeed and terminate w.r.t. the leftmost selection rule. In pure Prolog by the Lifting Lemma all generalizations of these ground queries also succeed ... but only in case of logic programming. In pure Prolog such a generalization can fail to terminate, and for the other two subsets it can end in an

error or flounder. So first we should think in terms of ground queries and then "lift" each of them, but "carefully".

4. Assertional proof methods, while helpful, do not reflect the essence of declarative programming.

5. Correctness of programs that use accumulators and difference lists should be preferably dealt with by means of program transformations.

6. The treatment of "ill-typed" programs is quite roundabout and justifies a systematic introduction of types (or sorts) into the basic framework of logic programming.

7. It would be interesting to develop a theory of correctness of non-terminating Prolog programs based on their declarative semantics (like the one developed in Chapter 6 of Lloyd [Llo87]).

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 paper. This research was partly supported by the ESPRIT Basic Research Action 6810 (Compulog 2).

References

[AD92] K.R. Apt and K. Doets. A new definition of SLDNF-resolution. ILLC Prepublication Series CT-92-03, Department of Mathematics and Computer Science, University of Amsterdam, The Netherlands, 1992. Accepted for publication in *Journal of Logic Programming*.

[AE93] K. R. Apt and S. Etalle. On the unification free Prolog programs. In S. Sokolowski, editor, *Proceedings of the Conference on Mathematical Foundations of Computer Science (MFCS 93)*, Lecture Notes in Computer Science, Berlin, 1993. Springer-Verlag. To appear.

[AL93] A. Aiken and T.K. Lakshman. Automatic mode checking for logic programs. Technical report, Department of Computer Science, University of Illinois at Urbana Champaign, 1993.

[AO91] K.R. Apt and E.-R. Olderog. *Verification of Sequential and Concurrent Programs*. Texts and Monographs in Computer Science, Springer-Verlag, New York, 1991.

[AP92] K. R. Apt and A. Pellegrini. On the occur-check free Prolog programs. Technical Report CS-R9238, CWI, Amsterdam, 1992. Accepted for publication in *ACM Toplas*.

[AP93] K. R. Apt and D. Pedreschi. Reasoning about termination of pure Prolog programs. *Information and Computation*, 1993. to appear.

[Apt90] K. R. Apt. Logic programming. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, pages 493–574. Elsevier, 1990. Vol. B.

[BD77] R.M. Burstall and J. Darlington. A transformation system for developing recursive programs. *Journal of the ACM*, 24(1):44–67, 1977.

[BLR92] F. Bronsard, T.K. Lakshman, and U.S. Reddy. A framework of directionality for proving termination of logic programs. In K.R. Apt, editor, *Proc. of the Joint International Conference and Symposium on Logic Programming*, pages 321–335. MIT Press, 1992.

[BW88] R. Bird and Ph. Wadler. *Introduction to Functional Programming.* International Series in Computer Science, Prentice Hall, Englewood Cliffs, NJ, 1988.

[CC88] H. Coelho and J.C. Cotta. *Prolog by Example.* Springer-Verlag, Berlin, 1988.

[Cla79] K. L. Clark. Predicate logic as a computational formalism. Res. Report DOC 79/59, Imperial College, Dept. of Computing, London, 1979.

[CM88] K.M. Chandy and J. Misra. *Parallel Program Design: A Foundation.* Addison-Wesley, New York, 1988.

[CT77] K. Clark and S-Å. Tärnlund. A First Order Theory of Data and Programs. In *Information Processing '77*, pages 939–944. North-Holland, 1977.

[Der90] P. Deransart. Proof methods of declarative properties of definite programs. Technical Report 1248, INRIA – Rocquencourt, 1990.

[Dev90] Y. Deville. *Logic Programming. Systematic Program Development.* International Series in Logic Programming. Addison-Wesley, 1990.

[DFT91] 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.

[Dij76] E.W. Dijkstra. *A Discipline of Programming.* Prentice-Hall, Englewood Cliffs, N.J., 1976.

[DM85] 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.

[Gri81] D. Gries. *The Science of Programming.* Springer-Verlag, New York, 1981.

[Hog84] C.J. Hogger. *Introduction to Logic Programming.* Academic Press, London, 1984.

[Kun88] 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.

[Kun89] K. Kunen. Signed data depedencies in logic programs. *Journal of Logic Programming*, 7:231–246, 1989.

[Llo84] J. W. Lloyd. *Foundations of Logic Programming.* Springer-Verlag, Berlin, 1984.

[Llo87] J. W. Lloyd. *Foundations of Logic Programming.* Springer-Verlag, Berlin, second edition, 1987.

[Mee86] L. Meertens. Algorithmics — towards programming as a mathematical activity. In J. W. de Bakker, M. Hazewinkel, and J.K. Lenstra, editors, *Proceedings of the CWI Symposium on Mathematics and Computer Science*, volume 1 of *CWI Monographs*, pages 289–334. North–Holland, 1986.

[MP92] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems.* Springer-Verlag, New York, 1992.

[MT92] M. Martelli and C. Tricomi. A new SLDNF-tree. *Information Processing Letters*, 43(2):57–62, 1992.

[SS86] L. Sterling and E. Shapiro. *The Art of Prolog.* MIT Press, 1986.

[Stä90] R. Stärk. A direct proof for the completeness of SLD-resolution. In E. Börger, H. Kleine Büning, and M.M. Richter, editors, *Computation Theory and Logic 89*, Lecture Notes in Computer Science 440, pages 382–383. Springer-Verlag, 1990.

[Str93] K. Stroetman. A completeness result for SLDNF resolution. *The Journal of Logic Programming*, 15:337–357, 1993.