Reasoning about Termination of Pure Prolog Programs

Krzysztof R. Apt
Centre for Mathematics and Computer Science
Kruislaan 413, 1098 SJ Amsterdam, The Netherlands
and
Faculty of Mathematics and Computer Science
University of Amsterdam, Plantage Muidergracht 24
1018 TV Amsterdam, The Netherlands

Dino Pedreschi
Dipartimento di Informatica, Università di Pisa
Corso Italia 40, 56125 Pisa, Italy

Abstract

We provide a theoretical basis for studying termination of (general) logic programs with the Prolog selection rule. To this end we study the class of *left terminating* programs. These are logic programs that terminate with the Prolog selection rule for all ground goals. We offer a characterization of left terminating positive programs by means of the notion of an *acceptable program* that provides us with a practical method of proving termination. The method is illustrated by giving a simple proof of termination of the quicksort program for the desired class of goals.

Then we extend this approach to the class of general logic programs by modifying the concept of acceptability. We prove that acceptable general programs are left terminating. The converse implication does not hold but we show that under the assumption of non-floundering from ground goals every left terminating general program is acceptable. Finally, we prove that various ways of defining semantics coincide for acceptable general programs. We illustrate the use of this extension by giving simple proofs of termination of a "game" program and the transitive closure program for the desired class of goals.

Keywords and Phrases: general Prolog programs, termination, declarative semantics, left terminating programs, acceptable programs.
1985 Mathematics Subject Classification: 68Q40, 68T15.
CR Categories: F.3.2., F.4.1, H.3.3, I.2.3.

Notes. This research was partly done during the authors' stay at the Department of Computer Sciences, University of Texas at Austin, Austin, Texas, U.S.A. . First author's work was partly supported by ESPRIT Basic Research Action 3020 (Integration). Second author's work was partly supported by ESPRIT Basic Research Action 3012 (Compulog) and by the Italian National Research Council – C.N.R.. This paper is based on our previous papers [AP90] and [AP91].

1 Introduction

Background

Prolog is a programming language based on logic programming. However, the use of a fixed selection rule combined with the depth first search in the resulting search trees makes Prolog and logic programming different. As a consequence various completeness results linking the procedural and declarative interpretation of logic programs cannot be directly applied to Prolog programs. This mismatch makes it difficult to study Prolog programs using only the logic programming theory. Clearly the main problem is the issue of termination: a Prolog interpreter will miss a solution if all success nodes lie to the right of an infinite path in the search tree.

The aim of this paper is to study termination of pure Prolog programs. By "pure" we mean here programs with no built-in's and no cut. We do handle negation but allow only its "safe" use which means that only ground negative literals are resolved by the negation as failure rule. Thus the syntax of "pure" Prolog programs coincides with that of general logic programs.

By termination we mean here finiteness of *all* possible Prolog derivations starting in the initial goal. This stronger notion of termination abstracts from the ordering of the program clauses and seems to follow a good programming practice. We consider here pure Prolog programs that terminate for all ground goals. We call such programs *left terminating*. The restriction to such programs leads to a simple to use method of proving termination that is applicable to an overwhelming class of pure Prolog programs considered in the book of Sterling and Shapiro [SS86]. To show that a program exhibits a proper termination behaviour we first show that it is left terminating and then that it terminates for certain types of non-ground inputs. Our method of showing the former will also allow us to establish the latter.

When studying Prolog programs from the point of view of termination it is useful to notice that some programs terminate for all ground goals for *all* selection rules. Such programs are extensively studied in Bezem [Bez89] where they are called *terminating programs*. These are usually programs whose termination depends on a simple reduction in one or more arguments. Examples of terminating programs are `append`, `member`, `N queens`, various tree insertion and deletion programs and several others.

However, some Prolog programs satisfy such a strong termination property but fail to terminate for certain desired forms of inputs for some selection rules.

An example is the following `append3` program in which the `append` program is used:

```
append3(Xs, Ys, Zs, Us) ←
  append(Xs, Ys, Vs),
  append(Vs, Zs, Us).
```

Then `append3` is a terminating program which terminates for the goal ← `append3(xs, ys, zs, Us)`, where `xs, ys, zs` are lists and `Us` a variable, when the Prolog selection rule is used but fails to terminate when the rightmost selection rule is used.

Worse yet, some programs fail to be terminating even though they terminate for the Prolog selection rule for the desired class of inputs. An example is the `flatten` program which collects all the nodes of a tree in a list:

```
flatten(nil, []) ←.
flatten(t(L, X, R), Xs) ←
  flatten(L, X1s),
  flatten(R, X2s),
  append(X1s, [X | X2s], Xs).
```

`flatten` is not a terminating program but it terminates for the goal $\leftarrow$ `flatten(x, Xs)`, where `x` is a ground term and `Xs` a variable, when the Prolog selection rule is used.

In general, the problem arises due to the use of local variables, i.e. variables which appear in the body of a clause but not in its head. Several left terminating Prolog programs use local variables in an essential way and consequently fail to be terminating. Examples of such programs are various sorting and permutation programs and graph searching programs. Programs which fall into this category are usually of the form "generate and test" or "divide and conquer".

In this paper we provide a framework to study left terminating programs. To this end we refine the ideas of Bezem [Bez89] and Cavedon [Cav89] and use their concept of a level mapping. This is a function assigning natural numbers to ground atoms. Our main tool is the concept of an *acceptable program*. Intuitively, a program P is acceptable if for some level mapping and a model I of P, for all ground instances of the clauses of the program, the level of the head is smaller than the level of atoms in a certain prefix of the body. Which prefix is considered is determined by the model I, which embodies the limited declarative knowledge about the program that is used in the termination proof.

We prove that the notions of left termination and acceptability coincide. The proof of this fact uses an iterated multiset ordering. This equivalence result provides us with a method of proving left termination. Moreover, it allows us to prove termination of a left terminating Prolog program for a class of non-ground goals. The method is illustrated by proving termination of the quicksort program.

Then we extend this approach to termination to general Prolog programs, i.e. programs allowing negative literals. More precisely, we consider here general logic programs executed with the leftmost selection rule used in Prolog. The approach is based on a modification of the concept of acceptability. In the case of general Prolog programs we require that the interpretation I should be a model of the considered program P *and* a model of Clark's completion of the "negative" fragment of P. We prove that acceptable general programs are left terminating. However, the converse implication does not hold due to the possibility of floundering. On the other hand, we show that for general programs that do not flounder from ground general goals the concepts of left termination and acceptability do coincide. Also, we prove that various ways of defining semantics coincide for acceptable general programs.

As before, once the left termination of a general Prolog program is established, non-ground terminating goals can be identified. We illustrate the use of this extension by providing simple proofs of termination of a "game" program and the transitive closure program for the desired class of goals.

Plan of the paper

This paper is organized as follows. In the next section we introduce the concept of a left terminating program. This is a program that terminates for all ground goals w.r.t. Prolog selection rule. Then we provide a useful characterization of left terminating programs by introducing the notion of an acceptable program and proving that the notions of acceptability and left termination coincide. The crucial concept here is that of a bounded goal. It allows us to characterize terminating goals.

Then, in Section 3 we prove left termination of the quicksort program by providing a simple proof of its acceptability. Using the concept of boundedness we show that the quicksort program terminates w.r.t. a desired class of non-ground goals.

In Section 4 we extend the notions of left termination and acceptability to general programs. We show that acceptable programs are left terminating, and that left terminating, non-floundering programs are acceptable. This allows us to extend our method to reason about termination of general Prolog programs.

Then, in Section 5 we apply this method to a "game" and a transitive closure program, by establishing their acceptability. Again, by using the concept of boundedness we prove that these programs terminate w.r.t. a desired class of goals.

In Section 6 we prove that various ways of defining semantics of general programs coincide. In particular, we show that the completion of an acceptable program has a unique Herbrand model, which coincides with its unique 3-valued Herbrand model. For non-floundering acceptable programs, this model coincides with the set of facts which can be inferred using the SLDNF-resolution with the leftmost selection rule.

Finally, in Section 7 we assess the proposed proof method and discuss related work.

Preliminaries

We use standard notation and terminology of Lloyd [Llo87] or Apt [Apt90] with the exception that general logic programs are called in Lloyd [Llo87] normal logic programs. In particular, we use the following abbreviations for a (general) logic program P (or simply a (*general*) *program*):
B_P for the Herbrand Base of P,
T_P for the immediate consequence operator of P,
M_P for the least Herbrand model of P,
$ground(P)$ for the set of all ground instances of clauses from P,
$comp(P)$ for Clark's completion of P.

Also, we use Prolog's convention identifying in the context of a program each string starting with a capital letter with a variable, reserving other strings for the names of constants, terms or relations. So, for example Xs stands for a variable whereas xs stands for a term.

In the programs we use the usual list notation. The constant $[\,]$ denotes the empty list and $[\,.\mid.\,]$ is a binary function which given a term x and a list xs produces a new list $[\,x \mid xs\,]$ with head x and tail xs. By convention, identifiers ending with "s", like xs, will range over lists. The standard notation $[\,x_1, \ldots, x_n\,]$, for $n \geq 0$, is used as an abbreviation of $[\,x_1 \mid [\ldots [\,x_n | [\,]\,] \ldots]]$. In general, the Herbrand Universe will also contain "impure" elements that contain $[\,]$ or $[\,.\mid.\,]$ but are not lists - for example $s([\,])$ or $[s(0) \mid 0]$ where 0 is a constant and s a unary function symbol. They will not cause any complications.

Throughout the paper we consider SLD-resolution with one selection rule only – namely that of Prolog, usually called the leftmost selection rule. As S in SLD stands for "selection rule", we denote this form of resolution by LD (*L*inear resolution for *D*efinite clauses). The concepts of LD-derivation, LD-refutation, LD-tree, etc. are then defined in the usual way. By "pure Prolog" we mean in this paper the LD-resolution combined with the depth first search in the LD-trees.

By choosing variables of the input clauses and the used mgu's in a fixed way we can assume that for every program P and goal G there exists exactly one LD-tree for $P \cup \{G\}$.

2 Left Termination of Positive Programs

Our interest here is in terminating Prolog programs. This motivates the following concept.

Definition 2.1 A program P is called *left terminating* if all LD-derivations of P starting in a ground goal are finite. □

In other words, a program is left terminating if all LD-trees for P with a ground root are finite. When studying Prolog programs, one is actually interested in proving termination of a given program not only for all ground goals but also for a class of non-ground goals constituting the intended queries. Our method of proving left termination will allow us to identify for each program such a class of non-ground goals.

Let us consider now how to prove that a program is left terminating. Starting from Floyd [Flo67] the classical proofs of program termination have been based on the use of well-founded orderings. This approach has been successfully used in the area of logic programming (see e.g. Bezem [Bez89], Cavedon [Cav89]) but with no attention paid to Prolog programs. The notable exception is Deville [Dev90].

We obtain the desired method by a modification of the ideas of Bezem [Bez89] and Cavedon [Cav89].

Recurrent Programs

It is useful to recall first some concepts from Bezem [Bez89] and Cavedon [Cav89].

Definition 2.2

(i) A *level mapping* for a program P is a function $|\ | : B_P \rightarrow N$ of ground atoms to natural numbers. For $A \in B_P$, $|A|$ is the level of A.

(ii) An atom A is *bounded with respect to a level mapping* $|\ |$ if $|\ |$ is bounded on the set $[A]$ of ground instances of A.

(iii) A goal is bounded if all its atoms are.

(iv) A program P is called *recurrent with respect to a level mapping* $|\ |$, if for every clause $A \leftarrow B_1, \ldots, B_n$ in $ground(P)$

$$|A| > |B_i| \text{ for } i \in [1, n].$$

A program P is called *recurrent* iff it is recurrent with respect to some level mapping. □

Thus level mapping is defined only on ground atoms, but the concept of boundedness allows us to "lift" the level mapping to non-ground atoms. Boundedness is crucial when considering termination.

Definition 2.3 A program is called *terminating*, if all its SLD-derivations starting in a ground goal are finite.

Hence, terminating programs have the property that the SLD-trees of ground goals are finite, and any search procedure in such trees will always terminate, independently from the adopted selection rule.

The following results of Bezem [Bez89] show the connection between these concepts.

Theorem 2.4

(i) Let P be a recurrent program and G a bounded goal. Then all SLD-derivations of $P \cup \{G\}$ are finite.

(ii) A program is recurrent if and only if it is terminating. □

Because of this result recurrent programs and bounded goals are too restrictive concepts to deal with Prolog programs, as a larger class of programs and goals is terminating when adopting a specific selection rule, e.g. Prolog selection rule.

Example 2.5
(i) Consider the following program `even` which defines even numbers and the "less than or equal" relation:

```
even(0) ←.
even(s(s(X))) ← even(X).

lte(0,Y) ←.
lte(s(X),s(Y)) ← lte(X,Y).
```

`even` is recurrent with $|even(s^n(0))| = n$ and $|lte(s^n(0), s^m(0))| = \min\{n, m\}$. Now consider the goal:

$$G = \leftarrow lte(x, s^{100}(0)), even(x)$$

which is supposed to compute the even numbers not exceeding 100. The LD-tree for G is finite, whereas there exists an infinite SLD-derivation when the rightmost selection rule is used. As a consequence of Theorem 2.4 (i) the goal G is not bounded, although it can be evaluated by a finite Prolog computation.

Actually, most "generate and test" Prolog programs are not recurrent, as they heavily depend on the left-to-right order of evaluation, like the example above.

(ii) Consider the following naive `reverse` program:

```
reverse([], []) ←.
reverse([X | Xs], Ys) ←
  reverse(Xs, Zs),
  append(Zs, [X], Ys).

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

The ground goal $\leftarrow reverse(xs, ys)$, for a list xs with at least two elements and an arbitrary list ys, has an infinite SLD-derivation, obtained by using the selection rule which selects the leftmost atom at the first two steps, and the second leftmost atom afterwards. By Theorem 2.4(ii) `reverse` is not recurrent.

(iii) Consider the following program `DC`, representing a (binary) "divide and conquer" schema; it is parametric with respect to the *base, conquer, divide* and *merge* predicates.

```
dc(X,Y) ←
  base(X),
  conquer(X,Y).
dc(X,Y) ←
  divide(X,X1,X2),
```

```
    dc(X1,Y1),
    dc(X2,Y2),
    merge(Y1,Y2,Y).
```

Many programs naturally fit into this schema, or its generalization to non fixed arity of the `divide/merge` predicates. Unfortunately, `DC` is not recurrent: it suffices to take a ground instance of the recursive clause with $X = a$, $X1 = a$, $Y = b$, $Y1 = b$, and observe that the atom *dc(a,b)* occurs both in the head and in the body of such a clause. In this example, the leftmost selection rule is needed to guarantee that the input data is divided into subcomponents before recurring on such subcomponents. □

Acceptable Programs

To cope with these difficulties we modify the definition of a recurrent program as follows.

Definition 2.6 Let P be a program, $|\,|$ a level mapping for P and I a (not necessarily Herbrand) model of P. P is called *acceptable with respect to* $|\,|$ *and* I if for every clause $A \leftarrow B_1, \ldots, B_n$ in $ground(P)$ the following implication holds for $i \in [1, n]$:

$$\text{if } I \models \wedge_{j=1}^{i-1} B_j \text{ then } |A| > |B_i|.$$

In other words, we have for every clause $A \leftarrow B_1, \ldots, B_n$ in $ground(P)$

$$|A| > |B_i| \text{ for } i \in [1, \bar{n}],$$

where

$$\bar{n} = \min(\{n\} \cup \{i \in [1,n] \mid I \not\models B_i\}).$$

P is called *acceptable* if it is acceptable with respect to some level mapping and a model of P. □

The use of the premise $I \models \wedge_{j=1}^{i-1} B_j$ forms the *only* difference between the concepts of recurrence and acceptability. Intuitively, this premise expresses the fact that when in the evaluation of the goal $B_1, \ldots, B_n$ using the leftmost selection rule the atom B_i is reached, the atoms $B_1, \ldots, B_{i-1}$ are already refuted. Consequently, by the soundness of the LD-resolution, these atoms are all true in I.

Alternatively, we may define $\bar{n}$ by

$$\bar{n} = \begin{cases} n & \text{if } I \models B_1 \wedge \ldots \wedge B_n, \\ i & \text{if } I \models B_1 \wedge \ldots \wedge B_{i-1} \text{ and } I \not\models B_1 \wedge \cdots \wedge B_i. \end{cases}$$

Thus, given a level mapping $|\,|$ for P and a model I of P, in the definition of acceptability w.r.t. $|\,|$ and I for every clause $A \leftarrow B_1, \ldots, B_n$ in $ground(P)$ we only require that the level of A is higher than the level of B_i's in a certain prefix of $B_1, \ldots, B_n$. Which B_i's are taken into account is determined by the model I. If $I \models B_1 \wedge \ldots \wedge B_n$ then all of them are considered and otherwise only those whose index is $\leq \bar{n}$, where $\bar{n}$ is the least index i for which $I \not\models B_i$.

The following observation is immediate.

Lemma 2.7 *Every recurrent program is acceptable.* □

Our aim is to prove that the notions of acceptability and left termination coincide.

Multiset ordering

To prove one half of this statement we use the multiset ordering. A *multiset*, sometimes called *bag*, is an unordered sequence. Given a (non-reflexive) ordering $<$ on a set W, the *multiset ordering over* $(W, <)$ is an ordering on finite multisets of the set W. It is defined as the transitive closure of the relation in which X is smaller than Y if X can be obtained from Y by replacing an element a of Y by a finite (possibly zero) number of elements each of which is smaller than a in the ordering $<$.

In symbols, first we define the relation $\prec$ by

$$X \prec Y \text{ iff } X = Y - \{a\} \cup Z \text{ for some } Z \text{ such that } b < a \text{ for } b \in Z,$$

where X, Y, Z are finite multisets of elements of W, and then define the multiset ordering over $(W, <)$ as the transitive closure of the relation $\prec$.

It is well-known (see e.g. Dershowitz [Der87]) that multiset ordering over a well-founded ordering is again well-founded. Thus it can be iterated while maintaining well-foundedness.

What we need in our case is two fold iteration. We start with the set of natural numbers N ordered by $<$ and apply the multiset ordering twice. We call the first iteration multiset ordering and the second *double multiset ordering.* Both are well-founded. The double multiset ordering is defined on the finite *multisets* of finite multisets of natural numbers, but we shall use it only on the finite *sets* of finite multisets of natural numbers. The following lemma will be of help when using the double multiset ordering.

Lemma 2.8 *Let X and Y be two finite sets of finite multisets of natural numbers. Suppose that*

$$\forall x \in X\, \exists y \in Y\, (y \textit{ majorizes } x),$$

where y majorizes *x means that x is smaller than y in the multiset ordering.*

Then X is smaller than Y in the double multiset ordering.

Proof. We call an element $y \in Y$ *majorizing* if it majorizes some $x \in X$. X can be obtained from Y by first replacing each majorizing $y \in Y$ by the elements of X it majorizes and then removing from Y the non-majorizing elements. This proves the claim. $\square$

Below we use the notation $\text{bag}(a_1, \ldots, a_n)$ to denote the multiset whose elements are $a_1, \ldots, a_n$.

Boundedness

Another important concept is boundedness. It allows us to identify goals from which no divergence can arise. Recall that an atom A is called *bounded* w.r.t. a level mapping $|\,|$ if $|\,|$ is bounded on the set $[A]$ of ground instances of A. If A is bounded, then $|[A]|$ denotes the maximum that $|\,|$ takes on $[A]$. Note that every ground atom is bounded.

Our concept of a bounded goal differs from that of Bezem [Bez89] (see Definition 2.2(ii)) in that it takes into account the model I. This results in a more complicated definition.

Definition 2.9 Let P be a program, $|\,|$ a level mapping for P, I a model of P and $k \geq 0$.

(i) With each ground goal $G = \leftarrow A_1, \ldots, A_n$ we associate a finite multiset $|G|_I$ of natural numbers defined by

$$|G|_I = \text{bag}\,(|A_1|, \ldots, |A_{\bar{n}}|),$$

where

$$\bar{n} = \min(\{n\} \cup \{i \in [1, n] \mid I \not\models A_i\}).$$

(ii) With each goal G we associate a set of multisets $|[G]|_I$ defined by

$$|[G]|_I = \{|G'|_I \mid G' \text{ is a ground instance of } G\}.$$

(iii) A goal G is called *bounded by k* w.r.t. $|\,|$ and I if $k \geq \ell$ for $\ell \in \cup|[G]|_I$.

A goal is called *bounded* w.r.t. $|\,|$ and I if it is bounded by some $k \geq 0$ w.r.t. $|\,|$ and I.

□

It is useful to note the following.

Lemma 2.10 *Let P be a program, $|\,|$ a level mapping for P and I a model of P. A goal G is bounded w.r.t. $|\,|$ and I iff the set $|[G]|_I$ is finite.*

Proof. Consider a goal G that is bounded by some k. Suppose that G has n atoms. Then each element of $|[G]|_I$ is a multiset of at most n numbers selected from $[0, k]$. The number of such multisets is finite.

The other implication is obvious. □

For goals with one atom it is often easy to establish boundedness by proving a stronger property.

Definition 2.11 Let $|\,|$ be a level mapping. A goal $\leftarrow A$ is called *rigid* w.r.t. $|\,|$ if $|\,|$ is constant on the set $[A]$ of ground instances of A.

This definition is inspired by the considerations of Bossi, Cocco and Fabris [BCF91] where level mappings applied to non-ground atoms are studied. Obviously, rigid goals are bounded.

The following lemma is an analogue of Lemma 2.5 of Bezem [Bez89].

Lemma 2.12 *Let P be a program that is acceptable w.r.t. a level mapping $|\,|$ and a model I. Let G be a goal that is bounded (w.r.t. $|\,|$ and I) and let H be an LD-resolvent of G from P. Then*

(i) H is bounded,

(ii) $|[H]|_I$ is smaller than $|[G]|_I$ in the double multiset ordering.

Proof. Let $G = \leftarrow A_1, \ldots, A_n$ $(n \geq 1)$. For some input clause $C = A \leftarrow B_1, \ldots, B_k$ $(k \geq 0)$ and mgu θ of A and A_1, $H = \leftarrow (B_1, \ldots, B_k, A_2, \ldots, A_n)\theta$.

First we show that for every ground instance H_0 of H there exists a ground instance G' of G such that $|H_0|_I$ is smaller that $|G'|_I$ in the multiset ordering.

So let H_0 be a ground instance of H. For some substitution δ

$$H_0 = \leftarrow B'_1, \ldots, B'_k, A'_2, \ldots, A'_n$$

and A'_1 is ground, where for brevity for any atom, clause or goal B, B' denotes $B\theta\delta$. Note that

$$C' = A'_1 \leftarrow B'_1, \ldots, B'_k$$

and

$$G' = \leftarrow A'_1, \ldots, A'_n,$$

since $A' = A'_1$ as $A\theta = A_1\theta$.

Case 1 For $i \in [1, k]$ $I \models B'_i$.
Then
$$|H_0|_I = \text{ bag } (|B'_1|, \ldots, |B'_k|,\ |A'_2|, \ldots, |A'_{\bar{n}}|)$$

where
$$\bar{n} = \min(\{n\} \cup \{i \in [2, n] \mid I \not\models A'_i\}).$$

Additionally, since I is a model of P, $I \models A'_1$. Thus
$$|G'|_I = \text{ bag } (|A'_1|,\ |A'_2|, \ldots, |A'_{\bar{n}}|).$$

This means that $|H_0|_I$ is obtained from $|G'|_I$ by replacing $|A'_1|$ by $|B'_1|, \ldots, |B'_k|$. But by the definition of acceptability
$$|B'_i| \ < \ |A'_1|$$

for $i \in [1, k]$, so $|H_0|_I$ is smaller than $|G'|_I$ in the multiset ordering. □

Case 2 For some $i \in [1, k]$ $I \not\models B'_i$.
Then
$$|H_0|_I = \text{ bag}(|B'_1|, \ldots, |B'_{\bar{k}}|)$$

where
$$\bar{k} = \min(\{i \in [1, k] \mid I \not\models B'_i\}).$$

Also by the definition of acceptability
$$|B'_i| \ < \ |A'_1|$$

for $i \in [1, \bar{k}]$, so $|H_0|_I$ is smaller than $|G'|_I$ in the multiset ordering. □

This implies claim (i) since G is bounded. By Lemma 2.10 $|[H]|_I$ is finite and claim (ii) now follows by Lemma 2.8. □

Corollary 2.13 *Let P be an acceptable program and G a bounded goal. Then all LD-derivations of $P \cup \{G\}$ are finite.*

Proof. The double multiset ordering is well-founded. □

Corollary 2.14 *Every acceptable program is left terminating.*

Proof. Every ground goal is bounded. □

LD-trees

To prove the converse of Corollary 2.14 we analyze the size of finite LD-trees. To this end we need the following lemma, where $nodes_P(G)$ for a program P and a goal G denotes the number of nodes in the LD-tree for $P \cup \{G\}$.

Lemma 2.15 *Let P be a program and G a goal such that the LD-tree for $P \cup \{G\}$ is finite. Then*

(i) for all substitutions θ, $nodes_P(G\theta) \le nodes_P(G)$,

(ii) for all prefixes H of G, $nodes_P(H) \le nodes_P(G)$,

(iii) for all non-root nodes H in the LD-tree for $P \cup \{G\}$, $nodes_P(H) < nodes_P(G)$.

Proof.
(i) By an application of a variant of the Lifting Lemma (see e.g. Lloyd [Llo87]) to LD-derivations we conclude that to every LD-derivation of $P \cup \{G\theta\}$ with input clauses $C_1, C_2, \ldots$, there corresponds an LD-derivation of $P \cup \{G\}$ with input clauses $C_1, C_2, \ldots$ of the same or larger length. This implies the claim.
(ii) Consider a prefix $H = \leftarrow A_1, \ldots, A_k$ of $G = \leftarrow A_1, \ldots, A_n$ $(n \ge k)$. By an appropriate renaming of variables (formally justified by the Variant Lemma 2.8 in Apt [Apt90]) we can assume that all input clauses used in the LD-tree for $P \cup \{H\}$ have no variables in common with G. We can now transform the LD-tree for $P \cup \{H\}$ into an initial subtree of the LD-tree for $P \cup \{G\}$ by replacing in it a node $\leftarrow B_1, \ldots, B_l$ by $\leftarrow B_1, \ldots, B_l, A_{k+1}\theta, \ldots, A_n\theta$, where θ is the composition of the mgu's used on the path from the root H to the node $\leftarrow B_1, \ldots, B_l$. This implies the claim.
(iii) Immediate by the definition. □

As stated at the beginning of Section 2, we are interested in proving not only left termination of a program, but also its termination for a class of non-ground goals. We now show that the concepts of acceptability and boundedness provide us with a complete method for proving both properties.

Theorem 2.16 *Let P be a left terminating program. Then for some level mapping $|\ |$ and a model I of P*

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

(ii) for every goal G, G is bounded w.r.t. $|\ |$ and I iff all LD-derivations of $P \cup \{G\}$ are finite.

Proof. Define the level mapping by putting for $A \in B_P$

$$|A| = nodes_P\ (\leftarrow A).$$

Since P is left terminating, this level mapping is well defined. Next, choose

$$I = \{A \in B_P \mid \text{there is an LD-refutation of } P \cup \{\leftarrow A\}\}.$$

By the strong completeness of SLD-resolution, $I = M_P$, so I is a model of P.
First we prove one implication of (ii).

(ii1) Consider a goal G such that all LD-derivations of $P \cup \{G\}$ are finite. We prove that G is bounded by $nodes_P(G)$ w.r.t. $|\,|$ and I.

To this end take $\ell \in \cup|[G]|_I$. For some ground instance $\leftarrow A_1, \ldots, A_n$ of G and $i \in [1, \bar{n}]$, where

$$\bar{n} = \min(\{n\} \cup \{i \in [1,n] \mid I \not\models A_i\}),$$

we have $\ell = |A_i|$. We now calculate

$$\begin{array}{ll}
 & nodes_P(G) \\
\geq & \quad \{\text{Lemma 2.15 (i)}\} \\
 & nodes_P(\leftarrow A_1, \ldots, A_n) \\
\geq & \quad \{\text{Lemma 2.15 (ii)}\} \\
 & nodes_P(\leftarrow A_1, \ldots, A_{\bar{n}}) \\
\geq & \quad \{\text{Lemma 2.15 (iii), noting that for } j \in [1, \bar{n}-1] \\
 & \quad \text{there is an LD-refutation of } P \cup \{\leftarrow A_1, \ldots, A_j\}\} \\
 & nodes_P(\leftarrow A_i, \ldots, A_{\bar{n}}) \\
\geq & \quad \{\text{Lemma 2.15 (ii)}\} \\
 & nodes_P\ (\leftarrow A_i) \\
= & \quad \{\text{definition of } |\,|\} \\
 & |A_i| \\
= & \ell.
\end{array}$$

(i) We now prove that P is acceptable w.r.t. $|\,|$ and I. Take a clause $A \leftarrow B_1, \ldots, B_n$ in P and its ground instance $A\theta \leftarrow B_1\theta, \ldots, B_n\theta$. We need to show that

$$|A\theta| > |B_i\theta| \text{ for } i \in [1, \bar{n}],$$

where

$$\bar{n} = \min(\{n\} \cup \{i \in [1,n] \mid I \not\models B_i\theta\}).$$

We have $A\theta\theta \equiv A\theta$, so $A\theta$ and A unify. Let $\mu = \text{mgu}(A\theta, A)$. Then $\theta = \mu\delta$ for some δ. By the definition of LD-resolution, $\leftarrow B_1\mu, \ldots, B_n\mu$ is an LD-resolvent of $\leftarrow A\theta$.

Then for $i \in [1, \bar{n}]$

$$\begin{array}{ll}
 & |A\theta| \\
= & \quad \{\text{definition of } |\,|\} \\
 & nodes_P\ (\leftarrow A\theta) \\
> & \quad \{\text{Lemma 2.15 (iii)}, \leftarrow B_1\mu, \ldots, B_n\mu \text{ is a resolvent of } \leftarrow A\theta\} \\
 & nodes_P\ (\leftarrow B_1\mu, \ldots, B_n\mu) \\
\geq & \quad \{\text{part (ii1), noting that } B_i\theta \in \cup|[\leftarrow B_1\mu, \ldots, B_n\mu]|_I\} \\
 & |B_i\theta|.
\end{array}$$

(ii2) Consider a goal G which is bounded w.r.t. $|\ |$ and I. Then by (i) and Corollary 2.13 all LD-derivations of $P \cup \{G\}$ are finite. $\square$

Corollary 2.17 *A program is left terminating iff it is acceptable.*

Proof. By Corollary 2.14 and Theorem 2.16. $\square$

3 Example – Quicksort

The equivalence between the left terminating and acceptable programs provides us with a method of proving termination of Prolog programs. The level mapping and the model used in the proof of Theorem 2.16 were quite involved and relied on elaborate information about the program at hand which is usually not readily available. However, in practical situations much simpler constructions suffice. The level mapping can be usually defined as a simple function of the terms of the ground atom and the model takes into account only some straightforward information about the program. We illustrate it by means of an example.

First, we define by structural induction a function $|\ |$ on ground terms by putting:

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

It is useful to note that for a list xs, $|xs|$ equals its length. The function $|\ |$ is called *listsize* in Ullman and Van Gelder [UvG88].

Consider now the following program `QS` (for quicksort):

```
(qs1)   qs([], []) ←.
(qs2)   qs([X | Xs], Ys) ←
          f(X, Xs, X1s, X2s),
          qs(X1s, Y1s),
          qs(X2s, Y2s),
          a(Y1s, [X | Y2s], Ys).

(f1)    f(X, [], [], []) ←.
(f2)    f(X, [Y | Xs], [Y | Y1s], Y2s) ←
          X > Y,
          f(X, Xs, Y1s, Y2s).
(f3)    f(X, [Y | Xs], Y1s, [Y | Y2s]) ←
          X ≤ Y,
          f(X, Xs, Y1s, Y2s).

(a1)    a([], Ys, Ys) ←.
(a2)    a([X | Xs], Ys, [X | Zs]) ←
          a(Xs, Ys, Zs).
```

We assume that `QS` operates on the domain of natural numbers over which the built-in relations $>$ and $\leq$, written in infix notation, are defined. We thus assume that this domain is part of the Herbrand universe of `QS`.

Denote now the program consisting of the clauses $(f_1), (f_2), (f_3)$ by `filter`, and the program consisting of the clauses $(a_1), (a_2)$ by `append`.

Lemma 3.1 `filter` *is recurrent with* $|f(x, xs, x1s, x2s)| = |xs|$. □

We adopted here the simplifying assumption that built-in's $>$ and $\leq$ are recurrent w.r.t. the level mapping $|s > t| = 0$ and $|s \leq t| = 0$.

Lemma 3.2 `append` *is recurrent with* $|a(xs, ys, zs)| = |xs|$. □

Lemma 3.3 `QS` *is not recurrent.*

Proof. Consider clause (qs_2) instantiated with the ground substitution

$$\{X/a,\ Xs/b,\ Ys/c,\ X1s/[a|b],\ Y1s/c\}.$$

Then the ground atom $qs([a|b], c)$ appears both in the head and the body of the resulting clause. □

To prove that `QS` is left terminating we show that it is acceptable. We define an appropriate level mapping $|\ |$ by extending the ones given in Lemmata 3.1 and 3.2 with

$$|qs(xs, ys)| = |xs|.$$

Next, we define a Herbrand interpretation of `QS` by putting

$$\begin{array}{rcl} I & = & \{qs(xs, ys) \mid |xs| = |ys|\} \\ & \cup & \{f(x, xs, y1s, y2s) \mid |xs| = |y1s| + |y2s|\} \\ & \cup & \{a(xs, ys, zs) \mid |xs| + |ys| = |zs|\} \\ & \cup & [X > Y] \\ & \cup & [X \leq Y]. \end{array}$$

Recall that $[A]$ for an atom A stands for the set of all ground instances of A.

Lemma 3.4 I *is a model of* `QS`.

Proof. First, note that $|[]| + |ys| = |ys|$ and that $|xs| + |ys| = |zs|$ implies $|[x|xs]| + |ys| = |[x|zs]|$. This implies that I is a model of `append`.

Next, note that $|[]| + |[]| = |[]|$ and that $|xs| = |y1s| + |y2s|$ implies $|[y|xs]| = |[y|y1s]| + |y2s|$ and $|[y|xs]| = |y1s| + |[y|y2s]|$. This implies that I is a model of `filter`.

Finally, note that $|[\,]| = |[\,]|$ and that $|xs| = |x1s| + |x2s|$, $|x1s| = |y1s|$, $|x2s| = |y2s|$ and $|y1s| + |[x|y2s]| = |ys|$ imply $|[x|xs]| = |ys|$. This implies that I is a model of `QS`. □

We now prove the desired result.

Theorem 3.5 `QS` *is acceptable w.r.t.* $|\ |$ *and* I.

Proof. As `filter` and `append` are recurrent w.r.t. $|\ |$, we only need to consider clauses (qs_1) and (qs_2). (qs_1) satisfies the appropriate requirement voidly.

Consider now a ground instance C of (qs_2). C is of the form $A \leftarrow B_1, B_2, B_3, B_4$. We now prove three facts which imply that C satisfies the appropriate requirement.

Fact 1 $|A| > |B_1|$.

Proof. Note that

$$|qs([x|xs],ys)]| = |[x|xs]| > |xs| = |f(x,xs,x1s,x2s)|.$$

□

Fact 2 Suppose $I \models B_1$. Then $|A| > |B_2|$ and $|A| > |B_3|$.

Proof. By assumption $|xs| = |x1s| + |x2s|$, so

$$|qs([x|xs],ys)| > |xs| \geq |x1s| = |qs(x1s,y1s)|$$

and analogously

$$|qs([x|xs],ys)| > |qs(x2s,y2s)|.$$

□

Fact 3 Suppose $I \models B_1$ and $I \models B_2$. Then $|A| > |B_4|$.

Proof. By Fact 2 $|qs([x|xs],ys)| > |qs(x1s,y1s)| = |x1s|$ and by assumption $|x1s| = |y1s|$, so

$$|qs([x|xs],ys)| > |y1s| = |a(y1s,[x|y2s],ys)|.$$

□
□

So far we only proved that `QS` is left terminating. We now prove that it terminates for a large class of goals.

Lemma 3.6 *For all terms $t, t_1, \ldots, t_k$, $k \geq 0$, a goal of the form*

$$\leftarrow qs([t_1,\ldots,t_k],t)$$

is rigid w.r.t. $|\;|$.

Proof. Let A be a ground instance of $qs([t_1,\ldots,t_k],t)$. Then $|A| = |[t_1,\ldots,t_k]| = k$. □

It is worth noting that every "ill typed" goal $\leftarrow qs(s,t)$, where s is a non-variable, non-list term is also rigid w.r.t. $|\;|$, as $|s'| = 0$ for every ground instance s' of s.

Corollary 3.7 *For all terms $t, t_1, \ldots, t_k$, $k \geq 0$, all LD-derivations of* `QS` $\cup \{\leftarrow qs([t_1,\ldots,t_k],t)\}$ *are finite.*

Proof. By Corollary 2.13. □

4 Left Termination of General Programs

We now address the problem of extending the notions of left termination and acceptability to general programs, i.e. programs that admit negative literals in clause bodies.

General Programs and LDNF-resolution

Recall that a *general clause* is a construct of the form

$$A \leftarrow L_1, \ldots, L_n$$

$(n \geq 0)$ where A is an atom and $L_1, \ldots, L_n$ are literals. In turn, a *general goal* is a construct of the form

$$\leftarrow L_1, \ldots, L_n$$

$(n \geq 0)$ where $L_1, \ldots, L_n$ are literals. A *general program* is a finite set of general clauses.

In this paper we consider SLDNF-resolution with one selection rule only – namely that of Prolog, that is the leftmost selection rule. As S in SLDNF stands for "selection rule", we denote this form of resolution by LDNF (*L*inear resolution for *D*efinite clauses with *N*egation as *F*ailure).

When studying termination of general Prolog programs, i.e. programs executed using the LDNF-resolution it is necessary to revise the standard definitions of Lloyd [Llo87]. Indeed, according to his definitions there is no LDNF-derivation for $\{p \leftarrow \neg p\} \cup \{\leftarrow p\}$ whereas the corresponding Prolog execution diverges.

The appropriate revision is achieved by viewing the LDNF-resolution as a top down interpreter which given a general program P and a general goal G attempts to build a search tree for $P \cup \{G\}$ by constructing its branches in parallel. The branches in this tree are called LDNF-derivations for $P \cup \{G\}$ and the tree itself is called the LDNF-tree for $P \cup \{G\}$.

Negative literals are resolved using the negation as failure rule which calls for the construction of a subsidiary search tree. If during this subsidiary construction the interpreter diverges, the main LDNF-derivation *is considered* to be infinite. Adopting this view the LDNF-derivation for $\{p \leftarrow \neg p\} \cup \{\leftarrow p\}$ diverges because the goal $\leftarrow p$ is resolved to $\leftarrow \neg p$ and the subsequent construction of the subsidiary LDNF-tree for $\{p \leftarrow \neg p\} \cup \{\leftarrow p\}$ diverges.

Recently Martelli and Tricomi [MT92], and later Apt and Doets [AD92], proposed two formalizations of the above revision of the (S)LDNF-resolution.

Summarizing, by termination of a general Prolog program we actually mean termination of the underlying interpreter. By choosing variables of the input clauses and the used mgu's in a fixed way we can assume that for every general program P and general goal G there exists exactly one LDNF-tree for $P \cup \{G\}$. The subsidiary LDNF-trees formed during the construction of this tree are called *subsidiary* LDNF-trees *for* $P \cup \{G\}$.

The following definition extends the notion of left termination to general programs.

Definition 4.1 A general program P is called *left terminating* if all LDNF-derivations of P starting in a ground general goal are finite. □

In other words, a general program is left terminating if all LDNF-trees for P with a ground root are finite. Again, our method of proving left termination will allow us to identify a class of terminating non-ground general goals which constitute the intended queries for the program.

The following lemma will be of use later.

Lemma 4.2 *Suppose that all LDNF-derivations of P starting in a ground goal are finite. Then P is left terminating.*

Proof. It suffices to show that for all ground literals L all LDNF-derivations of $P \cup \{\leftarrow L\}$ are finite. When L is positive it is a part of the assumptions and when L is negative, say $L = \neg A$, it follows from the fact that by assumption the subsidiary LDNF-tree for $P \cup \{\leftarrow A\}$ is finite. □

Acceptable General Programs

Our aim is to generalize the concept of acceptability of Section 2 to general Prolog programs. First, we extend in a natural way a level mapping to a mapping from ground literals to natural numbers by putting $|\neg A| = |A|$. Next, given a general program P, we define its subset P^-. In P^- we collect the definitions of the negated relations and relations on which these relations depend. More precisely, we define P^- as follows.

Definition 4.3 Let P be a general program and p, q relations.

(i) We say that p *refers to* q iff there is a general clause in P that uses p in its head and q in its body.

(ii) We say that p *depends on* q iff (p, q) is in the reflexive, transitive closure of the relation *refers to.*

□

Of course, not every relation needs to refer to itself, but by reflexivity every relation depends on itself.

Definition 4.4 Let P be a general program. Denote by Neg_P the set of relations in P which occur in a negative literal in a body of a general clause from P and by Neg^*_P the set of relations in P on which the relations in Neg_P depend. We define P^- to be the set of general clauses in P in whose head a relation from Neg^*_P occurs. □

We can now introduce the desired generalization of the notion of acceptability.

Definition 4.5 Let P be a general program, $|\ |$ a level mapping for P and I a model of P whose restriction to the relations from Neg^*_P is a model of $comp(P^-)$. P is called *acceptable with respect to* $|\ |$ *and* I if for every general clause $A \leftarrow L_1, \ldots, L_n$ in $ground(P)$ the following implication holds for $i \in [1, n]$:

$$\text{if } I \models \wedge_{j=1}^{i-1} L_j \text{ then } |A| > |L_i|.$$

In other words, we have for every general clause $A \leftarrow L_1, \ldots, L_n$ in $ground(P)$

$$|A| > |L_i| \text{ for } i \in [1, \bar{n}],$$

where

$$\bar{n} = \min(\{n\} \cup \{i \in [1, n] \mid I \not\models L_i\}).$$

P is called *acceptable* if it is acceptable with respect to some level mapping and a model of P whose restriction to the relations from Neg^*_P is a model of $comp(P^-)$. □

Note that for a program P we have $Neg^*_P = \emptyset$, so P^- is empty and the above definition coincides with the definition of acceptability for programs.

The idea underlying the definition of acceptability for general programs is similar to that of programs and can be illustrated as follows. Consider a general program P, a level mapping $|\ |$ for P and a model I of P whose restriction to the relations from Neg^*_P is a model of $comp(P^-)$, such that P is acceptable with respect to $|\ |$ and I. Let C be a ground instance of a general clause from

P, and $\neg A$ a negative literal in the body of C, such that $I \not\models \neg A$. By the fact that the restriction of I to the relations from Neg_P^* is a model of $comp(P^-)$, we have that $comp(P^-) \not\models \neg A$. This condition (by the soundness of SLDNF-resolution) excludes the existence of a refutation for $\neg A$, and consequently there is no point in checking that the level mapping decreases from the head of the general clause C to any literals occurring to the right of $\neg A$ in the body of C.

The concept of an acceptable general program also generalizes that of an acyclic program studied in Cavedon [Cav89] and Apt and Bezem [AB90].

Finally, note that this concept calls for the use of a model of $comp(P^-)$ and thus assumes consistency of $comp(P^-)$. This seems to indicate that its applicability is limited. However, we show below (Theorem 4.17) that in many cases left termination implies consistency of $comp(P)$.

Definition 4.6 Let P be a general program, $|\ |$ a level mapping for P. P is called *acyclic with respect to* $|\ |$ if for every general clause $A \leftarrow L_1, \ldots, L_n$ in $ground(P)$

$$|A| > |L_i| \text{ for } i \in [1, n].$$

P is called *acyclic* if it is acyclic with respect to some level mapping. □

Lemma 4.7 *Every acyclic program is acceptable.*

Proof. Let P be acyclic w.r.t. some level mapping $|\ |$. By Theorem 4.1 of Apt and Bezem [AB90] $comp(P)$ has a unique Herbrand model, M_P. Then P is acceptable w.r.t. $|\ |$ and M_P. □

As in the case of recurrent and acceptable programs, the use of the premise $I \models \wedge_{j=1}^{i-1} L_j$ forms the *only* difference between the concepts of acyclicity and acceptability. Apt and Bezem [AB90] proved among others that all SLDNF-derivations of an acyclic program starting in a ground general goal are finite. This implies that all acyclic programs are left terminating, so the concept of acyclicity is of obvious importance when studying termination of general Prolog programs. Indeed, in Apt and Bezem [AB90] the usefulness of this concept was demonstrated by proving termination of a general program which formalizes the Yale Shooting problem of Hanks and McDermott [HM87]. However, as we shall see in Section 5 of this paper, there exist natural left terminating general programs which are not acyclic. Thus the concept of acyclicity is of limited applicability when considering general Prolog programs.

Boundedness

The concept of boundedness also extends directly to literals and general programs. A literal L is called *bounded* w.r.t. a level mapping $|\ |$ if $|\ |$ is bounded on the set $[L]$ of ground instances of L. If L is bounded, then $|[L]|$ denotes the maximum that $|\ |$ takes on $[L]$. Note that every ground literal is bounded.

Our concept of a bounded general goal directly generalizes that of a bounded goal given in Definition 2.9.

Definition 4.8 Let P be a general program, $|\ |$ a level mapping for P, I a model of P whose restriction to the relations from Neg_P^* is a model of $comp(P^-)$, and $k \geq 0$.

(i) With each ground general goal $G = \leftarrow L_1, \ldots, L_n$ we associate a finite multiset $|G|_I$ of natural numbers defined by

$$|G|_I = \text{bag}(|L_1|, \ldots, |L_{\bar{n}}|),$$

where

$$\bar{n} = \min(\{n\} \cup \{i \in [1, n] \mid I \not\models L_i\}).$$

(ii) With each general goal G we associate a set of multisets $|[G]|_I$ defined by

$$|[G]|_I = \{|G'|_I \mid G' \text{ is a ground instance of } G\}.$$

(iii) A general goal G is called *bounded by* k w.r.t. $|\,|$ and I if $k \geq \ell$ for $\ell \in \cup|[G]|_I$, where $\cup|[G]|_I$ stands for the set-theoretic union of the elements of $|[G]|_I$.

(iv) A general goal is called *bounded* w.r.t. $|\,|$ and I if it is bounded by some $k \geq 0$ w.r.t. $|\,|$ and I.

□

Lemma 2.10 immediately extends to general programs.

Lemma 4.9 *Let P be a general program, $|\,|$ a level mapping for P and I a model of P whose restriction to the relations from Neg^*_P is a model of $comp(P^-)$. A general goal G is bounded w.r.t. $|\,|$ and I iff the set $|[G]|_I$ is finite.* □

The following lemma is an analogue of Lemma 2.12 for general programs.

Lemma 4.10 *Let P be a general program that is acceptable w.r.t. a level mapping $|\,|$ and an interpretation I. Let G be a general goal which is a descendant of a goal and which is bounded (w.r.t. $|\,|$ and I) and let H be an LDNF-resolvent of G from P. Then*

(i) H is bounded,

(ii) $|[H]|_I$ is smaller than $|[G]|_I$ in the double multiset ordering.

Proof. The proof is analogous to the proof of Lemma 2.12. Due to the presence of negative literals we only have to consider one additional case.

First we show that for every ground instance H_0 of H there exists a ground instance G' of G such that $|H_0|_I$ is smaller that $|G'|_I$ in the multiset ordering.

In the case that H is obtained from G by the proper resolution step, the proof is the same as in the proof of Lemma 2.12. Otherwise, H is obtained from G by the negation as failure rule. Let $G = \leftarrow L_1, \ldots, L_n$ $(n \geq 1)$. Then L_1 is a ground negative literal, say $L_1 = \neg A$, and $H = \leftarrow L_2, \ldots, L_n$.

Denote by T the finitely failed LDNF-tree for $P \cup \{\leftarrow A\}$. By the definition of Neg_P and the fact that G is a descendant of a goal, the relation occurring in A is in Neg_P. Thus all relations which occur in the general goals of the tree T are elements of Neg^*_P. So T is in fact a finitely failed LDNF-tree for $P^- \cup \{\leftarrow A\}$. By the soundness of the SLDNF-resolution, $comp(P^-) \models \neg A$, so $I \models L_1$.

Let H_0 be a ground instance of H. For some substitution δ

$$H_0 = \leftarrow L'_2, \ldots, L'_n,$$

where L'_i denotes $L_i\delta$. Thus

$$G' = \leftarrow L_1, L'_2, \ldots, L'_n,$$

is a ground instance of G. Then

$$|H_0|_I = \text{ bag } (|L'_2|, \ldots, |L'_{\bar{n}}|)$$

where

$$\bar{n} = \min(\{n\} \cup \{i \in [2,n] \mid I \not\models L'_i\}).$$

and, since $I \models L_1$,

$$|G'|_I = \text{ bag } (|L_1|,\ |L'_2|, \ldots, |L'_{\bar{n}}|).$$

This shows that $|H_0|_I$ is smaller than $|G'|_I$ in the multiset ordering.

The statement we just proved implies claim (i) since G is bounded. By Lemma 4.9 $|[H]|_I$ is finite and claim (ii) now follows by Lemma 2.8. □

Corollary 4.11 *Let P be an acceptable general program and G a bounded general goal. Then all LDNF-derivations of $P \cup \{G\}$ are finite.*

Proof. The double multiset ordering is well-founded. □

Corollary 4.12 *Every acceptable general program is left terminating.*

Proof. By the fact that every ground general goal is bounded, Corollary 4.11 and Lemma 4.2. □

Thus to prove that a general program is left terminating it suffices to show that it is acceptable.

To apply Corollaries 4.11 and 4.12 we need a method for verifying that an interpretation is a model of $comp(P^-)$. In the case of Herbrand interpretations this task becomes much simpler thanks to the following theorem due to Apt, Blair and Walker [ABW88]. Here an interpretation is *supported* if for all ground atoms A, $I \models A$ implies that for some general clause $A \leftarrow L_1, \ldots, L_n$ in $ground(P)$ we have $I \models L_1 \wedge \ldots \wedge L_n$.

Theorem 4.13 *A Herbrand interpretation I is a model of $comp(P)$ iff it is a supported model of P.* □

Non-floundering General Programs

The converse of Corollary 4.12 does not hold. This is in contrast to the case of programs. Below we say that an LDNF-derivation *flounders* if there occurs in it or in any of its subsidiary LDNF-trees a general goal with the first literal being non-ground and negative. An LDNF-tree is called *non-floundering* if none of its branches flounders.

Example 4.14 Consider the general program P which consists of only one general clause: $p(0) \leftarrow \neg p(X)$. Then the only LDNF-derivation of $P \cup \{\leftarrow p(0)\}$ flounders, so it is finite. By the definition of SLDNF-resolution the only LDNF-derivation of $P \cup \{\leftarrow \neg p(0)\}$ flounders, as well. Thus P is left terminating, since the only ground general goals are of the form $G = \leftarrow L_1, \ldots, L_n$ $(n \geq 1)$ where each L_i is either $p(0)$ or $\neg p(0)$. On the other hand P is not acceptable since $p(0) \leftarrow \neg p(0)$ is in $ground(P)$ and by definition for any level mapping $|p(0)| = |\neg p(0)|$. □

The above example exploits the fact that SLDNF-derivations may terminate by floundering. We now show that in the absence of floundering Corollary 4.12 can be reversed. We proceed analogously to the case of programs and first study the size of finite LDNF-trees. We need the following analogue of Lemma 2.15, where $nodes_P(G)$ for a general program P and a general goal G denotes the total number of nodes in the LDNF-tree for $P \cup \{G\}$ and in all the subsidiary LDNF-trees for $P \cup \{G\}$.

Lemma 4.15 *Let P be a general program and G a general goal such that the LDNF-tree for $P \cup \{G\}$ is finite and non-floundering. Then*

(i) for all substitutions θ, the LDNF-tree for $P \cup \{G\theta\}$ is finite and non-floundering and $nodes_P(G\theta) \ \leq \ nodes_P(G)$,

(ii) for all prefixes H of G, the LDNF-tree for $P \cup \{H\}$ is finite and non-floundering and $nodes_P(H) \ \leq \ nodes_P(G)$,

(iii) for all non-root nodes H in the LDNF-tree for $P \cup \{G\}$, $nodes_P(H) \ < \ nodes_P(G)$.

Proof. Because of the additional requirement of non-floundering the proof is more complicated than that of Lemma 2.15.
(i) The proof proceeds by structural induction on the LDNF-tree T for $P \cup \{G\}$.

The Base Case. Then T is formed by the only node G. The following three subcases arise.

Subcase 1 $G = \Box$. Then $G = G\theta$, and the claim trivially holds.
Subcase 2 $G = \ \leftarrow A, L_2, \ldots, L_k$. Then A does not unify with the head of any general clause in P and neither does $A\theta$. As a consequence, the general goal $G\theta$ also immediately fails, and the LDNF-tree T for $P \cup \{G\theta\}$ is formed by the only node $G\theta$.
Subcase 3 $G = \ \leftarrow \neg A, L_2, \ldots, L_k$. By the fact that T has no floundering derivation, A is ground. The general goal G immediately fails, so by the definition of the LDNF-resolution there is an LDNF-refutation of $P \cup \{\leftarrow A\}$. Then $G\theta$ also immediately fails as $A = A\theta$. Hence the LDNF-tree T for $P \cup \{G\theta\}$ is formed by the only node $G\theta$. By definition

$$nodes_P(G\theta) = 1 + nodes_P(\leftarrow A\theta) = 1 + nodes_P(\leftarrow A) = nodes_P(G).$$

The Induction Case. Two subcases arise here.

Subcase 1 $G = \ \leftarrow A, L_2, \ldots, L_k$. Assume that $H_1, \ldots, H_m$ are the resolvents of G from P. Consider $G\theta = \ \leftarrow (A, L_2, \ldots, L_k)\theta$, and let $H'_1, \ldots, H'_l$ be the resolvents of $G\theta$ from P. Clearly, for all i in $[1, l]$ there exist j in $[1, m]$ and a substitution δ such that $H'_i = H_j\delta$. By the induction hypothesis, $nodes_P(H'_i) \leq nodes_P(H_j)$. Hence:

$$\begin{aligned}&nodes_P(G\theta) = 1 + nodes_P(H'_1) + \ldots + nodes_P(H'_l) \leq \\ &1 + nodes_P(H_1) + \ldots + nodes_P(H_m) = nodes_P(G).\end{aligned}$$

Moreover, the LDNF-tree for $P \cup \{G\theta\}$ is finite and non-floundering since by the induction hypothesis the LDNF-trees for the resolvents of $G\theta$ are finite and non-floundering.
Subcase 2 $G = \ \leftarrow \neg A, L_2, \ldots, L_k$. By the fact that T has no floundering derivation, A is ground. The fact that G is not a terminal node in T implies that there exists an LDNF-refutation of $P \cup \{\leftarrow \neg A\}$, i.e. the LDNF-tree for $P \cup \{\leftarrow A\}$ is finitely failed. Then G has only one resolvent, namely $\leftarrow L_2, \ldots, L_k$. Moreover, $G\theta = \ \leftarrow \neg A, (L_2, \ldots, L_k)\theta$, since A is ground, so $\leftarrow (L_2, \ldots, L_k)\theta$ is the only resolvent of $G\theta$. By the induction hypothesis, $nodes_P(\leftarrow (L_2, \ldots, L_k)\theta) \leq nodes_P(\leftarrow L_2, \ldots, L_k)$. Hence:

$nodes_P(G\theta) = 1 + nodes_P(\leftarrow A) + nodes_P(\leftarrow (L_2, \ldots, L_k)\theta) \leq$
$1 + nodes_P(\leftarrow A) + nodes_P(\leftarrow L_2, \ldots, L_k) = nodes_P(G)$.

Moreover, the LDNF-tree for $P \cup \{G\theta\}$ is finite and non-floundering, since by the induction hypothesis the LDNF-tree for the resolvent of $G\theta$ is finite and non-floundering.

(ii) Consider a prefix $H = \leftarrow L_1, \ldots, L_k$ of $G = \leftarrow L_1, \ldots, L_n$ $(n \geq k)$. By an appropriate renaming of variables (formally justified by a straightforward extension to the LDNF-resolution of the Variant Lemma 2.8 in Apt [Apt90]) we can assume that all input general clauses used in the LDNF-tree for $P \cup \{H\}$ have no variables in common with G. We can now transform the LDNF-tree for $P \cup \{H\}$ into an initial subtree of the LDNF-tree for $P \cup \{G\}$ by replacing in it a node $\leftarrow M_1, \ldots, M_l$ by $\leftarrow M_1, \ldots, M_l, L_{k+1}\theta, \ldots, L_n\theta$, where θ is the composition of the mgu's used on the path from the root H to the node $\leftarrow M_1, \ldots, M_l$. This implies the claim, since every subsidiary LDNF-tree for $P \cup \{H\}$ is also a subsidiary LDNF-tree for $P \cup \{G\}$.

(iii) Immediate by the definition. □

This definition will now be useful.

Definition 4.16 We call a general program P *non-floundering* if no LDNF-derivation starting in a ground general goal flounders. □

The following result is of independent interest.

Theorem 4.17 *Let P be a left terminating, non-floundering general program. Then $comp(P)$ is consistent.*

Proof. Let

$$I = \{A \in B_P \mid \text{there is an LDNF-refutation of } P \cup \{\leftarrow A\}\}.$$

We show that I is a Herbrand model of $comp(P)$. To this end, we use Theorem 4.13 and show that I is a supported model of P.

To establish that I is a model of P, assume by contradiction that some ground instance $A \leftarrow L'_1, \ldots, L'_n$ of a general clause C from P is false in I. Then $I \models L'_1 \wedge \ldots \wedge L'_n$ and $I \not\models A$. Since P is left terminating and non-floundering, $I \not\models A$ implies that the LDNF-tree for $P \cup \{\leftarrow A\}$ is finitely failed and non-floundering.

For some ground substitution γ, $A = B\gamma$ where B is the head of the general clause C. Thus $A\gamma = B\gamma\gamma = B\gamma$, so A and B unify.

Let $\leftarrow L_1, \ldots, L_n$ be the resolvent of $\leftarrow A$ from the general clause C. The LDNF-tree for $P \cup \{\leftarrow L_1, \ldots, L_n\}$ is also finitely failed and non-floundering. As $L'_1, \ldots, L'_n = (L_1, \ldots, L_n)\theta$ for some substitution θ, we have by Lemma 4.15(i) that the LDNF-tree for $P \cup \{\leftarrow L'_1, \ldots, L'_n\}$ is non-floundering. Moreover, it is finitely failed, since a direct consequence of the proof of Lemma 4.15(i) is that the general goals present in the LDNF-tree for $P \cup \{\leftarrow L'_1, \ldots, L'_n\}$ are all instances of the general goals present in the LDNF-tree for $P \cup \{\leftarrow L_1, \ldots, L_n\}$. But the fact that the LDNF-tree for $P \cup \{\leftarrow L'_1, \ldots, L'_n\}$ is finitely failed and non-floundering contradicts the hypothesis that $I \models L'_1 \wedge \ldots \wedge L'_n$.

To establish that I is a supported interpretation of P, consider $A \in B_P$ such that $I \models A$, and let C be the first input general clause used in an LDNF-refutation of $P \cup \{\leftarrow A\}$. Let $\leftarrow L_1, \ldots, L_n$ be the resolvent of $\leftarrow A$ from the general clause C. Clearly, an LDNF-refutation for $P \cup \{\leftarrow L_1, \ldots, L_n\}$, with a computed answer substitution θ, can be extracted from the

LDNF-refutation of $P \cup \{\leftarrow A\}$. Let $L'_1, \ldots, L'_n$ be a ground instance of $(L_1, \ldots, L_n)\theta$. By a straightforward generalization of Lemma 3.20 in [Apt90] to the LDNF-resolution there exists an LDNF-refutation for $P \cup \{\leftarrow L'_1, \ldots, L'_n\}$. We conclude that $I \models L'_1 \wedge \ldots \wedge L'_n$. This establishes that I is a supported interpretation of P. □

We can now show that Corollary 4.12 can be reversed under the additional assumption of non-floundering, thus obtaining an analogue of Theorem 2.16 for general programs.

Theorem 4.18 *Let P be a left terminating, non-floundering general program. Then for some level mapping $|\ |$ and a model I of $comp(P)$*

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

(ii) for every general goal G, G is bounded w.r.t. $|\ |$ and I iff all LDNF-derivations of $P \cup \{G\}$ are finite.

Proof. The proof is similar to that of Theorem 2.16. Define the level mapping by putting for $A \in B_P$

$$|A| = nodes_P\,(\leftarrow A).$$

Since P is left terminating, this level mapping is well defined. Note that by definition, for $A \in B_P$

$$nodes_P(\leftarrow \neg A) > nodes_P(\leftarrow A) = |A| = |\neg A|,$$

so

$$nodes_P(\leftarrow \neg A) \geq |\neg A|. \tag{1}$$

Next, let I be the model of $comp(P)$ considered in the proof of Theorem 4.17, i.e.

$$I = \{A \in B_P \mid \text{there is an LDNF-refutation of } P \cup \{\leftarrow A\}\}.$$

First we prove one implication of (ii).

(ii1) Consider a general goal G such that all LDNF-derivations of $P \cup \{G\}$ are finite. We prove that G is bounded by $nodes_P(G)$ w.r.t. $|\ |$ and I.

To this end take $\ell \in \cup|[G]|_I$. For some ground instance $\leftarrow L_1, \ldots, L_n$ of G and $i \in [1, \bar{n}]$, where

$$\bar{n} = \min(\{n\} \cup \{i \in [1,n] \mid I \not\models L_i\}),$$

we have $\ell = |L_i|$. We now calculate

$$
\begin{array}{ll}
 & nodes_P(G) \\
\geq & \quad \{\text{Lemma 4.15 (i)}\} \\
 & nodes_P(\leftarrow L_1, \ldots, L_n) \\
\geq & \quad \{\text{Lemma 4.15 (ii)}\} \\
 & nodes_P(\leftarrow L_1, \ldots, L_{\bar{n}}) \\
\geq & \quad \{\text{Lemma 4.15 (iii), noting that for } j \in [1, \bar{n}-1] \\
 & \quad \text{there is an LDNF-refutation of } P \cup \{\leftarrow L_1, \ldots, L_j\}\}
\end{array}
$$

$$
\begin{array}{ll}
 & nodes_P(\leftarrow L_i, \ldots, L_{\bar{n}}) \\
\geq & \quad \{\text{Lemma 4.15 (ii)}\} \\
 & nodes_P\ (\leftarrow L_i) \\
\geq & \quad \{\text{definition of } |\ |, (1)\} \\
 & |L_i| \\
= & \ell.
\end{array}
$$

(i) We now prove that P is acceptable w.r.t. $||$ and I. I is a model of $comp(P)$, so the restriction of I to the relations in Neg_P^* is trivially a model of $comp(P^-)$. To complete the proof, take a general clause $A \leftarrow L_1, \ldots, L_n$ in P and its ground instance $A\theta \leftarrow L_1\theta, \ldots, L_n\theta$. We need to show that

$$|A\theta| > |L_i\theta| \text{ for } i \in [1, \bar{n}],$$

where

$$\bar{n} = \min(\{n\} \cup \{i \in [1,n] \mid I \not\models L_i\theta\}).$$

We have $A\theta\theta \equiv A\theta$, so $A\theta$ and A unify. Let $\mu = \text{mgu}(A\theta, A)$. Then $\theta = \mu\delta$ for some δ. By the definition of LDNF-resolution, $\leftarrow L_1\mu, \ldots, L_n\mu$ is an LDNF-resolvent of $\leftarrow A\theta$.

Then for $i \in [1, \bar{n}]$

$$
\begin{array}{ll}
 & |A\theta| \\
= & \quad \{\text{definition of } |\ |\} \\
 & nodes_P\ (\leftarrow A\theta) \\
> & \quad \{\text{Lemma 4.15(iii)}, \leftarrow L_1\mu, \ldots, L_n\mu \text{ is a resolvent of } \leftarrow A\theta\} \\
 & nodes_P\ (\leftarrow L_1\mu, \ldots, L_n\mu) \\
\geq & \quad \{\text{part (ii1), noting that } L_i\theta \in \cup|[\leftarrow L_1\mu, \ldots, L_n\mu]|_I\} \\
 & |L_i\theta|.
\end{array}
$$

(ii2) Consider a general goal G which is bounded w.r.t. $|\ |$ and I. Then by (i) and Corollary 4.10 all LDNF-derivations of $P \cup \{G\}$ are finite. □

Corollary 4.19 *A non-floundering general program is left terminating iff it is acceptable.*

Proof. By Corollary 4.12 and Theorem 4.18. □

5 Examples – the Game and Transitive Closure Programs

Theorem 4.18 shows that our method of proving termination based on the concepts of acceptability and boundedness is complete for left terminating, non-floundering general Prolog programs. In this section we illustrate its use by proving termination of two simple, well-known programs. None of them can be handled within the framework of Apt and Bezem [AB90].

A GAME Program

Suppose that $\mathcal{G}$ is an acyclic finite graph. Consider the following general program GAME:

```
win(X) ← move(X,Y), ¬ win(Y).
move(a,b) ←        for (a,b) ∈ G.
```

Lemma 5.1 GAME *is not acyclic.*

Proof. For any ground instance $win(a) \leftarrow move(a,a), \neg win(a)$ of the first general clause and a level mapping $|\ |$ we have $|win(a)| = |\neg win(a)|$. □

We now proceed to show that GAME is acceptable. Since $\mathcal{G}$ is acyclic and finite, there exists a function f from the elements of its domain to natural numbers such that for $a \in dom(\mathcal{G})$

$$f(a) = \begin{cases} 0 & \text{if for no } b,\ (a,b) \in \mathcal{G} \\ 1 + max\ \{f(b) \mid (a,b) \in \mathcal{G}\} & \text{otherwise.} \end{cases}$$

We define appropriate level mapping by putting for all $(a,b) \in dom(\mathcal{G})$

$$|move(a,b)| = f(a)$$

and for $a \in dom(\mathcal{G})$

$$|win(a)| = f(a) + 1.$$

Next, since $\mathcal{G}$ is acyclic and finite, there exists a function g from the elements of its domain to $\{0,1\}$ such that for $a \in dom(\mathcal{G})$

$$g(a) = \begin{cases} 0 & \text{if for no } b,\ (a,b) \in \mathcal{G} \\ 1 - min\ \{g(b) \mid (a,b) \in \mathcal{G}\} & \text{otherwise.} \end{cases}$$

Let

$$\begin{aligned} I \ = & \quad \{move(a,b) \mid\ (a,b) \in \mathcal{G}\} \\ & \cup\ \{win(a) \mid\ g(a) = 1\}. \end{aligned}$$

Lemma 5.2 *I is a model of comp(*GAME*).*

Proof. The following two statements hold.
(a) I is a model of GAME.
Indeed, consider a ground instance

$$win(a) \leftarrow move(a,b), \neg win(b)$$

of the first general clause of GAME and suppose that

$$I \models move(a,b)\ \wedge\ \neg win(b).$$

Then $(a,b) \in \mathcal{G}$ and $g(b) = 0$, so $g(a) = 1$ and consequently

$$I \models win(a).$$

Additionally, I is a model for all `move` clauses.

(b) I is a supported interpretation of `GAME`.

Indeed, consider an atom $win(a) \in I$. Then $g(a) = 1$, so for some $b \in \mathcal{G}$ we have $(a, b) \in \mathcal{G}$ and $g(b) = 0$. We conclude that

$$I \models move(a, b) \wedge \neg win(b).$$

By Theorem 4.13 we conclude that I is a model of $comp($`GAME`$)$. □

We can now prove the desired result.

Theorem 5.3 `GAME` *is acceptable w.r.t.* $|\,|$ *and* I.

Proof. For a general program P every model of $comp(P)$ is also a model of P, thus I is a model of `GAME`. Moreover, $\texttt{GAME}^- = \texttt{GAME}$.

Consider a ground instance

$$win(a) \leftarrow move(a, b), \neg win(b)$$

of the first general clause of `GAME`. Then by definition

$$|win(a)| = f(a) + 1 > f(a) = |move(a, b)|.$$

Suppose now that $I \models move(a, b)$. Then $move(a, b) \in I$, so $(a, b) \in \mathcal{G}$ and consequently $f(a) > f(b)$. Thus

$$|win(a)| = f(a) + 1 > f(b) + 1 = |\neg win(b)|.$$

□

Corollary 5.4 `GAME` *is left terminating.*

Proof. By Corollary 4.12. □

Corollary 5.5 *For all terms* t, *the goal* $\leftarrow win(t)$ *is bounded w.r.t.* $|\,|$ *and* I.

Proof. The goal $\leftarrow win(t)$ is bounded by $max\ \{f(a) + 1 \mid a \in dom(\mathcal{G})\}$. Note that because of the syntax of `GAME`, t is either a variable or a constant. In the latter case we can improve the bound to $f(t) + 1$. □

Corollary 5.6 *For all terms* t, *all LDNF-derivations of* $\texttt{GAME} \cup \{\leftarrow win(t)\}$ *are finite.*

Proof. By Corollary 4.11. □

Transitive Closure

Consider the following general program computing the transitive closure of a graph.

```
(r1)  r(X,Y,E,V) ←
        member([X,Y],E).
(r2)  r(X,Z,E,V) ←
        member([X,Y],E),
        ¬ member(Y,V),
        r(Y,Z,E,[Y|V]).

(m1)  member(X,[X|T]) ←.
(m2)  member(X,[Y|T]) ←
        member(X,T).
```

In a typical use of this program one evaluates a goal $\leftarrow r(x, y, e, [x])$ where x, y are nodes and e is a graph specified by a list of its edges. The nodes of e belong to a finite set $\mathcal{A}$. This goal is supposed to succeed when $[x, y]$ is in the transitive closure of e. The last argument of $r(x, y, e, v)$ acts as an accumulator in which one maintains the list of nodes which should not be reused when looking for a path connecting x with y in e (to keep the path acyclic).

To ensure that the elements of $\mathcal{A}$ are in the Herbrand Universe of the program we add to the program the clauses

```
(e)   element(a) ←        for a ∈ A,
```

and call the resulting general program `TRANS`.

Lemma 5.7 `TRANS` *is not acyclic.*

Proof. By Lemma 4.1 of Apt and Bezem [AB90] all SLDNF-derivations of an acyclic program P starting with a ground goal are finite. Thus it suffices to exhibit an infinite SLDNF-derivation of `TRANS` starting in a ground goal. Such a derivation is obtained by using the rightmost selection rule and starting with the ground goal $\leftarrow r(x, z, e, v)$ repeatedly using general clause (r_2). □

We now prove that `TRANS` is acceptable. Below we call a list consisting of two elements a *pair*.

First, we define by structural induction a function *set* by putting

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

Then for a list xs, $set(xs)$ is the set of its elements.

Define now a Herbrand interpretation I by

$$I = [r(X, Y, E, V)] \cup I_1 \cup \{element(x) \mid x \in \mathcal{A}\}$$

where

$$I_1 = \{member(x, xs) \mid x \in set(xs)\}.$$

Recall that for an atom A, $[A]$ stands for the set of all ground instances of A.

We now prove two lemmata about I and I_1.

Lemma 5.8 *I is a model of* TRANS.

Proof. I is clearly a model of (r_1), (r_2) and of the clauses (e). I is also a model of the clauses (m_1) and (m_2) because by definition $x \in set([x|t])$ holds and $x \in set(t)$ implies $x \in set([y|t])$. □

Lemma 5.9 *I_1 is a model of $comp(\texttt{TRANS}^-)$.*

Proof. Note that $\texttt{TRANS}^- = \{(m_1), (m_2)\}$. We prove that I_1 is a supported interpretation of $\{(m_1), (m_2)\}$. Consider an atom $member(x, xs) \in I_1$. We prove that there exists a ground instance $member(x, xs) \leftarrow L_1, \ldots, L_n$ of (m_1) or (m_2) such that $I \models L_1 \wedge \ldots \wedge L_n$.

By definition $x \in set(xs)$, so for some y and t we have $xs = [y|t]$ and $x \in \{y\} \cup set(t)$. If $x = y$, then $xs = [x|t]$, and the desired clause is an instance of (m_1). Otherwise $x \in set(t)$, so $member(x, t) \in I$, i.e. $I \models member(x, t)$. In this case the desired clause is an instance of (m_2).

By Lemma 5.8 I_1 is a model of $\{(m_1), (m_2)\}$, so by Theorem 4.13 we now conclude that I_1 is a model of $comp(\{(m_1), (m_2)\})$. □

We now define an appropriate level mapping. We use here the listsize function $|\,|$ which maps ground terms to natural numbers and is defined in Section 3. It is clear that by putting

$$|member(x, y)| = |y|$$

we obtain the desired decrease for clause (m_2). Having made this choice in order to obtain the desired decrease for clause (r_1) we need to have

$$|r(x, z, e, v)| > |e|. \tag{2}$$

Additionally, to obtain the desired decrease for general clause (r_2) we need to have (assuming that $I \models member([x, y], e)$)

$$|r(x, z, e, v)| > |v| \tag{3}$$

and, assuming

$$I \models member([x, y], e) \wedge \neg member(y, v), \tag{4}$$

we need to prove

$$|r(x, z, e, v)| > |r(y, z, e, [y|v])|. \tag{5}$$

To define $|r(x, z, e, v)|$ we first define two auxiliary functions. Let

$$nodes(e) = \{x \mid \text{for some pair } b,\ x \in set(b) \ \text{ and } \ b \in set(e)\}.$$

If e is a list of pairs that specifies the edges of a graph $\mathcal{G}$, then $nodes(e)$ is the set of nodes of $\mathcal{G}$.

Let

$$out(e, v) = \{x \mid x \in nodes(e) \ \text{ and } x \notin set(v)\}.$$

If e is a list of pairs that specify the edges of a graph $\mathcal{G}$ and v is a list, then $out(e, v)$ is the set of nodes of $\mathcal{G}$ that are not elements of v.

We now put

$$|r(x,z,e,v)| = |e| + |v| + 2 \cdot card\ out(e,v) + 1,$$

where $card\ X$ stands for the cardinality of the set X.

Then (2) and (3) hold. Assume now (4). Then $[x,y] \in set(e)$ and $y \notin set(v)$. Thus $y \in nodes(e)$ and consequently $y \in out(e,v)$.

On the other hand $set([y|v]) = \{y\} \cup set(v)$. Thus $y \notin out(e,[y|v])$ and $out(e,v) = \{y\} \cup out(e,[y|v])$ so $card\ out(e,v) = card\ out(e,[y|v]) + 1$.

We now have

$$\begin{aligned} |r(x,z,e,v)| &= |e| + |v| + 2 \cdot card\ out(e,v) + 1 \\ &= |e| + |v| + 2 \cdot card\ out(e,[y|v]) + 3 \\ &> |e| + |[y|v]| + 2 \cdot card\ out(e,[y|v]) + 1 \\ &= |r(y,z,e,[y|v])| \end{aligned}$$

which proves (5).

Summarizing, we proved the following result.

Theorem 5.10 `TRANS` *is acceptable w.r.t.* $|\ |$ *and* I. □

Corollary 5.11 `TRANS` *is left terminating.*

Proof. By Corollary 4.12. □

Corollary 5.12 *For all terms* x, y *and lists* e, v*, the goal* $\leftarrow r(x,y,e,v)$ *is rigid w.r.t.* $|\ |$.

Proof. For any ground instance A of $r(x,y,e,v)$ we have $|A| = |e| + |v| + 2 \cdot card\ out(e,v) + 1$. □

Corollary 5.13 *For all terms* x, y *and lists* e, v*, all LDNF-derivations of* `TRANS`$\cup\{\leftarrow r(x,y,e,v)\}$ *are finite.*

Proof. By Corollary 4.11. □

6 Semantic Considerations

In this section we study semantics of acceptable general programs. We show here that various ways of defining their semantics coincide.

We recall first the relevant definitions and results. Given a monotonic operator T on a complete partial ordering L with the least element $\perp$, we define the *upward ordinal powers* of T starting at $\perp$ in the standard way and denote them by $T \uparrow \alpha$ where α is an ordinal. If L has the greatest element, say $\top$, (this is the case when for example L is a complete lattice) we define the *downward ordinal powers* of T starting at $\top$ in the standard way and denote them by $T \downarrow \alpha$.

We use below Fitting's approach to the semantics of general programs. Fitting [Fit85] uses a 3-valued logic based on a logic due to Kleene [Kle52]. In Kleene's logic there are three truth values: **t** for true, **f** for false and **u** for undefined.

A Herbrand interpretation for this logic (called a 3-*valued* Herbrand interpretation) is defined as a pair (T, F) of disjoint sets of ground atoms. Given such an interpretation $I = (T, F)$ a ground atom A is true in I if $A \in T$, false in I if $A \in F$ and undefined otherwise; $\neg A$ is true in I if A is false in I and $\neg A$ is false in I if A is true in I.

Every binary connective takes the value **t** or **f** if it takes that value in 2-valued logic for all possible substitutions of **u**'s by **t** or **f**; otherwise it takes value **u**.

Given a formula ϕ and a 3-valued Herbrand interpretation I, we write ϕ is $true_3$ in I (respectively ϕ is $false_3$ in I) to denote the fact that ϕ is true in I (respectively that ϕ is false in I) in the above defined sense.

Given $I = (T, F)$ we denote T by I^+ and F by I^-. Thus $I = (I^+, I^-)$. If $I^+ \cup I^- = B_P$, we call I a *total* 3-valued Herbrand interpretation for the general program P.

Every (2-valued) Herbrand interpretation I for a general program P determines a total 3-valued Herbrand interpretation $(I, B_P - I)$ for P. This allows us to identify every 2-valued Herbrand interpretation I for a general program P with its 3-valued counterpart $(I, B_P - I)$. For uniformity, given a (2-valued) Herbrand interpretation I we write ϕ is $true_2$ in I instead of $I \models \phi$ and ϕ is $false_2$ in I instead of $I \not\models \phi$. The following proposition relates truth in 3- and 2-valued interpretations and will be useful later.

Proposition 6.1 *Let I be a 3-valued interpretation and L a literal. Then*

(i) L is $true_3$ in I implies L is $true_2$ in I^+,

(ii) L is $true_2$ in I^+ implies L is not $false_3$ in I, i.e. L is either $true_3$ or undefined in I.

Proof.
(i) If $L = A$, L is $true_3$ in I implies $A \in I^+$, hence A is $true_2$ in I^+. If $L = \neg A$, $\neg A$ is $true_3$ in I implies $A \in I^-$, which implies $A \notin I^+$. Hence $\neg A$ is $true_2$ in I^+.
(ii) If $L = A$, L is $true_2$ in I^+ implies $A \in I^+$, hence A is $true_3$ in I. If $L = \neg A$, $\neg A$ is $true_2$ in I^+ implies $A \notin I^+$. Hence $\neg A$ is either $true_3$ or undefined in I. $\square$

Given a general program P, the 3-valued Herbrand interpretations for P form a complete partial ordering with the ordering $\subseteq$ defined by

$$I \subseteq J \text{ iff } I^+ \subseteq J^+ \wedge I^- \subseteq J^-$$

and with the least element $(\emptyset, \emptyset)$. Note that in this ordering every total 3-valued Herbrand interpretation is $\subseteq$-maximal. Intuitively, $I \subseteq J$ if J decides both truth and falsity for more atoms than I does.

Following Fitting [Fit85], given a general program P we define an operator Φ_P on the complete partial ordering of 3-valued Herbrand interpretations for P as follows:

$$\Phi_P(I) = (T, F),$$

where

$$T = \{A \mid \text{for some } A \leftarrow L_1, \ldots, L_k \text{ in } ground(P),\ L_1 \wedge \ldots \wedge L_k \text{ is } true_3 \text{ in } I\},$$
$$F = \{A \mid \text{for all } A \leftarrow L_1, \ldots, L_k \text{ in } ground(P),\ L_1 \wedge \ldots \wedge L_k \text{ is } false_3 \text{ in } I\}.$$

It is easy to see that T and F are disjoint, so $\Phi_P(I)$ is indeed a 3-valued Herbrand interpretation. Φ_P is a natural generalization of the usual immediate consequence operator T_P to the case of 3-valued logic. Φ_P is easily seen to be monotonic.

The upward ordinal powers of Φ_P, denoted by $\Phi_P \uparrow \alpha$, are defined starting the iteration at the $\subseteq$-least 3-valued Herbrand interpretation, $(\emptyset, \emptyset)$. In particular

$$\Phi_P \uparrow \omega = \bigcup_{n<\omega} \Phi_P \uparrow n.$$

Before studying semantics of acceptable general programs we prove a number of auxiliary results about the operators T_P and Φ_P. The following lemma relates these two operators.

Lemma 6.2 *Let I be a 3-valued interpretation and P a general program. Then*

$$\Phi_P(I)^+ \subseteq T_P(I^+) \subseteq B_P - \Phi_P(I)^-.$$

Moreover, if I is total then $\Phi_P(I)^+ = T_P(I^+) = B_P - \Phi_P(I)^-$.

Proof. By definition of T_P and Φ_P we obtain:

$A \in \Phi_P(I)^+$	iff for some $A \leftarrow L_1, \ldots, L_k$ in $ground(P)$ $L_1 \wedge \ldots \wedge L_k$ is $true_3$ in I,
$A \in T_P(I^+)$	iff for some $A \leftarrow L_1, \ldots, L_k$ in $ground(P)$ $L_1 \wedge \ldots \wedge L_k$ is $true_2$ in I^+,
$A \in B_P - \Phi_P(I)^-$	iff for some $A \leftarrow L_1, \ldots, L_k$ in $ground(P)$ $L_1 \wedge \ldots \wedge L_k$ is not $false_3$ in I.

Hence, the implication $A \in \Phi_P(I)^+ \Rightarrow A \in T_P(I^+)$ (respectively $A \in T_P(I^+) \Rightarrow A \in B_P - \Phi_P(I)^-$) directly follows from Proposition 6.1(i) (respectively Proposition 6.1(ii)).

If I is total, then $L_1 \wedge \ldots \wedge L_k$ is $true_3$ in I iff $L_1 \wedge \ldots \wedge L_k$ is $true_2$ in I^+ iff $L_1 \wedge \ldots \wedge L_k$ is not $false_3$ in I. □

The following corollaries relate the fixpoints of the operators T_P and Φ_P.

Corollary 6.3 *Let $I = (I^+, B_P - I^+)$ be a total 3-valued interpretation and P a general program. Then I^+ is a fixpoint of T_P if and only if I is a fixpoint of Φ_P.*

Proof.
($\Rightarrow$) Assume $I^+ = T_P(I^+)$. By Lemma 6.2 we have $\Phi_P(I)^+ = T_P(I^+) = B_P - \Phi_P(I)^-$. Hence $I^+ = \Phi_P(I)^+$ and $I^- = B_P - I^+ = \Phi_P(I)^-$, i.e. $I = \Phi_P(I)$.
($\Leftarrow$) Assume $I = \Phi_P(I)$. Then by Lemma 6.2 we have

$$I^+ = \Phi_P(I)^+ \subseteq T_P(I^+) \subseteq B_P - \Phi_P(I)^- = B_P - I^- = I^+.$$

Hence I^+ is a fixpoint of T_P. □

Corollary 6.4 *If Φ_P has exactly one fixpoint I and I is total, then I^+ is the unique fixpoint of T_P.*

Proof. By Corollary 6.3. □

The fixpoints of the operator T_P are of interest for us because of the following result of Apt, Blair and Walker [ABW88].

Theorem 6.5 *A Herbrand interpretation I is a model of $comp(P)$ iff it is a fixpoint of T_P.* □

Corollary 6.6 *If I is a Herbrand model of $comp(P)$ then $\Phi_P \uparrow \omega \subseteq (I, B_P - I)$.*

Proof. Suppose I is a Herbrand model of $comp(P)$. Then by Theorem 6.5 I is a fixpoint of T_P, so by Corollary 6.3 $(I, B_P - I)$ is fixpoint of Φ_P. By the monotonicity of Φ_P the least fixpoint of Φ_P, $lfp(\Phi_P)$, exists and $\Phi_P \uparrow \omega \subseteq lfp(\Phi_P)$. But $lfp(\Phi_P) \subseteq (I, B_P - I)$, so $\Phi_P \uparrow \omega \subseteq (I, B_P - I)$. $\square$

We are now ready to analyze the semantics of acceptable general programs.

Theorem 6.7 *Let P be an acceptable general program w.r.t. $|\,|$ and I. Then $\Phi_P \uparrow \omega$ is total.*

Proof. To establish that $\Phi_P \uparrow \omega$ is total we prove that, for $n \in \omega$ and $A \in B_P$, $|A| = n$ implies that A is not undefined in $\Phi_P \uparrow (n+1)$, i.e. A is either $true_3$ or $false_3$ in $\Phi_P \uparrow (n+1)$. The proof proceeds by induction on n. Fix $A \in B_P$.

In the base case we have $|A| = 0$ and since P is acceptable, two possibilities arise: (i) there is a unit clause $A \leftarrow$ in $ground(P)$ and (ii) there is no general clause in $ground(P)$ with A as conclusion. In case (i) A is $true_3$ in $\Phi_P \uparrow 1$, and in case (ii) A is $false_3$ in $\Phi_P \uparrow 1$.

In the induction case we have $|A| = n > 0$. Consider the set C_A of the general clauses in $ground(P)$ with A as conclusion. If C_A is empty then A is $false_3$ in $\Phi_P \uparrow 1$ and, by the monotonicity of Φ_P, it is $false_3$ in $\Phi_P \uparrow (n+1)$. If C_A is non-empty, take a general clause $A \leftarrow L_1, \ldots, L_k$ from C_A, and let $\bar{k} = \min(\{k\} \cup \{i \in [1,k] \mid L_i \text{ is } false_2 \text{ in } I\})$. We now prove that $L_1 \wedge \ldots \wedge L_k$ is not undefined in $\Phi_P \uparrow n$. To this end we consider two subcases.

Subcase 1. $\bar{k} = k$ and L_k is $true_2$ in I. Then, by the acceptability of P, $n = |A| > |L_k|$ for $i \in [1,k]$. By the induction hypothesis L_i is either $true_3$ or $false_3$ in $\Phi_P \uparrow n$, for $i \in [1,k]$.

Subcase 2. $\bar{k} \leq k$ and $L_{\bar{k}}$ is $false_2$ in I. Then $n = |A| > |L_k|$ for $i \in [1, \bar{k}]$. By the induction hypothesis, L_i is either $true_3$ or $false_3$ in $\Phi_P \uparrow n$, for $i \in [1, \bar{k}]$. Moreover, we claim that $L_{\bar{k}}$ is $false_3$ in $\Phi_P \uparrow n$. To establish this point, the following two possibilities have to be taken into account.

Suppose the relation occurring in $L_{\bar{k}}$ is in Neg^*_P. A simple proof by induction on n shows that $\Phi_P \uparrow n$ and $\Phi_{P^-} \uparrow n$ coincide on the relations in Neg^*_P. Thus $L_{\bar{k}}$ is $true_3$ in $\Phi_P \uparrow n$ implies $L_{\bar{k}}$ is $true_3$ in $\Phi_{P^-} \uparrow n$. Hence, by Corollary 6.6 and Proposition 6.1(i), $L_{\bar{k}}$ is $true_2$ in the restriction of I to the relations in Neg^*_P which is a model of $comp(P^-)$. This contradicts the fact that $L_{\bar{k}}$ is $false_2$ in I.

If the relation occurring in $L_{\bar{k}}$ is not in Neg^*_P, then $L_{\bar{k}}$ is a positive literal. We show that in this case $L_{\bar{k}}$ is $true_3$ in $\Phi_P \uparrow n$ implies $L_{\bar{k}}$ is $true_2$ in I by induction on the stage i at which $L_{\bar{k}}$ becomes $true_3$ in $\Phi_P \uparrow i$. For $i = 0$ there is nothing to prove. If $L_{\bar{k}}$ becomes $true_3$ in $\Phi_P \uparrow i$, then there is a general clause $L_{\bar{k}} \leftarrow M_1, \ldots, M_m$ in $ground(P)$ with $M_1 \wedge \ldots \wedge M_m$ being $true_3$ in $\Phi_P \uparrow (i-1)$. For $j \in [1,m]$, if the relation occurring in M_j is in Neg^*_P, then M_j is $true_3$ in $\Phi_P \uparrow (i-1)$ implies M_j is $true_2$ in I by Corollary 6.6 and Proposition 6.1(i). If the relation occurring in M_j is not in Neg^*_P, then M_j is $true_3$ in $\Phi_P \uparrow (i-1)$ implies M_j is $true_2$ in I by the induction hypothesis. Hence $M_1 \wedge \ldots \wedge M_m$ is $true_2$ in I, which implies $L_{\bar{k}}$ is $true_2$ in I, since I is a model of $L_{\bar{k}} \leftarrow M_1, \ldots, M_m$. This contradicts the fact that $L_{\bar{k}}$ is $false_2$ in I.

In both Subcase 1 and 2, we have that $L_1 \wedge \ldots \wedge L_k$ is not undefined in $\Phi_P \uparrow n$, as it is either $true_3$ or $false_3$ in Subcase 1, and $false_3$ in Subcase 2. As a consequence, A is either $true_3$ or $false_3$ in $\Phi_P \uparrow (n+1)$, which establishes the claim. $\square$

Corollary 6.8 *Let P be an acceptable general program. Then $\Phi_P \uparrow \omega$ is the unique fixpoint of Φ_P.*

Proof. We have $\Phi_P \uparrow \omega \subseteq \Phi_P \uparrow (\omega + 1)$, i.e. $\Phi_P \uparrow \omega \subseteq \Phi_P(\Phi_P \uparrow \omega)$. By Theorem 6.7 $\Phi_P \uparrow \omega$ is total, so in fact $\Phi_P \uparrow \omega = \Phi_P(\Phi_P \uparrow \omega)$, i.e. $\Phi_P \uparrow \omega$ is a fixpoint of Φ_P. Moreover, by the monotonicity of Φ_P, every fixpoint of Φ_P of the form $\Phi_P \uparrow \alpha$ is contained in any other fixpoint, so in fact $\Phi_P \uparrow \omega$ is the unique fixpoint of Φ_P. □

The following corollary summarizes the relevant properties of $N_P = \Phi_P \uparrow \omega$.

Corollary 6.9 *Let P be an acceptable general program. Then*

(i) N_P is total,

(ii) N_P is the unique fixpoint of Φ_P,

(iii) N_P is the unique 3-valued Herbrand model of $comp(P)$,

(iv) N_P^+ is the unique fixpoint of T_P,

(v) N_P^+ is the unique Herbrand model of $comp(P)$,

(vi) for all ground atoms A such that no LDNF-derivation of $P \cup \{\leftarrow A\}$ flounders,

$$A \in N_P^+ \text{ iff there exists an LDNF-refutation of } P \cup \{\leftarrow A\}.$$

In particular, this equivalence holds for all ground atoms A when P is non-floundering.

Proof.
(i) By Theorem 6.7.
(ii) By Corollary 6.8.
(iii) By (ii) and the result of Fitting [Fit85] stating that a 3-valued Herbrand interpretation is a model of $comp(P)$ iff it is a fixpoint of Φ_P.
(iv) By Theorem 6.7 and Corollaries 6.8 and 6.4.
(v) By Theorem 6.5.
(vi) Consider a ground atom A such that no LDNF-derivation of $P \cup \{\leftarrow A\}$ flounders. By the soundness of the SLDNF-resolution and (v) if there exists an LDNF-refutation of $P \cup \{\leftarrow A\}$ then $A \in N_P^+$. To prove the converse implication assume $A \in N_P^+$. By Corollary 4.11 all LDNF-derivations of $P \cup \{\leftarrow A\}$ are finite. Suppose by contradiction that none of them is successful. Then the LDNF-tree for $P \cup \{\leftarrow A\}$ is non-floundering and finitely failed. By the soundness of the SLDNF-resolution and (v), $N_P^+ \models \neg A$, i.e. $A \notin N_P^+$ which is a contradiction.

□

Clause (v) of the above Corollary shows that when $P^- = P$, $comp(P^-)$ has exactly one Herbrand model. This implies that for such general programs essentially full semantic information has to be used to reason about their termination. An example of such a program is the `GAME` program discussed in Section 5.

Clause (vi) can be seen as a completeness result for acceptable general programs that relates the LDNF-resolution to the model N_P^+.

By restricting our attention to programs we get the following additional conclusions.

Corollary 6.10 *Let P be an acceptable program. Then*

(i) $T_P \uparrow \omega$ is the unique fixpoint of T_P,

(ii) $T_P \uparrow \omega = T_P \downarrow \omega$.

Proof. By the result of Fitting [Fit85]

$$\Phi_P \uparrow \alpha = \ (T_P \uparrow \alpha,\ B_P - T_P \downarrow \alpha),$$

so

$$T_P \uparrow \omega = (\Phi_P \uparrow \omega)^+ = N_P^+.$$

Now (i) follows by Corollarry 6.9 (iv) and (ii) follows by Corollarry 6.9 (i). □

7 Conclusions

Assessment of the method

Our approach to termination is limited to the study of left terminating (general) programs, so it is useful to reflect on the relevance of this restriction.

First, observe that the notion of left termination is insensitive to the ordering of clauses in the programs. This seems to follow a good programming practice.

The main result of Bezem [Bez89] states that every total recursive function can be computed by a recurrent program. As recurrent programs are left terminating, the same property is shared by left terminating programs.

It is useful to notice a simple consequence of our approach to termination. By proving that a program P is acceptable and a goal G is bounded, we can conclude by Corollary 2.13 that the LD-tree for $P \cup \{G\}$ is finite. Thus, for the leftmost selection rule, the set of computed answer substitutions for $P \cup \{G\}$ is finite and consequently, by virtue of the strong completeness of SLD-resolution, we can use the LD-resolution to compute the set of all correct answer substitutions for $P \cup \{G\}$. In other words, query evaluation of bounded goals can be implemented using pure Prolog. The same remark applies to general non-floundering programs.

For a further analysis of left terminating programs we first introduce the following notions, essentially due to Mellish [Mel81]. Given an n-ary relation symbol p, by a *mode* for p we mean a function d_p from $\{1, \ldots, n\}$ to the set $\{+, -\}$. We write d_p in a more suggestive form $p(d_p(1), \ldots, d_p(n))$.

Modes indicate how the arguments of a relation should be used. If $d_p(i) =$ '$+$', we call i the *input position* of p and if $d_p(i) =$ '$-$', we call i the *output position* of p (both w.r.t. d_p) . The input positions should be replaced by ground terms and the output positions by variables. This motivates the following notion.

Given a mode d_p for a relation p, we say that an atom $A = p(t_1, \ldots, t_n)$ *respects* d_p if for $i \in [1, n]$, t_i is ground if i is an input position of p w.r.t. d_p and t_i is a variable if i is an output position of p w.r.t. d_p.

A *mode* for a program P is a function which assigns to each relation symbol of P a non-empty set of modes. Given a mode for a program P, we say that an atom A *respects moding* if A respects some mode in the set of modes associated with the relation p used in A.

As an example consider the mode for the program `append` represented by the following set:

$$\{\texttt{append}(+, +, -),\ \ \texttt{append}(-, -, +)\}.$$

It indicates that `append` should be called either with its first two arguments ground and the third being a variable, or with its first two arguments being a variable and the third argument ground. Then any atom $append(xs, ys, zs)$, where either xs, ys are ground and zs is a variable, or xs, ys are variables and zs is ground, respects moding.

The following simple theorem shows that the property of left termination is quite natural.

Theorem 7.1 *Let P be a program with a mode such that for all atoms A which respect moding, all LD-derivations of $P \cup \{\leftarrow A\}$ are finite. Then P is left terminating.*

Proof. Consider a ground atom A. A is a ground instance of some atom B which respects moding. By a variant of the Lifting Lemma applied to the LD-resolution we conclude that all LD-derivations of $P \cup \{\leftarrow A\}$ are finite. This implies that P is left terminating. □

The assumptions of the above theorem are satisfied by an overwhelming class of pure Prolog programs listed in the book of Sterling and Shapiro [SS86].

As Theorem 2.16 shows, the method presented in this paper is a complete method for proving termination of left terminating Prolog programs. We believe that it is also a useful method, since it allows us to factor termination proofs into simpler, separate proofs, which consist of checking the guesses for the level mapping $|\ |$ and the model I. Moreover, the method is modular, because termination proofs provided for subprograms can be reused in later proofs.

In this paper, the method is used as an "a posteriori" technique for verifying termination of existing Prolog programs. However, it could also provide a guideline for the program development, if the program is constructed together with its termination proof. A specific level mapping and a model could suggest, in particular, a specific ordering of atoms in clause bodies.

It is worth noting that some fragments of the proof of accceptability can be automated, at least in the case of the examples presented in Section 3, and in Apt and Pedreschi [AP90]. In our examples, where the function *listsize* is used, the task of checking the guesses for both the model I and the level mapping $|\ |$ 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 [Sho77] presented for this class a decision algorithm which is exponential. This is substantially lower than the complexity of the decision procedure for Presburger arithmetic. To illustrate this point, consider the following program `PERM` (for permutation):

```
(p1)  p([], [])←.
(p2)  p(Xs, [X | Ys]) ←
        a(X1s, [X|X2s], Xs),
        a(X1s, X2s, Zs),
        p(Zs, Ys).
```

augmented by the clauses (a_1) and (a_2) of Section 3 which define the relation a (for append).

The intention is to invoke p with its first argument instantiated. Clause (p_1) states that the empty list is a permutation of itself. Clause (p_2) takes care of a non-empty list xs - one should first split it into two sublists $x1s$ and $[x|x2s]$ and concatenate $x1s$ and $x2s$ to get zs. If now ys is a permutation of zs, $[x|ys]$ is a permutation of xs.

Consider now the following guess I for a model for the program `PERM`:

$$\begin{array}{rcl} I & = & \{p(zs, ys) \mid |zs| = |ys|\} \\ & \cup & \{a(x1s, x2s, zs) \mid |x1s| + |x2s| = |zs|\}. \end{array}$$

To show that I is a model of, say clause (p_2), we have to prove the following implication:

$$\{a(x1s, [x|x2s], xs), a(x1s, x2s, zs), p(zs, ys)\} \subseteq I \Rightarrow p(xs, [x|ys]) \in I.$$

By homomorphically mapping lists onto their lengths, i.e. by mapping [ ] to 0 and [|] to the successor function $s(.)$, we get the following formula of Presburger arithmetic:

$$x_1 + x_2 + 1 = x \ \wedge \ x_1 + x_2 = z \ \wedge \ z = y \Rightarrow x = y + 1$$

where $x_1 = |x1s|, x_2 = |x2s|, x = |xs|, z = |zs|, y = |ys|$.

The level mapping for `PERM` can be given by:

$$\begin{aligned} |p(zs, ys)| &= |zs| + 1, \\ |a(x1s, x2s, zs)| &= \min(|x1s|, |zs|). \end{aligned}$$

Then, for example, to establish that

$$|p(xs, [x|ys])| > |p(zs, ys)|$$

under the assumption that $I \models a(x1s, [x|x2s], xs) \wedge a(x1s, x2s, zs)$ it suffices to verify the following formula of Presburger arithmetic:

$$x_1 + x_2 + 1 = x \ \wedge \ x_1 + x_2 = z \Rightarrow x + 1 > z + 1.$$

This approach to partial automation of the termination proofs is described in detail in Pedreschi and Pieramico [PP92]. In particular, they implemented the above sketched procedure for checking left termination and verified mechanically that the quicksort program `QS` is left terminating.

Finally, let us mention that it is not immediately obvious how to extend the approach of this paper to "impure" Prolog programs. Some points like the use of cut to prune infinite branches or the use of negation as failure rule to resolve non-ground negative literals (so ignoring floundering) are in our opinion a bad programming practice and should be avoided instead of being formally analyzed.

Other issues, like the use of built-in's considerably complicate the matters and call for new insights. Termination of programs that use first order built-in's (so `var`, `nonvar`, `ground` etc.) is studied in Apt, Marchiori and Palamidessi [APM92] where for this purpose a new declarative semantics based on non-ground atoms is introduced.

Related work

Of course the subject of termination of Prolog programs has been studied by others. Without aiming at completeness we mention here the following related work.

Vasak and Potter [VP86] identified two forms of termination for logic programs – existential and universal one and characterized the class of universal terminating goals for a given program with selected selection rules. However, this characterization cannot be easily used to prove termination. Using our terminology, given a program P, a goal G is existentially terminating w.r.t. the leftmost selection rule if in the LD-tree for $P \cup \{G\}$ no infinite LD-derivation to the left of the leftmost successful derivation exists, and is universally terminating w.r.t. the leftmost selection rule if the LD-tree for $P \cup \{G\}$ is finite.

Baudinet [Bau88] presented a method for proving existential termination of (general) Prolog program in which with each program a system of equations is associated whose least fixpoint is the meaning of the program. By analyzing this least fixpoint various termination properties can be proved. The main method of reasoning is fixpoint or structural induction. In her proposal negation is treated indirectly by dealing with termination in presence of the *cut* operator using which negation can be simulated.

Recently, Bal Wang and Shyamasundar [BS91] provided a method of proving universal termination based on a concept of so-called U-graph in which the relevant connections through unification between the atoms of the goal and of the program are recorded. The method can also be used to establish termination of general Prolog programs. This method calls for the use of pre- and post-conditions that are associated with the nodes of the U-graph.

Bossi, Cocco and Fabris [BCF91] refined this method by exploiting level mappings applied to non-ground atoms. These level mappings are constructed from level mappings defined on non-ground terms. The key concept is that of *rigidity* that allows us to identify the terms whose level mapping is invariant under instantiation.

Ullman and Van Gelder [UvG88] considered the problem of automatic verification of termination of a Prolog program and a goal. In their approach first some sufficient set of inequalities between the sizes of the arguments of the relation symbols are generated, and then it is verified if they indeed hold. Termination of the programs studied in the Section 3 and 5 is beyond the scope of their method. This approach was improved in Plümer [Plü90b], [Plü90a], who allowed a more general form of the inequalities and the way sizes of the arguments are measured. This resulted in a more powerful method. The quicksort program studied in Section 3 can be handled using Plümer's method. However, the examples in Section 5, as well as the mergesort example considered in Apt and Pedreschi [AP91] remain beyond its scope. It is worth noting the complementary aim of our approach with respect to that of Ullman and Van Gelder [UvG88] and Plümer [Plü90b, Plü90a]. Their goal is the automatic verification of termination of a pure Prolog program and a goal. In their approach, some sufficient conditions for termination are identified, which can be statically checked. Obviously, such an approach cannot be complete due to the undecidability of the halting problem.

We propose instead a complete proof method, which characterizes precisely the left terminating (non-floundering, general) programs. Additionally, in the present paper and in Apt and Pedreschi [AP90] we provide simple proofs of termination for programs and goals which cannot be handled using the cited approach. On the other hand, we do not determine here any conditions under which our method could be automated. This should form part of a future research.

Deville [Dev90] also considers termination in his proposal of systematic program development. In his framework, termination proofs exploit well-founded orderings together with mode and multiplicity information, the latter representing an upper bound to the number of answer substitutions for goals which respect a given mode. For instance, a termination proof of the program `DC` of Example 2.5(iii) for the goal $\leftarrow dc(x, Y)$ would involve verification of the following statements (assuming that x is a ground term):

1. the goal $\leftarrow divide(x, X1, X2)$ respects moding, and both $X1$ and $X2$ are bound to ground terms, $x1$ and $x2$ respectively, by any computed answer substitution for such a goal;

2. both $x1$ and $x2$ are smaller than x w.r.t. some well-founded ordering;

3. the mode $divide(+, -, -)$ has a finite multiplicity.

Our approach seems to be simpler as it relies on fewer concepts. Also, it suggests a more uniform methodology. On the other hand, in Deville's approach more information about the program is obtained.

Acknowledgement

Marc Bezem made us aware of the importance of including subsidiary LDNF-trees in the definition of $nodes_P(G)$. One of the referees of a previous version of the paper made a number of helpful suggestions.

References

[AB90] K. R. Apt and M. Bezem. Acyclic programs. In D. H. D. Warren and P. Szeredi, editors, *Proceedings of the Seventh International Conference on Logic Programming*, pages 617–633. The MIT Press, 1990.

[ABW88] 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.

[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.

[AP90] K. R. Apt and D. Pedreschi. Studies in pure Prolog: termination. In J.W. Lloyd, editor, *Symposium on Computional Logic*, pages 150–176, Berlin, 1990. Springer-Verlag.

[AP91] K. R. Apt and D. Pedreschi. Proving termination of general Prolog programs. In T. Ito and A. Meyer, editors, *Proceeding of the International Conference on Theoretical Aspects of Computer Software*, Lecture Notes in Computer Science 526, pages 265–289, Berlin, 1991. Springer-Verlag.

[APM92] K.R. Apt, C. Palamidessi, and E. Marchiori. A theory of first-order built-in's of Prolog. In H. Kirchner and G. Levi, editors, *Proceeding of the Third International Conference on Algebraic and Logic Programming (ALP 92)*, Lecture Notes in Computer Science 632, pages 69–83, Berlin, 1992. Springer-Verlag.

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

[Bau88] M. Baudinet. Proving termination properties of PROLOG programs. In *Proceedings of the 3rd Annual Symposium on Logic in Computer Science (LICS)*, pages 336–347, Edinburgh, Scotland, 1988.

[BCF91] A. Bossi, N. Cocco, and M. Fabris. Termination of logic programs by exploiting term properties. In S. Abramsky and T.S.E. Maibaum, editors, *Proceedings CCPSD-TAPSOFT '91*, Lecture Notes in Computer Science 494, pages 153–180, Berlin, 1991. Springer-Verlag.

[Bez89] M. Bezem. Characterizing termination of logic programs with level mappings. In E. L. Lusk and R. A. Overbeek, editors, *Proceedings of the North American Conference on Logic Programming*, pages 69–80. The MIT Press, 1989.

[BS91] Bal Wang and R.K. Shyamasundar. Methodology for proving termination of logic programs. In M. Jantzen, editor, *Proceedings STACS '91*, Lecture Notes in Computer Science 480, pages 214–227, Berlin, 1991. Springer-Verlag.

[Cav89] L. Cavedon. Continuity, consistency, and completeness properties for logic programs. In G. Levi and M. Martelli, editors, *Proceedings of the Sixth International Conference on Logic Programming*, pages 571–584. The MIT Press, 1989.

[Der87] N. Dershowitz. Termination of rewriting. *Journal of Symbolic Computation*, 8:69–116, 1987.

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

[Fit85] M. Fitting. A Kripke-Kleene semantics for general logic programs. *Journal of Logic Programming*, 2:295–312, 1985.

[Flo67] R. W. Floyd. Assigning meanings to programs. In *Proceedings Symposium on Applied Mathematics, 19, Math. Aspects in Computer Science*, pages 19–32. American Society, 1967.

[HM87] S. Hanks and D. McDermott. Nonmonotonic logic and temporal projection. *Artificial Intelligence*, 33:379–412, 1987.

[Kle52] S. C. Kleene. *Introduction to Metamathematics.* van Nostrand, New York, 1952.

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

[Mel81] 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.

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

[Plü90a] L. Plümer. *Termination Proofs for Logic Programs.* Lecture Notes in Artificial Intelligence 446, Springer-Verlag, Berlin, 1990.

[Plü90b] L. Plümer. Termination proofs for logic programs based on predicate inequalities. In D. H. D. Warren and P. Szeredi, editors, *Proceedings of the Seventh International Conference on Logic Programming*, pages 634–648. The MIT Press, 1990.

[PP92] D. Pedreschi and C. Pieramico. Partial automation of termination proofs for Prolog programs. Technical report, Department of Computer Science, University of Pisa, Pisa, Italy, 1992. to appear.

[Sho77] R.E. Shostak. On the SUP-INF method for proving Presburger formulas. *J. ACM*, 24(4):529–543, 1977.

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

[UvG88] J. D. Ullman and A. van Gelder. Efficient tests for top-down termination of logical rules. *J. ACM*, 35(2):345–373, 1988.

[VP86] T. Vasak and J. Potter. Characterization of terminating logic programs. In *Proceedings of the 1986 IEEE Symposium on Logic Programming*, 1986.