Modular termination proofs for logic and pure Prolog programs

K.R. Apt, D. Pedreschi

Computer Science/Department of Software Technology

Report CS-R9316 March 1993

CWI is the National Research Institute for Mathematics and Computer Science. CWI is part of the Stichting Mathematisch Centrum (SMC), the Dutch foundation for promotion of mathematics and computer science and their applications. SMC is sponsored by the Netherlands Organization for Scientific Research (NWO). CWI is a member of ERCIM, the European Research Consortium for Informatics and Mathematics.

Copyright © Stichting Mathematisch Centrum
P.O. Box 4079, 1009 AB Amsterdam (NL)
Kruislaan 413, 1098 SJ Amsterdam (NL)
Telephone +31 20 592 9333
Telefax +31 20 592 4199

Modular Termination Proofs for Logic and Pure Prolog Programs

Krzysztof R. Apt
CWI
P.O. Box 4079, 1009 AB 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 uniform and simplified presentation of the methods of Bezem [Bez93] (first published as [Bez89]) and of Apt and Pedreschi [AP93] (first published as [AP90]) for proving termination of logic and Prolog programs. Then we show how these methods can be refined so that they can be used in a modular way.

1991 Mathematics Subject Classification: 68Q40, 68T15.
CR Categories: F.3.2., F.4.1, H.3.3, I.2.3.
Keywords and Phrases: termination, Prolog programs, modularity.

Notes. This research was partly supported by the ESPRIT Basic Research Action 6810 (Compulog 2). This paper will appear in: Proceedings of Fourth International School for Computer Science Researchers, Acireale, Oxford University Press, G. Levi (editor).

1 Introduction

1.1 Motivation

The theory of logic programming ensures us that SLD-resolution is a sound and complete procedure for executing logic programs. As a consequence, given a program P, every SLD-tree for a goal G is a complete search space for finding an SLD-refutation of G. In the actual implementations of logic programming, the critical choice is that of a tree-searching algorithm. Two basic tree-search strategies are: the *breadth-first* search which explores the tree by levels, and the *depth-first* search which explores the tree by branches. The former is a complete strategy, in the sense that it finds a success node if one exists, whereas the latter one is incomplete, since success nodes can be missed if an infinite branch is explored first.

However, for the efficiency reasons most implementations of logic programming do adopt the depth-first strategy; in case of Prolog additionally a fixed selection rule is adopted. This "destroys" the completeness results linking the declarative and operational semantics of logic programming and makes difficult to use the basic theory of logic programming for reasoning about programs.

These complications motivate research on methods for proving termination of logic programs, and in particular the approach of Bezem [Bez93], who proposed a method for proving termination with respect to all selection rules, and the approach of Apt and Pedreschi [AP93], who refined Bezem's method to the leftmost selection rule of Prolog. (For a discussion of related work the reader is referred to these two papers.)

Report CS-R9316
ISSN 0169-118X
CWI
P.O. Box 4079, 1009 AB Amsterdam, The Netherlands

The aim of the present paper is twofold. First, we provide a uniform and simplified presentation of these two methods, which shows that the method of Apt and Pedreschi for dealing with pure Prolog programs is a natural extension of Bezem's method for dealing with logic programs. Secondly, we provide an extension of both methods, which supports a compositional methodology for combining termination proofs of separate programs to obtain proofs of larger programs. A number of applications is presented to substantiate the effectiveness of these modular methods in breaking down termination proofs into smaller and simpler ones, and their ability to deal with *program schemes*. In particular, simple termination proofs are exhibited for a *divide and conquer* scheme, a *generate and test* scheme, and two schemes borrowed from functional programming: a *map* scheme and a *fold* scheme.

The paper is organized as follows. In Section 2 we present the method due to Bezem [Bez93] for proving termination of logic programs, and in Section 3 its modification due to Apt and Pedreschi [AP93] for proving termination of pure Prolog programs. Then in Sections 4 and 5 we refine these methods so that they can be used in a modular way.

1.2 Preliminaries

Throughout this paper we use the standard notation of Lloyd [Llo87] and Apt [Apt90]. In particular, for a logic program P (or simply a *program*) we denote the Herbrand Base of P by B_P and the least Herbrand model of P by M_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 [. | .] is a binary function which given a term x and a list xs produces a new list [x | 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 | [$\ldots$[x_n|[]]$\ldots$]].

Throughout the paper we consider SLD-resolution and LD-resolution. The latter is obtained from SLD-resolution by using Prolog first-left selection rule. The concepts of LD-derivation, LD-refutation, LD-tree, etc. are 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 $P \cup \{G\}$.

In what follows we shall use the multiset ordering. A *multiset*, sometimes called *bag*, is an unordered sequence. We denote a multiset consisting of elements $a_1, \ldots, a_n$ by $bag\,(a_1, \ldots, a_n)$. 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

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

where X, Y, Z are finite multisets of elements of W, and then define the multiset ordering $\prec_m$ 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. In particular, the multiset ordering over the set of natural numbers with their usual ordering is well-founded.

2 Termination

2.1 Motivation

Consider the following simple program `LIST`:

`list(Xs)` $\leftarrow$ `Xs` is a list.

```
list([H | Ts]) ← list(Ts).
list([]).
```

It is easy to see that

- for a list `t` the goal ← `list(t)` successfully terminates,
- for a *ground* term `t` which is not a list, the goal ← `list(t)` finitely fails.

Note that in the second statement we required that `t` is ground. Can we drop this restriction? The answer is "No". Indeed, consider the goal ← `list(X)` with a variable `X`.

As `X` unifies with `[H | Ts]`, we see that using the first clause ← `list(Ts)` is a resolvent of ← `list(X)`. By repeating this procedure we obtain an infinite LD-derivation which starts with ← `list(X)`. There is an easy fix to this problem — it suffices to reorder the clauses of the program. Then the goal ← `list(X)` terminates with the c.a.s. {`X`/[ ]}. So termination depends on the clause ordering.

Another, more interesting, possibility is to make the notion of termination independent of the clause ordering. According to this definition, a goal terminates if *all* derivations starting with it are finite. Then the goal ← `list(X)` does not terminate in this sense. Once the clause ordering becomes irrelevant, it is possible to adopt the view of logic programming theory and consider the program as the set (and not sequence) of clauses.

It is useful to note a simple consequence of this notion of termination. When a goal terminates in this strong sense, the corresponding computation tree is finite. Thus the depth first search in this tree terminates, and consequently it is possible to compute by means of Prolog *all* c.a.s.'s of the goal under consideration.

In what follows we shall study this stronger notion of termination. Our aim will be to identify for a given pure Prolog program those goals which terminate in the above sense. Clearly, our discussion concerning the program `LIST` and the goal ← `list(X)` is equally applicable to other programs.

2.2 Terminating programs

We begin our study of termination by analyzing termination in a very strong sense, namely w.r.t. all selection rules. This notion of termination is more applicable to logic programs than to Prolog programs. However, it is easier to handle and it will provide us with a useful basis from which a transition to the case of pure Prolog programs will be quite natural.

In this section we study the terminating programs in the following sense.

Definition 2.1 A program is called *terminating* if all its SLD-derivations starting with 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. 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. The method of proving termination considered here will allow us to identify for each program such a class of non-ground goals. As we shall see below, many Prolog programs, including `SUM`, `LIST` and `APPEND` are terminating.

To prove that a program is terminating the following concepts due to Bezem [Bez93] and Cavedon [Cav89] will play a crucial role.

Definition 2.2

- A *level mapping* for a program P is a function $|\ |: B_P \to N$ of ground atoms to natural numbers. For $A \in B_P$, $|A|$ is the *level of* A.
- A clause of P is called *recurrent with respect to a level mapping* $|\ |$, if for every ground instance $A \leftarrow \mathbf{A}, B, \mathbf{B}$ of it

$$|A| > |B|.$$

- A program P is called *recurrent with respect to a level mapping* $|\ |$, if all its clauses are. P is called *recurrent* if it is recurrent with respect to some level mapping. □

First, following Bezem [Bez93], let us "lift" the concept of level mapping to non-ground atoms.

Definition 2.3

- An atom A is called *bounded with respect to a level mapping* $|\ |$, if $|\ |$ is bounded on the set $[A]$ of ground instances of A. For A bounded w.r.t. $|\ |$, we define $|A|$, the *level of* A w.r.t. $|\ |$, as the maximum $|\ |$ takes on $[A]$.
- A goal is called *bounded with respect to a level mapping* $|\ |$, if all its atoms are. For $G = \leftarrow A_1, \ldots, A_n$ bounded w.r.t. $|\ |$, we define $|G|$, the *level of* G w.r.t. $|\ |$, as the multiset $bag(|A_1|, \ldots, |A_n|)$. If $|A_i| \leq k$ for $i \in [1, n]$, we say that G is *bounded by* k. □

The concept of boundedness is crucial when considering termination, as the following lemma shows. Recall that $\prec_m$ stands for the multiset ordering defined in the preliminaries.

Lemma 2.4 *Let P be a program that is recurrent w.r.t. a level mapping $|\ |$. Let G_1 be a goal that is bounded w.r.t. $|\ |$ and let G_2 be an SLD-resolvent of G_1 from P. Then*

- *G_2 is bounded w.r.t. $|\ |$,*
- *$|G_2| \prec_m |G_1|$.*

Proof. An SLD-resolvent of a goal and a clause is obtained by means of the following three operations:

- instantiation of the goal,
- instantiation of the clause,
- replacement of an atom, say H, of a goal by the body of a clause whose head is H.

Thus the lemma is an immediate consequence of the fact that an instance of a recurrent clause w.r.t. $|\ |$ is recurrent w.r.t. $|\ |$, and the following claims in which we refer to the given level mapping.

Claim 1 *An instance G' of a bounded goal G is bounded and $|G'| \preceq_m |G|$.*

Proof. It suffices to note that an instance A' of a bounded atom A is bounded and $|A'| \leq |A|$. □

Claim 2 *For every recurrent clause $H \leftarrow \mathbf{B}$, if $\leftarrow H$ is bounded, then $\leftarrow \mathbf{B}$ is bounded and $|\leftarrow \mathbf{B}| \prec_m |\leftarrow H|$.*

Proof. Consider an atom C occurring in a ground instance of $\leftarrow \mathbf{B}$. Then it occurs in the body of a ground instance of $H \leftarrow \mathbf{B}$, say $H\theta \leftarrow \mathbf{B}\theta$. By the recurrence of $H \leftarrow \mathbf{B}$ we get $|C| < |H\theta|$, so $|C| < |H|$. This proves the claim. □

Claim 3 *For every recurrent clause $H \leftarrow \mathbf{B}$ and sequences of atoms $\mathbf{A}$ and $\mathbf{C}$, if $\leftarrow \mathbf{A}, H, \mathbf{C}$ is bounded, then $\leftarrow \mathbf{A}, \mathbf{B}, \mathbf{C}$ is bounded and $|\leftarrow \mathbf{A}, \mathbf{B}, \mathbf{C}| \prec_m |\leftarrow \mathbf{A}, H, \mathbf{C}|$.*

Proof. Immediate by Claim 2 and the definition of the multiset ordering. □
□

The following conclusions are now immediate.

Corollary 2.5 *Let P be an recurrent program and G a bounded goal. Then all SLD-derivations of $P \cup \{G\}$ are finite.*

Proof. The multiset ordering is well-founded. □

Corollary 2.6 *Every recurrent program is terminating.*

Proof. Every ground goal is bounded. □

These corollaries can be easily applied to various Prolog programs. The level mapping can be usually defined as a simple function of the terms of the ground atom. The following natural concept, due to Ullman and Van Gelder [UvG88], will often be useful.

Define by induction a function $|\ |$, called *listsize*, which assigns natural numbers to ground terms:

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

Note that for a list xs, $|xs|$ equals its length.

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

Definition 2.7 Let $|\ |$ be a level mapping. An atom A is called *rigid* w.r.t. $|\ |$ if $|\ |$ is constant on the set $[A]$ of ground instances of A. □

Obviously, rigid atoms are bounded.

Example 2.8
(i) Consider the program `LIST`. Define

`|list(t)| = |t|.`

It is straightforward to see that `LIST` is recurrent w.r.t. $|\ |$ and that for a list `t`, the atom `list(t)` is rigid w.r.t. $|\ |$. By Corollary 2.6 we conclude that `LIST` is terminating and by Corollary 2.5 we conclude that for a list `t`, all SLD-derivations of `LIST` $\cup$ { $\leftarrow$ `list(t)`} are finite.

(ii) Consider now the program `MEMBER`:

```
member(Element, List) ← Element is an element of the list List.

member(X, [Y | Xs]) ← member(X, Xs).
member(X, [X | Xs]).
```

Using the level mapping

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

we conclude by Corollary 2.6 that `MEMBER` is terminating and by Corollary 2.5 that for a list `t`, all SLD-derivations of `MEMBER` $\cup$ { $\leftarrow$ `member(s, t)`} are finite.

□

We now prove the converse of Corollary 2.6. With a goal G we associate the set of SLD-derivations of $P \cup \{G\}$. These SLD-derivations can be structured as a tree which we call an *S-tree* for $P \cup \{G\}$. In this tree the resolvents of a goal w.r.t. all selection rules and all input clauses constitute its direct descendants.

Lemma 2.9 *An S-tree for $P \cup \{G\}$ is finite iff all SLD-derivations of $P \cup \{G\}$ are finite.*

Proof. By the fact that we fixed the choice of mgu's and the fact that logic programs are finite, the S-trees are finitely branching. The claim now follows by König's Lemma König [Kön27]. □

This lemma allows us to concentrate on S-trees. For a program P and a goal G, we denote by $nodes_P(G)$ the number of nodes in the S-tree for $P \cup \{G\}$.

Lemma 2.10 *Let P be a program and G a goal such that the S-tree for $P \cup \{G\}$ is finite. Then*

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

(ii) for all atoms A of G, $nodes_P(\leftarrow A) \leq nodes_P(G)$,

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

Proof.
(i) By the Lifting Lemma (see Lloyd [Llo87]) we conclude that to every SLD-derivation of $P \cup \{G\theta\}$ with input clauses $C_1, C_2, \ldots$, there corresponds an SLD-derivation of $P \cup \{G\}$ with variants of input clauses $C_1, C_2, \ldots$, of the same or larger length. This implies the claim.
(ii), (iii) Immediate by the definition. □

We can now prove the desired result.

Theorem 2.11 *Let P be a terminating program. Then for some level mapping $|\ |$*

(i) P is recurrent w.r.t. $|\ |$,

(ii) for every goal G, G is bounded w.r.t. $|\ |$ iff all SLD-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 terminating, by Lemma 2.9 this level mapping is well defined. First we prove one implication of (ii).

(ii1) Consider a goal G such that all SLD-derivations of $P \cup \{G\}$ are finite. We prove that G is bounded by $nodes_P(G)$ w.r.t. $|\ |$.
To this end take a ground instance $\leftarrow A_1, \ldots, A_n$ of G and $i \in [1, n]$. We have

$$\begin{array}{ll} & nodes_P(G) \\ \geq & \quad \{\text{Lemma 2.10 (i)}\} \\ & nodes_P(\leftarrow A_1, \ldots, A_n) \\ \geq & \quad \{\text{Lemma 2.10 (ii)}\} \\ & nodes_P(\leftarrow A_i) \\ = & \quad \{\text{definition of } |\ |\} \\ & |A_i|, \end{array}$$

which proves the claim.

(i) We now prove that P is recurrent w.r.t. $|\ |$. 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

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

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 SLD-resolution, $\leftarrow B_1\mu, \ldots, B_n\mu$ is an SLD-resolvent of $\leftarrow A\theta$.
Then for $i \in [1, n]$

$$\begin{array}{ll} & |A\theta| \\ = & \quad \{\text{definition of } |\ |\} \\ & nodes_P(\leftarrow A\theta) \\ > & \quad \{\text{Lemma 2.10 (iii), } \leftarrow B_1\mu, \ldots, B_n\mu \text{ is a resolvent of } \leftarrow A\theta\} \\ & nodes_P(\leftarrow B_1\mu, \ldots, B_n\mu) \\ \geq & \quad \{\text{part (ii1), with } G := \leftarrow B_1\mu, \ldots, B_n\mu \text{ and } A_i := B_i\theta\} \\ & |B_i\theta|. \end{array}$$

(ii2) Consider a goal G which is bounded w.r.t. $|\ |$. Then by (i) and Corollary 2.5 all SLD-derivations of $P \cup \{G\}$ are finite. □

Corollary 2.12 *A program is terminating iff it is recurrent.*

Proof. By Corollary 2.6 and Theorem 2.11. □

2.3 Examples

Subset

Consider the following program SUBSET:

`subset(Xs, Ys)` ← each element of the list `Xs` is a member of the list `Ys`.

```
subset([X | Xs], Ys) ← member(X, Ys), subset(Xs, Ys).
subset([], Ys).
```

augmented by the MEMBER program.

To prove that SUBSET is recurrent we use the following level mapping:

$$\begin{aligned} |\texttt{member(x,xs)}| &= |\texttt{xs}|, \\ |\texttt{subset(xs,ys)}| &= |\texttt{xs}| + |\texttt{ys}|. \end{aligned}$$

By Corollary 2.6 SUBSET is terminating and consequently by Corollary 2.5 if `xs` and `ys` are lists, all SLD-derivations of SUBSET $\cup$ { ← `subset(xs,ys)`} are finite.

In general, various choices for the level mapping exist and for each choice different conclusions can be drawn. The following three simple examples illustrate this point.

Append

Consider the program APPEND:

`app(Xs, Ys, Zs)` ← `Zs` is the result of concatenating the lists `Xs` and `Ys`.

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

It is easy to check that APPEND is recurrent w.r.t. the level mapping `|app(xs, ys, zs)| = |xs|` and also with respect to the level mapping `|app(xs, ys, zs)| = |zs|`. In each case we get different class of goals which are bounded. The level mapping

```
|app(xs, ys, zs)| = min(|xs|,|zs|)
```

combines the advantages of both of them. APPEND is easily seen to be recurrent w.r.t. this level mapping and if `xs` is a list or `zs` is a list, `app(xs, ys, zs)` is bounded (though not rigid). By Corollary 2.6 APPEND is terminating and by Corollary 2.5 if `xs` is a list or `zs` is a list, all SLD-derivations of APPEND $\cup$ { ← `app(xs, ys, zs)`} are finite.

Select

Consider the program SELECT:

`select(X, Xs, Zs)` ← `Zs` is the result of deleting one occurrence of `X` from the list `Xs`.

```
select(X, [X | Xs], Xs).
select(X, [Y | Xs], [Y | Zs]) ← select(X, Xs, Zs).
```

As in the case of the APPEND program, it is most advantageous to use the level mapping

```
|select(xs, ys, zs)| = min(|ys|,|zs|).
```

Then SELECT is recurrent w.r.t | | and if `ys` is a list or `zs` is a list, all SLD-derivations of SELECT $\cup$ { ← `select(xs, ys, zs)`} are finite.

Sum

Finally, consider the following program SUM:

```
sum(X, Y, Z) ← X, Y, Z are natural numbers such that Z is the sum of X and Y.

sum(X, s(Y), s(Z)) ← sum(X, Y, Z).
sum(X, 0, X).
```

Again, it is most advantageous to use here the level mapping

$$|\texttt{sum(x, y, z)}| = \min(\texttt{size(y)}, \texttt{size(z)}),$$

where for a term t, size(t) denotes the number of symbols in t.

Then SUM is recurrent w.r.t. $|\ |$ and for a ground y or z, sum(x, y, z) is bounded w.r.t. $|\ |$. By Corollary 2.6 SUM is terminating and by Corollary 2.5 for a ground y or z, all SLD-derivations of SUM $\cup$ $\{\leftarrow$ sum(x, y, z)$\}$ are finite.

3 Left Termination

3.1 Motivation

Because of Corollary 2.12, 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 3.1
(i) First we consider a terminating program P such that for some goal G all LD-derivations of $P \cup \{G\}$ are finite, whereas some SLD-derivation of $P \cup \{G\}$ is infinite.

Examine the following program EVEN:

```
even(X) ← X is an even natural number.

even(s(s(X))) ← even(X).
even(0).

lte(X, Y) ← X, Y are natural numbers such that X is smaller or equal than Y.

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

EVEN is recurrent with |even(x)| = size(x) and |lte(x,y)|= min(size(x), size(y)), so by Corollary 2.6 it is terminating. Now consider the goal:

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

which is supposed to compute the even numbers not exceeding 100. One can show that all LD-derivations of G are finite, whereas there exists an infinite SLD-derivation when the rightmost selection rule is used. As a consequence of Corollary 2.5 the goal G is not bounded, although it can be evaluated by a finite Prolog computation.

This example is a contrived instance of the *generate-and-test* programming technique. This technique involves two procedures, one which generates the set of candidates, and another which tests whether these candidates are solutions to the problem. Actually, most Prolog programs that are implementations of the "generate-and-test" technique are not recurrent, as they heavily depend on the left-to-right order of evaluation, like the above goal.

(ii) Next, we consider a program P which is not terminating but such that all LD-derivations starting with a ground goal are finite. The following NAIVE REVERSE program is often used as a benchmark for Prolog applications:

```
reverse(Xs, Ys) ← Ys is a reverse of the list Xs.

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

augmented by the APPEND program.

It is easy to check that the ground goal ← `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. Thus `reverse` is not terminating. However, one can show that all LD-derivations starting with a goal ← `reverse(s, y)` for `s` ground (or `s` list) are finite.

(iii) More generally, consider the following program DC, representing a (binary) *divide and conquer* schema; it is parametric with respect to the relations `base, conquer, divide` and `merge`.

```
dcsolve(X, Y) ←
   base(X),
   conquer(X, Y).
dcsolve(X, Y) ←
   divide(X, X0, X1, X2),
   dcsolve(X1, Y1),
   dcsolve(X2, Y2),
   merge(X0, Y1, Y2, Y).
```

Many programs naturally fit into this schema, or its generalization to non fixed arity of the relations `divide/merge`. 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 `dcsolve(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. □

To cope with these difficulties we first modify the definition of a terminating program.

3.2 Left terminating programs

Definition 3.2 A program is called *left terminating* if all its LD-derivations starting with a ground goal are finite. □

This notion of termination is clearly more appropriate for the study of Prolog programs than that of a terminating program. To prove that a program is left terminating, and to characterize the goals that terminate w.r.t. such a program, we introduce the following concepts due to Apt and Pedreschi [AP93].

Definition 3.3 Let P be a program, $||$ a level mapping for P and I a (not necessarily Herbrand) interpretation of P.

- A clause of P is called *acceptable with respect to* $||$ *and* I, if I is its model and for every ground instance $A \leftarrow \mathbf{A}, B, \mathbf{B}$ of it such that $I \models \mathbf{A}$

$$|A| > |B|.$$

  In other words, for every ground instance $A \leftarrow B_1, \ldots, B_n$ of the clause

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

  where

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

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

The use of the premise $I \models \mathbf{A}$ 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 $\leftarrow \mathbf{A}, B, \mathbf{B}$ using the leftmost selection rule the atom B is reached, the atoms $\mathbf{A}$ 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 an interpretation I of P, in the definition of acceptability w.r.t. $||$ and I, for every ground instance $A \leftarrow B_1, \ldots, B_n$ if a clause in 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 shows that the notion of acceptability generalizes that of recurrence.

Lemma 3.4 *A program is recurrent w.r.t.* $||$ *iff it is acceptable w.r.t.* $||$ *and* B_P. □

Our aim is to prove that the notions of acceptability and left termination coincide. To this end we need the notion of boundedness. The concept of a bounded goal used here differs from that introduced in Definition 2.3 in that it takes into account the interpretation I. This results in a more complicated definition.

In what follows, assume that the maximum function $max : 2^\omega \rightarrow N \cup \{\infty\}$ is defined as:

$$max\ S \;=\; \begin{cases} 0 & \text{if } S = \emptyset, \\ n & \text{if } S \text{ is finite and non-empty, and } n \text{ is the maximum of } S, \\ \infty & \text{if } S \text{ is infinite.} \end{cases}$$

Then $max\ S < \infty$ iff the set S is finite.

Definition 3.5 Let P be a program, $||$ a level mapping for P and I an interpretation of P.

- With each goal $G = \leftarrow A_1, \ldots, A_n$ we associate n sets of natural numbers defined as follows, for $i \in [1, n]$:

$$|G|_i^I = \{|A'_i| \mid \leftarrow A'_1, \ldots, A'_n \text{ is a ground instance of } G \text{ and } I \models A'_1 \wedge \ldots \wedge A'_{i-1}\}.$$

- A goal G is called *bounded* w.r.t. $||$ and I if $|G|_i^I$ is finite, for $i \in [1, n]$.
- For $G = \leftarrow A_1, \ldots, A_n$ bounded w.r.t. $||$ and I we define a multiset $|G|_I$ of natural numbers as follows:

$$|G|_I \;=\; bag\,(max\,|G|_1^I, \ldots, max\,|G|_n^I).$$

- For G bounded w.r.t. $||$ and I, and $k \geq 0$, we say that G is *bounded by* k (w.r.t. $||$ and I) if $k \geq h$ for $h \in |G|_I$. □

Note that a goal G is bounded w.r.t. $||$ and B_P iff it is bounded w.r.t. $||$ in the sense of Definition 2.3.

Lemma 3.6 *Let P be a program that is acceptable w.r.t. a level mapping $||$ and an interpretation I. Let G_1 be a goal that is bounded w.r.t. $||$ and I, and let G_2 be an LD-resolvent of G_1 from P. Then*

(i) G_2 *is bounded w.r.t.* $||$ *and* I,

(ii) $|G_2|_I \prec_m |G_1|_I$.

Proof. It suffices to prove the following claims in which we refer to the given level mapping and interpretation I.

Claim 1 *An instance* G' *of a bounded goal* $G = \leftarrow A_1, \ldots, A_n$ *is bounded and* $|G'|_I \preceq_m |G|_I$.

Proof. It suffices to note that $|G'|_i^I \subseteq |G|_i^I$ for $i \in [1, n]$. □

Claim 2 *For every acceptable clause* $A \leftarrow \mathbf{B}$ *and sequence of atoms* $\mathbf{C}$, *if* $\leftarrow A, \mathbf{C}$ *is bounded, then* $\leftarrow \mathbf{B}, \mathbf{C}$ *is bounded and* $|\leftarrow \mathbf{B}, \mathbf{C}|_I \prec_m |\leftarrow A, \mathbf{C}|_I$.

Proof. Let $\mathbf{B} = B_1, \ldots, B_n$ and $\mathbf{C} = C_1, \ldots, C_m$, for $n, m \geq 0$. We first prove the following facts.

Fact 1 *For* $i \in [1, n]$, $|\leftarrow B_1, \ldots, B_n, C_1, \ldots, C_m|_i^I$ *is finite, and*

$$max|\leftarrow B_1, \ldots, B_n, C_1, \ldots, C_m|_i^I < max|\leftarrow A, C_1, \ldots, C_m|_1^I.$$

Proof. We have

$$\begin{aligned}
& max|\leftarrow B_1, \ldots, B_n, C_1, \ldots, C_m|_i^I \\
= \quad & \{\text{Definition 3.5 }\} \\
& max\{|B_i'| \mid \leftarrow B_1', \ldots, B_n' \text{ is a ground instance of } \leftarrow \mathbf{B} \text{ and } I \models B_1' \wedge \ldots \wedge B_{i-1}'\} \\
= \quad & \{\text{for some } A',\ A' \leftarrow B_1', \ldots, B_n' \text{ is a ground instance of } A \leftarrow \mathbf{B}\} \\
& max\{|B_i'| \mid A' \leftarrow B_1', \ldots, B_n' \text{ is a ground instance of } A \leftarrow \mathbf{B} \text{ and } I \models B_1' \wedge \ldots \wedge B_{i-1}'\} \\
< \quad & \{\text{Definition 3.3 and the fact that } \forall x \in S\ \exists y \in R : x < y \text{ implies } max\ S < max\ R\} \\
& max\{|A'| \mid A' \text{ is a ground instance of } A\} \\
= \quad & \{\text{Definition 3.5}\} \\
& max|\leftarrow A, C_1, \ldots, C_m|_1^I.
\end{aligned}$$

□

Fact 2 *For* $j \in [1, m]$, $|\leftarrow B_1, \ldots, B_n, C_1, \ldots, C_m|_{j+n}^I$ *is finite, and*

$$max|\leftarrow B_1, \ldots, B_n, C_1, \ldots, C_m|_{j+n}^I \leq max|\leftarrow A, C_1, \ldots, C_m|_{j+1}^I.$$

Proof. We have

$$\begin{aligned}
& max|\leftarrow B_1, \ldots, B_n, C_1, \ldots, C_m|_{j+n}^I \\
= \quad & \{\text{Definition 3.5}\} \\
& max\{|C_j'| \mid \leftarrow B_1', \ldots, B_n', C_1', \ldots, C_m' \text{ is a ground instance of } \leftarrow \mathbf{B}, \mathbf{C} \\
& \qquad \text{and } I \models B_1' \wedge \ldots \wedge B_n' \wedge C_1' \wedge \ldots \wedge C_{j-1}'\} \\
\leq \quad & \{\text{for some } A',\ A' \leftarrow B_1', \ldots, B_n' \text{ is a ground instance of } A \leftarrow \mathbf{B},\ I \text{ is a model of } P, \\
& \qquad \text{and } S \subseteq R \text{ implies } max\ S \leq max\ R\} \\
& max\{|C_j'| \mid \leftarrow A', C_1', \ldots, C_m' \text{ is a ground instance of } \leftarrow A, \mathbf{C} \\
& \qquad \text{and } I \models A' \wedge C_1' \wedge \ldots \wedge C_{j-1}'\} \\
= \quad & \{\text{Definition 3.5}\} \\
& max|\leftarrow A, C_1, \ldots, C_m|_{j+1}^I.
\end{aligned}$$

□

As a consequence of Facts 1 and 2 $\leftarrow \mathbf{B}, \mathbf{C}$ is bounded and

$$bag(max|\leftarrow \mathbf{B}, \mathbf{C}|_1^I, \ldots, max|\leftarrow \mathbf{B}, \mathbf{C}|_{n+m}^I) \prec_m bag(max|\leftarrow A, \mathbf{C}|_1^I, \ldots, max|\leftarrow A, \mathbf{C}|_{m+1}^I)$$

which establishes the claim. □

□

Corollary 3.7 *Let P be an acceptable program and G a bounded goal. Then all LD-derivations of $P \cup \{G\}$ are finite.*

Proof. The multiset ordering is well-founded. □

Corollary 3.8 *Every acceptable program is left terminating.*

Proof. Every ground goal is bounded. □

We now prove the converse of Corollary 3.8. To this end we proceed analogously as in the case of terminating programs and analyze the size of finite LD-trees. We need the following analogue of Lemma 2.10, where for a program P and a goal G we now denote by $nodes_P(G)$ the number of nodes in the LD-tree for $P \cup \{G\}$.

Lemma 3.9 *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, $nodes_P(G\theta) \leq nodes_P(G)$,

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

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

Proof.
(i) By the Lifting Lemma (see Lloyd [Llo87]) 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 variants of 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 $G = \leftarrow A_1, \ldots, A_n$ $(n \geq k)$. By an appropriate renaming of variables (formally justified by the Variant Lemma (see 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 $\leftarrow \mathbf{B}$ by $\leftarrow \mathbf{B}, A_{k+1}\theta, \ldots, A_n\theta$, where θ is the composition of the mgu's used on the path from the root H to the node $\leftarrow \mathbf{B}$. This implies the claim.

(iii) Immediate by the definition. □

We can now demonstrate the desired result.

Theorem 3.10 *Let P be a left terminating program. Then for some level mapping $|\ |$ and an interpretation I of P*

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

(ii) for every 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).

(iil) 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

$$\begin{array}{ll} & nodes_P(G) \\ \geq & \quad \{\text{Lemma 3.9 (i)}\} \\ & nodes_P(\leftarrow A_1, \ldots, A_n) \\ \geq & \quad \{\text{Lemma 3.9 (ii)}\} \\ & nodes_P(\leftarrow A_1, \ldots, A_{\bar{n}}) \\ \geq & \quad \{\text{Lemma 3.9 (iii), noting that for } j \in [1, \bar{n}-1] \\ & \quad \text{there is an LD-refutation of } P \cup \{\leftarrow A_1, \ldots, A_j\}\} \\ & nodes_P(\leftarrow A_i, \ldots, A_{\bar{n}}) \\ \geq & \quad \{\text{Lemma 3.9 (ii)}\} \\ & nodes_P(\leftarrow A_i) \\ = & \quad \{\text{definition of } |\ |\} \\ & |A_i| \\ = & \ell. \end{array}$$

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

$$\begin{array}{ll} & |A\theta| \\ = & \quad \{\text{definition of } |\ |\} \\ & nodes_P(\leftarrow A\theta) \\ > & \quad \{\text{Lemma 3.9 (iii), } \leftarrow B_1\mu, \ldots, B_n\mu \text{ is a resolvent of } \leftarrow A\theta\} \\ & nodes_P(\leftarrow B_1\mu, \ldots, B_n\mu) \\ \geq & \quad \{\text{part (ii1), with } G := \leftarrow B_1\mu, \ldots, B_n\mu \text{ and } A_i := B_i\theta\} \\ & |B_i\theta|. \end{array}$$

(ii2) Consider a goal G which is bounded w.r.t. $|\ |$ and I. Then by (i) and Corollary 3.7 all LD-derivations of $P \cup \{G\}$ are finite. □

Corollary 3.11 *A program is left terminating iff it is acceptable.*

Proof. By Corollary 3.8 and Theorem 3.10. □

3.3 Examples

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

3.10 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. We illustrate it by means of two examples. We use in them the defined before function listsize $|\ |$, which assigns natural numbers to ground terms.

In the following, we present the proof of acceptability (w.r.t. a level mapping $|\ |$ and an interpretation I) of a given clause $C = A_0 \leftarrow A_1, \ldots, A_n$ by means of the following *proof outline*:

$$
\begin{array}{lll}
 & \{f_0\} & \\
A_0 & \leftarrow & \{t_0\} \\
 & A_1, & \{t_1\} \\
 & \quad \{f_1\} & \\
 & \vdots & \\
 & A_{n-1}, & \{t_{n-1}\} \\
 & \quad \{f_{n-1}\} & \\
 & A_n. & \{t_n\} \\
 & \quad \{f_n\} &
\end{array}
$$

Here, t_i and f_i, for $i \in [0, n]$ are integer expressions and first-order formulas, respectively, such that all ground instances of the following properties are satisfied:

1. $t_i = |A_i|$, for $i \in [0, n]$,
2. $f_i \equiv I \models A_i$, for $i \in [0, n]$,
3. $f_1 \wedge \ldots \wedge f_n \Rightarrow f_0$,
4. For $i \in [1, n]$: $f_1 \wedge \ldots \wedge f_{i-1} \Rightarrow t_0 > t_i$.

We omit $\{f_i\}$ (resp. $\{t_i\}$) in the proof outlines if $f_i =$ **true** (resp. $t_i = 0$.) It is immediate that a proof outline satisfying properties 1 – 4 corresponds to the proofs that I is a model of the clause C, and that C is acceptable w.r.t. $|\ |$ and I. We found it convenient to use proof outlines to present the proofs of acceptability, as most steps in these proofs are trivial and can be omitted without loss of information.

Permutation

Consider the following program PERMUTATION:

```
perm(Xs, Ys) ← Ys is a permutation of the list Xs.

perm(Xs, [X | Ys]) ←
   app(X1s, [X | X2s], Xs),
   app(X1s, X2s, Zs),
   perm(Zs, Ys).
perm([], []).
```

augmented by the APPEND program.

The intention is to invoke **perm** with its first argument instantiated. The first clause 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**, then **[x | ys]** is a permutation of **xs**. The second clause states that the empty list is a permutation of itself.

Observe the following:

- PERMUTATION is not recurrent. Indeed, consider the SLD-derivation of PERMUTATION $\cup$ { ← perm(xs, [x | ys])} with **xs**, **x**, **ys** ground, in whose second goal the middle atom **app(x1s, x2s, zs)** is selected. By repeatedly applying the recursive clause of APPEND we obtain an infinite derivation. Thus PERMUTATION is not terminating and so by Corollary 2.6 it is not recurrent.

- The Herbrand interpretation

$$I_{APP} = \{\mathtt{app(xs,ys,zs)} \mid |\mathtt{xs}| + |\mathtt{ys}| = |\mathtt{zs}|\}$$

is a model of the program APPEND. Indeed, I_{APP} is trivially a model of the non-recursive clause of the app relation and the following proof outline shows that I_{APP} is a model of the recursive clause:

$$\begin{array}{ll} \{1 + |\mathtt{xs}| + |\mathtt{ys}| = 1 + |\mathtt{zs}|\} & \\ \mathtt{app([x|xs],ys,[x|zs])} & \leftarrow \\ & \mathtt{app(xs,ys,zs)}. \\ & \quad \{|\mathtt{xs}| + |\mathtt{ys}| = |\mathtt{zs}|\} \end{array}$$

- The program PERMUTATION is acceptable w.r.t. the level mapping $|\ |$ and the interpretation I_{PERM} defined by:

$$\begin{aligned} |\mathtt{perm(xs,ys)}| &= |\mathtt{xs}| + 1, \\ |\mathtt{app(xs,ys,zs)}| &= \min(|\mathtt{xs}|, |\mathtt{zs}|), \end{aligned}$$

$$I_{PERM} = [\mathtt{perm(Xs,Ys)}] \cup I_{APP}.$$

Recall that $[A]$ for an atom A stands for the set of all ground instances of A. We already noted in Example 2.8 that APPEND is recurrent w.r.t. $|\ |$. The proof outline for the non-recursive clause of the perm relation is obvious. For the recursive clause take the following proof outline:

$$\begin{array}{lll} \mathtt{perm(xs,[x|ys])} & \leftarrow & \{|\mathtt{xs}| + 1\} \\ & \mathtt{app(x1s,[x|x2s],xs)}, & \{\min(|\mathtt{x1s}|, |\mathtt{xs}|)\} \\ & \quad \{|\mathtt{x1s}| + 1 + |\mathtt{x2s}| = |\mathtt{xs}|\} & \\ & \mathtt{app(x1s,x2s,zs)}, & \{\min(|\mathtt{x1s}|, |\mathtt{zs}|)\} \\ & \quad \{|\mathtt{x1s}| + |\mathtt{x2s}| = |\mathtt{zs}|\} & \\ & \mathtt{perm(zs,ys)}. & \{|\mathtt{zs}| + 1\} \end{array}$$

Using Corollary 3.8 we conclude that PERMUTATION is acceptable. Moreover, we obtain that, for a list s, the atom perm(s,t) is rigid and hence bounded. Consequently, by Corollary 3.7, all LD-derivations of PERMUTATION $\cup\{\leftarrow \mathtt{perm(s,t)}\}$ are finite.

Quicksort

Consider now the following program QUICKSORT:

```
qs(Xs, Ys)  ←  Ys is an ordered permutation of the list Xs.

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

part(X, [Y | Xs], [Y | Ls], Bs) ← X > Y, part(X, Xs, Ls, Bs).
part(X, [Y | Xs], Ls, [Y | Bs]) ← X ≤ Y, part(X, Xs, Ls, Bs).
part(X, [], [], []).
```

augmented by the APPEND program.

According to this sorting procedure, using its first element X, a list is first partitioned in two sublists, one consisting of elements smaller than X, and the other consisting of elements larger or equal than X. Then each sublist is quicksorted, and the resulting sorted sublists are appended with the element X put in the middle.

We assume that QUICKSORT 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 QUICKSORT.

Observe the following:

- QUICKSORT is not recurrent. In fact, consider the first clause instantiated with the grounding substitution

  $$\{\mathtt{X/a},\ \mathtt{Xs/b},\ \mathtt{Ys/c},\ \mathtt{Littles/[a\mid b]},\ \mathtt{Ls/c}\}.$$

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

- The clauses defining the relation part are trivially recurrent with $|\mathtt{part(x,xs,ls,bs)}| = |\mathtt{xs}|$, $|\mathtt{s} > \mathtt{t}| = 0$ and $|\mathtt{s} \leq \mathtt{t}| = 0$.

- Extend now the above level mapping with

  $$\begin{aligned} |\mathtt{qs(xs,ys)}| &= |\mathtt{xs}|, \\ |\mathtt{app(xs,ys,zs)}| &= |\mathtt{xs}|. \end{aligned}$$

  Recall that APPEND is recurrent w.r.t. $|\ |$. Next, define a Herbrand interpretation of QUICKSORT by putting

  $$\begin{aligned} I \;=\; & \{\mathtt{qs(xs,ys)} \mid |\mathtt{xs}| \geq |\mathtt{ys}|\} \\ \cup\ & \{\mathtt{part(x,xs,ls,bs)} \mid |\mathtt{xs}| \geq |\mathtt{ls}| + |\mathtt{bs}|\} \\ \cup\ & \{\mathtt{app(xs,ys,zs)} \mid |\mathtt{xs}| + |\mathtt{ys}| \geq |\mathtt{zs}|\} \\ \cup\ & [\mathtt{X} > \mathtt{Y}] \\ \cup\ & [\mathtt{X} \leq \mathtt{Y}]. \end{aligned}$$

  The following proof outlines show that QUICKSORT is acceptable w.r.t. $|\ |$ and I. The proof outlines for the non-recursive clauses are obvious and omitted.

  $$\begin{array}{ll} \{1 + |\mathtt{xs}| + |\mathtt{ys}| \geq 1 + |\mathtt{zs}|\} & \\ \mathtt{app([x|xs],ys,[x|zs])} & \leftarrow \\ & \mathtt{app(xs,ys,zs).} \\ & \quad \{|\mathtt{xs}| + |\mathtt{ys}| \geq |\mathtt{zs}|\} \end{array}$$

  $$\begin{array}{ll} \{1 + |\mathtt{xs}| \geq 1 + |\mathtt{ls}| + |\mathtt{bs}|\} & \\ \mathtt{part(x,[y|xs],[y|ls],bs)} & \leftarrow \\ & \mathtt{X > Y,} \\ & \mathtt{part(x,xs,ls,bs).} \\ & \quad \{|\mathtt{xs}| \geq |\mathtt{ls}| + |\mathtt{bs}|\} \end{array}$$

  $$\begin{array}{ll} \{1 + |\mathtt{xs}| \geq |\mathtt{ls}| + 1 + |\mathtt{bs}|\} & \\ \mathtt{part(x,[y|xs],ls,[y|bs])} & \leftarrow \\ & \mathtt{X \leq Y,} \\ & \mathtt{part(x,xs,ls,bs).} \\ & \quad \{|\mathtt{xs}| \geq |\mathtt{ls}| + |\mathtt{bs}|\} \end{array}$$

$\{1 + |\mathtt{xs}| \geq |\mathtt{ys}|\}$

$\mathtt{qs}([\mathtt{x}\|\mathtt{xs}], \mathtt{ys})$	$\leftarrow$	$\{1 + \|\mathtt{xs}\|\}$
	$\mathtt{part}(\mathtt{x}, \mathtt{xs}, \mathtt{littles}, \mathtt{bigs}),$	$\{\|\mathtt{xs}\|\}$
	$\quad \{\|\mathtt{xs}\| \geq \|\mathtt{littles}\| + \|\mathtt{bigs}\|\}$	
	$\mathtt{qs}(\mathtt{littles}, \mathtt{ls}),$	$\{\|\mathtt{littles}\|\}$
	$\quad \{\|\mathtt{littles}\| \geq \|\mathtt{ls}\|\}$	
	$\mathtt{qs}(\mathtt{bigs}, \mathtt{bs}),$	$\{\|\mathtt{bigs}\|\}$
	$\quad \{\|\mathtt{bigs}\| \geq \|\mathtt{bs}\|\}$	
	$\mathtt{app}(\mathtt{ls}, [\mathtt{x}\|\mathtt{bs}], \mathtt{ys}).$	$\{\|\mathtt{ls}\|\}$
	$\quad \{\|\mathtt{ls}\| + 1 + \|\mathtt{bs}\| \geq \|\mathtt{ys}\|\}$	

Using Corollary 3.8 we conclude that QUICKSORT is acceptable. Moreover, we obtain that, for a list s, the atom qs(s,t) is rigid and hence bounded. By Corollary 3.7 we conclude that all LD-derivations of QUICKSORT $\cup\{\leftarrow \mathtt{qs}(\mathtt{s}, \mathtt{t})\}$ are finite.

4 A Modular Approach to Termination

4.1 Drawbacks of the proof method

The proof method for (left) termination introduced in the previous sections suffers from two drawbacks.

- The level mapping used in the proof of recurrence/acceptability is sometimes different from the expected natural candidate. Consider for instance the program PERMUTATION. The relation perm is defined by induction on the length of its first argument, which is a list, and therefore a natural candidate for |perm(xs,ys)| is |xs|. Nevertheless, it is needed to add 1 to such a value in order to enforce a strict decreasing from the relation perm to the relation app, as required by the definition of acceptability.
- The proposed proof method does not provide means for constructing modular proofs, hence no straightforward technique is available to combine proofs for separate programs to obtain proofs of combined programs.

As the module hierarchy of a program becomes more complex, such adjustments of natural level mappings become more artificial and consequently more difficult to discover. For example, to prove that the program

```
overlap(Xs, Ys) :- member(X, Xs), member(X, Ys).
has_a_or_b(Xs) :- overlap(Xs, [a,b]).
```

augmented by the MEMBER program.

is recurrent we need to put |overlap(xs, ys)| = |xs|+1 to enforce the decrease in the first clause and then |has_a_or_b(xs)| = |xs|+2 to enforce the decrease in the second clause.

Both drawbacks pose serious limitations for the practical applicability of the proposed proof method. First, as argued by De Schreye, Verschaetse and Bruynooghe [SVB92], unnatural level mappings are difficult to discover by automated tools. Secondly, modularity is essential in mastering the complexity of large-scaled programs. These drawbacks share their originating cause. The notions of recurrence/acceptability are based on the fact that level mappings decrease from clause heads to clause bodies. This is used for two different purposes:

1. in (mutually) recursive calls, to ensure termination of (mutually) recursive procedures, and
2. in non (mutually) recursive calls, to ensure that non (mutually) recursive procedures are called with terminating goals.

Although a decreasing of the level mappings is apparently essential for the first purpose, this is not the case for the second purpose, since a weaker condition can be adopted to ensure that non-recursive procedures are properly called.

The next subsections will elaborate on this idea, by presenting alternative definitions of recurrence/acceptability, that we qualify with the prefix *semi*. These notions are actually proved equivalent to the original ones, but they give rise to more flexible proof methods, which avoid the cited drawbacks.

4.2 Semi-recurrent programs

Following the intuition that recursive and non-recursive procedures should be handled separately in proving termination, we introduce a natural ordering over the relation names occurring in a program P, with the intention that for relations p and q, $p \sqsupseteq q$ holds if procedure p *can call* procedure q. The next definition makes this concept precise. We denote by Π_P the set of relations occurring in a program P.

Definition 4.1 Let P be a program and p, q relations in Π_P.

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

(ii) We say that p *depends on* q *in* P, and write $p \sqsupseteq q$, if (p, q) is in the reflexive, transitive closure of the relation *refers to*. □

Observe that according to the above definition $p \simeq q \equiv p \sqsupseteq q \wedge q \sqsupseteq p$ means that p and q are mutually recursive (i.e., they are in the same recursive clique), and $p \sqsupset q \equiv p \sqsupseteq q \wedge q \not\sqsupseteq p$ means that p calls q as a subprogram. It is important to notice that the ordering $\sqsupset$ over Π_P is well-founded.

The following definition of *semi-recurrence* exploits the ordering over the relation names. The level mapping is required to decrease from an atom A in the head of a clause to an atom B in the body of that clause only if the relations of A and B are mutually recursive. Additionally, the level mapping is required not to increase from A to B if the relations of A and B are not mutually recursive. We adopt the notation $rel(A)$ to denote the relation symbol occurring in atom A.

Definition 4.2

- A clause is called *semi-recurrent with respect to a level mapping* $|\ |$, if for every ground instance $A \leftarrow \mathbf{A}, B, \mathbf{B}$ of it

  (i) $|A| > |B|$ if $rel(A) \simeq rel(B)$,

  (ii) $|A| \geq |B|$ if $rel(A) \sqsupset rel(B)$.

- A program P is called *semi-recurrent with respect to a level mapping* $|\ |$, if all its clauses are. P is called *semi-recurrent* if it is semi-recurrent with respect to some level mapping. □

The following observation is immediate.

Lemma 4.3 *If a program is recurrent w.r.t.* $|\ |$, *then it is semi-recurrent w.r.t.* $|\ |$. □

The converse of Lemma 4.3 also holds, in a sense made precise by the following result.

Lemma 4.4 *If a program is semi-recurrent w.r.t.* $|\ |$, *then it is recurrent w.r.t. a level mapping* $||\ ||$. *Moreover, for each atom* A, *if* A *is bounded w.r.t.* $|\ |$, *then* A *is bounded w.r.t.* $||\ ||$.

Proof. In order to define the level mapping $||\ ||$, we first introduce (by overloading the symbol $|\ |$) a mapping $|\ | : \Pi_P \rightarrow N$ such that, for $p, q \in \Pi_P$:

$$p \simeq q \text{ implies } |p| = |q|, \tag{1}$$

$$p \sqsupset q \text{ implies } |p| > |q|. \tag{2}$$

A mapping $|\ |$ satisfying properties (2) and (3) obviously exists, as Π_P is finite. Note that this mapping preserves the $\sqsupset$ ordering. Next, we define a level mapping $||\ ||$ for P by putting for $A \in B_P$:

$$||A|| = |A| + |rel(A)|\,. \tag{3}$$

We now prove that P is recurrent w.r.t. $||\ ||$. Let $A \leftarrow \mathbf{A}, B, \mathbf{B}$ be a ground instance of a clause from P. The following two cases arise.

Case 1 $rel(A) \simeq rel(B)$.
We calculate:

$$\begin{array}{ll} & ||A|| \\ = & \quad \{(4)\} \\ & |A| + |rel(A)| \\ > & \quad \{|A| > |B| \text{ by Definition 4.2 (i)}\} \\ & |B| + |rel(A)| \\ = & \quad \{|rel(A)| = |rel(B)| \text{ by } rel(A) \simeq rel(B) \text{ and (2)}\} \\ & |B| + |rel(B)| \\ = & \quad \{(4)\} \\ & ||B||. \end{array}$$

Case 2 $rel(A) \sqsupset rel(B)$.
We calculate:

$$\begin{array}{ll} & ||A|| \\ = & \quad \{(4)\} \\ & |A| + |rel(A)| \\ > & \quad \{|rel(A)| > |rel(B)| \text{ by } rel(A) \sqsupset rel(B) \text{ and (3)}\} \\ & |A| + |rel(B)| \\ \geq & \quad \{|A| \geq |B| \text{ by } rel(A) \sqsupset rel(B) \text{ and Definition 4.2 (ii)}\} \\ & |B| + |rel(B)| \\ = & \quad \{(4)\} \\ & ||B||. \end{array}$$

In both cases we proved $||A|| > ||B||$, which establishes the first claim. The second claim follows directly from the definition of $||\ ||$. □

The following is an immediate conclusion of Lemmata 4.3 and 4.4.

Corollary 4.5 *A program is recurrent iff it is semi-recurrent.* □

In what follows we study conditions which allow us to deduce termination of a program from termination of its components. The simplest form of program composition takes place when a program is constructed from two subprograms which use disjoint sets of relations. The following obvious composition theorem allows us to deal with this case.

Theorem 4.6 *Let P and Q be two programs such that no relation occurs in both of them. Suppose that*

- *Q is semi-recurrent w.r.t. level mapping $|\ |_Q$,*
- *P is semi-recurrent w.r.t. level mapping $|\ |_P$.*

Then $P \cup Q$ is semi-recurrent w.r.t. $|\ |$ defined as follows:

$$|A| = \begin{cases} |A|_P & \text{if } rel(A) \text{ is defined in } P, \\ |A|_Q & \text{if } rel(A) \text{ is defined in } Q. \end{cases}$$

□

Obviously, this theorem is of very limited use. We now consider a situation when a program uses another one as a subprogram. The following notion of *extension* of a program formalizes this situation.

Definition 4.7 Let P and Q be two programs.

(i) A relation p is *defined in* a program P if p occurs in a head of a clause from P.

(ii) P *extends* Q if no relation defined in P occurs in Q. □

Informally, P extends Q if P defines new (w.r.t. Q) relations. From now we assume without loss of generality that, for a given program P and a level mapping $|\ |$ for P, $|A| = 0$ if $rel(A)$ is not defined in P. Notice that such an assumption is indeed immaterial for the notion of (semi-) recurrence, since if $rel(A)$ does not occur in the head of any clause of P, then any constraint put on $|A|$ is satisfied when $|A| = 0$.

Observe that the definition of semi-recurrence allows us to compose termination proofs. Indeed, the following result holds.

Theorem 4.8 *Let P and Q be two programs such that P extends Q. Suppose that*

1. *Q is semi-recurrent w.r.t. $|\ |_Q$,*
2. *P is semi-recurrent w.r.t. $|\ |_P$,*
3. *for every ground instance $A \leftarrow \mathbf{A}, B, \mathbf{B}$ of a clause of P*

$$|A|_P \geq |B|_Q \text{ if } rel(B) \text{ is defined in } Q.$$

Then $P \cup Q$ is semi-recurrent w.r.t. $|\ |$ defined as follows:

$$|A| = \begin{cases} |A|_P & \text{if } rel(A) \text{ is defined in } P, \\ |A|_Q & \text{if } rel(A) \text{ is defined in } Q. \end{cases} \quad (4)$$

Proof. It suffices to note that for every ground instance $A \leftarrow \mathbf{A}, B, \mathbf{B}$ of a clause from $P \cup Q$ the following two implications hold:
(i) if $rel(A) \simeq rel(B)$, then either both relations are defined in P or both are defined in Q,
(ii) if $rel(A) \sqsupset rel(B)$, then either $rel(A)$ is defined in P or $rel(B)$ is not defined in P. □

This theorem suggests a natural way of composing termination proofs *provided* the level mappings of the programs P and Q satisfy condition 3. In general, it is difficult to expect that two independently constructed level mappings happen to satisfy such a relation. (An example illustrating this complication can be found below.)

Consequently, we need a more general approach. The result we now present makes possible to construct termination proofs in a modular way in full generality and is the main motivation for the introduction of the notion of semi-recurrence.

Theorem 4.9 *Let P and Q be two programs such that P extends Q. Suppose that*

1. *Q is semi-recurrent w.r.t. $|\ |_Q$,*
2. *P is semi-recurrent w.r.t. $|\ |_P$,*
3. *there exists a level mapping $||\ ||_P$ such that for every ground instance $A \leftarrow \mathbf{A}, B, \mathbf{B}$ of a clause from P*

(a) $||A||_P \geq ||B||_P$ *if* $rel(B)$ *is defined in* P,
(b) $||A||_P \geq |B|_Q$ *if* $rel(B)$ *is defined in* Q.

Then $P \cup Q$ *is semi-recurrent w.r.t.* $|\ |$ *defined as follows:*

$$|A| = \begin{cases} |A|_P + ||A||_P & \text{if } rel(A) \text{ is defined in } P, \\ |A|_Q & \text{if } rel(A) \text{ is defined in } Q. \end{cases} \quad (5)$$

Proof. It suffices to prove that each clause from P is semi-recurrent w.r.t. $|\ |$. Let $A \leftarrow \mathbf{A}, B, \mathbf{B}$ be a ground instance of a clause from P. The following two cases arise.

Case 1 $rel(A) \simeq rel(B)$.
Then by Definition 4.7, $rel(B)$ is defined in P. According to Definition 4.2(i) we need to prove $|A| > |B|$. We calculate:

$$\begin{array}{ll} & |A| \\ = & \quad \{(5)\} \\ & |A|_P + ||A||_P \\ > & \quad \{|A|_P > |B|_P \text{ by assumption 2 and Definition 4.2 (i)}, \\ & \quad \text{and } ||A||_P > ||B||_P \text{ by assumption 3(a)}\} \\ & |B|_P + ||B||_P \\ = & \quad \{(5)\} \\ & |B|. \end{array}$$

Case 2 $rel(A) \sqsupset rel(B)$.
According to Definition 4.2(ii), we need to prove $|A| \geq |B|$. Two subcases arise.

Subcase 2.1 $rel(B)$ is defined in P.
We calculate:

$$\begin{array}{ll} & |A| \\ = & \quad \{(5)\} \\ & |A|_P + ||A||_P \\ \geq & \quad \{|A|_P \geq |B|_P \text{ by assumption 2 and Definition 4.2 (ii)}, \\ & \quad \text{and } ||A||_P \geq ||B||_P \text{ by assumption 3(a)}\} \\ & |B|_P + ||B||_P \\ = & \quad \{(5)\} \\ & |B|. \end{array}$$

Subcase 2.2 $rel(B)$ is defined in Q.
We calculate:

$$\begin{array}{ll} & |A| \\ = & \quad \{(5)\} \\ & |A|_P + ||A||_P \\ \geq & \quad \{||A||_P \geq |B|_Q \text{ by assumption 3(b)}\} \\ & |A|_P + |B|_Q \\ \geq & |B|_Q \\ = & \quad \{(5)\} \\ & |B|. \end{array}$$

□

4.3 Methodology

Theorems 4.6,4.8 and 4.9 provide us with an incremental, bottom-up method for proving termination of logic programs. Given a program P, the method can be informally illustrated as follows.

1. Partition the relation names in P in the equivalence classes w.r.t. the equivalence $\simeq$ induced by the depends on relation $\sqsupseteq$. Such equivalence classes correspond to the recursive cliques of program P. Let $P_1, \ldots, P_n$ be the partition of the clauses from P such that each P_i contains the clauses defining the relation(s) belonging to the same equivalence class. The relation $\sqsupset$ defined on the relations induces a corresponding well- founded ordering $>$ on the programs P_i:

   $P_i > P_j$ iff $p \sqsupset q$ for some p defined in P_i and q defined in P_j.

2. Prove by induction w.r.t. the ordering $>$ that for every program P_i, $i \in [1, n]$ the program $P_i \cup \bigcup_{P_j < P_i} P_j$ is semi-recurrent.

   The base case. Consider all P_i, $i \in [1, n]$, which are minimal w.r.t. $>$.

   - Prove that each such P_i is semi-recurrent (w.r.t. some $|\ |_{P_i}$).
     Notice that this is the same as proving that P_i is recurrent w.r.t. $|\ |_{P_i}$, as procedures in P_i do not call any subprograms.

   The induction step. Consider a P_i, $i \in [1, n]$, such that all P_j for which $P_j < P_i$ have already been proved semi-recurrent.

   - Prove that P_i (in isolation) is recurrent w.r.t. some $|\ |_{P_i}$.
     Notice that the assumption that $|A|_{P_i} = 0$ if $rel(A)$ is not defined in P_i allows us to abstract from the relations that are not defined in P_i. Consequently, we only need to prove that $|\ |_{P_i}$ decreases on (mutually) recursive calls. This facilitates the choice of a "natural" candidate for $|\ |_{P_i}$, which directly mirrors the inductive structure of the procedures defined in P_i.
   - Use Theorem 4.6 to conclude that $\bigcup_{P_j < P_i} P_j$ is semi-recurrent.
   - Use Theorem 4.8 or Theorem 4.9 to prove that $P_i \cup \bigcup_{P_j < P_i} P_j$ is semi-recurrent.
     Here we only need to come up with a level mapping $||\ ||$ which is usually directly suggested by the level mappings $|\ |_{P_j}$, where $P_j < P_i$.

4.4 Examples

Mergesort

Consider the following program `MERGESORT` which is an instance of the divide and conquer schema:

```
ms(Xs, Ys) ← Ys is an ordered permutation of the list Xs.

ms([X, Y | Xs], Ys) ←
   split([X, Y | Xs], X1s, X2s),
   ms(X1s, Y1s),
   ms(X2s, Y2s),
   merge(Y1s, Y2s, Ys).
ms([X], [X]).
ms([], []).

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

merge([X | Xs], [Y | Ys], [X | Zs]) ←
```

```
    X ≤ Y,
    merge(Xs,[Y | Ys], Zs).
merge([X | Xs], [Y | Ys], [Y | Zs]) ←
    X > Y,
    merge([X | Xs], Ys, Zs).
merge([], Xs, Xs).
merge(Xs, [], Xs).
```

According to this sorting procedure, a list of length at least 2 is first split in two lists of roughly equal length (by means of the reversed order of parameters in the recursive call of `split`), then each sublist is mergesorted, and finally the resulting sorted sublists are merged, preserving the ordering.

Note that MERGESORT is not recurrent. Indeed, due to the introduction of the local variables `X1`,`X2`,`Y1`, `Y2` in the body of the recursive clause defining `ms`, it is not terminating. By adding an additional parameter Bezem [Bez93] modified this program so that it becomes terminating:

```
ms(Xs, Ys, Xs) ← Ys is an ordered permutation of the list Xs.

ms([X, Y | Xs], Ys, [H | Ls]) ←
    split([X, Y | Xs], X1s, X2s, [H | Ls]),
    ms(X1s, Y1s, Ls),
    ms(X2s, Y2s, Ls),
    merge(Y1s, Y2s, Ys, [H | Ls]).
ms([X], [X], Ls).
ms([], [], Ls).

split([X | Xs], [X | Ys], Zs, [H | Ls]) ← split(Xs, Zs, Ys, Ls).
split([], [], [], Ls).

merge([X | Xs], [Y | Ys], [X | Zs], [H | Ls]) ←
    X ≤ Y,
    merge(Xs,[Y | Ys], Zs, Ls).
merge([X | Xs], [Y | Ys], [Y | Zs], [H | Ls]) ←
    X > Y,
    merge([X | Xs], Ys, Zs, Ls).
merge([], Xs, Xs, Ls).
merge(Xs, [], Xs, Ls).
```

(A misprint leaked in to Bezem [Bez93] where instead of calling `merge`, `ms` calls itself.) We prove this fact using Theorems 4.6 and 4.8. Call the above program MERGESORT' and denote the subprograms of MERGESORT' which define the relations `ms`, `split` and `merge` by MS, SPLIT and MERGE, correspondingly. Thanks to the addition of the last argument MS is recurrent w.r.t. the level mapping

$$|\mathtt{ms}(\mathtt{xs},\mathtt{ys},\mathtt{ls})| = |\mathtt{ls}|,$$

SPLIT is recurrent w.r.t. the level mapping

$$|\mathtt{split}(\mathtt{xs},\mathtt{ys},\mathtt{zs},\mathtt{ls})| = |\mathtt{ls}|,$$

and MERGE is recurrent w.r.t. the level mapping

$$|\mathtt{merge}(\mathtt{xs},\mathtt{ys},\mathtt{zs},\mathtt{ls})| = |\mathtt{ls}|.$$

By Theorem 4.6 SPLIT ∪ MERGE is recurrent w.r.t. $|\ |$. Assumption 3 of Theorem 4.8 applied to the programs MS and SPLIT ∪ MERGE is obviously satisfied, so we conclude by this theorem that MERGESORT' is semi-recurrent w.r.t. $|\ |$, and hence terminating.

To prove this fact Bezem [Bez93] used the concept of a recurrent program which to deal with the subprogram calls in the recursive clause defining `ms` requires a more artificial level mapping in which `|ms(xs, ys, ls)| = |ls|+1`.

Curry's type assignment

Consider the following program for Curry's type assignment (see e.g. Reddy [Red86]). In Curry's type system, a *type assignment* `E ⊢ M : T` expresses the fact that λ-term `M` is assigned type `T` w.r.t. environment `E`. Here, λ-terms are represented using the function symbols `var` (for variables), `apply` (for application), and `lambda` (for λ-abstraction). Type terms are represented using the function symbol `arrow` (for the function type). For the sake of concreteness, we augment the program with extra constants (say `v, w, z`) representing λ-variables, and others (say `Nat, Bool`) representing basic types. Finally, environments are represented as lists of pairs (λ-variable, type term).

```
      type(E,M,T) ← E ⊢ M : T

(t1)  type(E,var(X),T) ← in(E,X,T).
(t2)  type(E,apply(M,N),T) ← type(E,M,arrow(S,T)), type(E,N,S).
(t3)  type(E,lambda(X,M),arrow(S,T)) ← type([(X,S)|E],M,T).

      in(E,X,T) ← X is bound to T in E

(i1)  in([(X,T)|E],X,T).
(i2)  in([(Y,T)|E],X,T) ← X ≠ Y, in(E,X,T).
```

Denote by `CURRY` the program formed by clauses t_1, t_2 and t_3, and by `ENV` the program formed by clauses i_1 and i_2. Clearly, `CURRY` extends `ENV`, and `type`$\square$ `in` $\square \neq$ `in curry` $\cup$ `env`. Observe the following:

- relation `in` is defined by induction on the length of its first argument, which is a list. As a result, the program `ENV` is recurrent w.r.t. $|\ |_{ENV}$ defined as:

$$|\mathtt{in(e,x,t)}|_{ENV} = |\mathtt{e}|.$$

- Relation `type` is defined by induction on the size of its first argument, which is a λ-term. As a result, the program `CURRY` is recurrent w.r.t. $|\ |_{CURRY}$ defined as:

$$|\mathtt{type(e,m,t)}|_{CURRY} = \mathrm{size}(\mathtt{m}).$$

- In any derivation starting from a goal $\leftarrow$ `type(e,m,t)`, the length of the environment is bounded by `|e|`+size(`m`), since the length of the environment is incremented together with the decrease of the size of the λ-term (clause t_3). As a result, by defining

$$||\mathtt{type(e,m,t)}||_{CURRY} = |\mathtt{e}| + \mathrm{size}(\mathtt{m})$$

  we satisfy for $||\ ||_{CURRY}$ the assumptions 3(a) and (b) of Theorem 4.9.

  Note that the level mappings $|\ |_{ENV}$ and $|\ |_{CURRY}$ do not satisfy condition 3 of Theorem 4.8, so this theorem cannot be used here.

As a consequence, by Theorem 4.9, Lemma 4.4 and Corollary 2.6 we conclude that `CURRY` $\cup$ `ENV` is terminating. Additionally, we obtain that a goal $\leftarrow$ `type(e,m,t)` is bounded if `e` is a list and `m` is ground. This latter result is relevant, since it justifies the fact that program `CURRY` $\cup$ `ENV` can be used to implement type *inference* by means of the goals of the kind $\leftarrow$ `type(e,m,T)`, where `e` is a list, `m` is a ground λ-term, and `T` is a variable.

As a final remark, notice that it is possible to arrive at the same conclusion by showing directly that `CURRY` $\cup$ `ENV` is recurrent w.r.t. the level mapping $|\mathtt{type(e,m,t)}| = |\mathtt{e}| + 2 \times \mathrm{size}(\mathtt{m})$, but such a level mapping is unnatural. Moreover, such a proof cannot be readily explained in a compositional way, as a combination of the separate proofs for `CURRY` and `ENV`.

A relational `MAP` program

Consider the following program `MAP`, implementing a relational equivalent of the ubiquitous higher-order combinator *map* of functional programming:

`map([X1, ..., Xn], [Y1, ..., Yn])` $\leftarrow$ `p(Xi, Yi)` holds for `i` $\in$ `[1,n]`.

`map([X|Xs], [Y|Ys])` $\leftarrow$ `p(X, Y), map(Xs, Ys).`
`map([], []).`

The program `MAP` is parametric w.r.t. relation `p`. Let `P` be a program defining the relation `p`, such that `MAP` extends `P` (hence: `map` $\sqsupset$ `p`.) Assume that `P` is recurrent w.r.t. $|\ |_P$ defined as $|\mathtt{p(x,y)}|_P = \mathtt{f(x)}$, where `f(x)` denotes some function assigning natural numbers to ground terms.

We observe the following:

- The program `MAP` is trivially recurrent w.r.t. $|\ |_{MAP}$ defined by

$$|\mathtt{map(xs,ys)}|_{MAP} \quad = \quad |\mathtt{xs}|.$$

- Define $||\ ||_{MAP}$ by recursion as follows:

$$\begin{aligned} ||\mathtt{map([\,],ys)}||_{MAP} &= 0, \\ ||\mathtt{map([x|xs],ys)}||_{MAP} &= \mathtt{f(x)} + ||\mathtt{map(xs,ys)}||_{MAP}. \end{aligned}$$

Assumption 3 of Theorem 4.9 is satisfied by $||\ ||_{MAP}$. Indeed, consider a ground instance

$$\mathtt{map([x|xs],[y|ys])} \leftarrow \mathtt{p(x,y)}, \mathtt{map(xs,ys)}.$$

of the recursive clause of program `MAP`, and observe that:

$||\mathtt{map([x|xs],[y|ys])}||_{MAP} = \mathtt{f(x)} + ||\mathtt{map(xs,[y|ys])}||_{MAP} \geq \mathtt{f(x)} = |\mathtt{p(x,y)}|_P,$

$||\mathtt{map([x|xs],[y|ys])}||_{MAP} = \mathtt{f(x)} + ||\mathtt{map(xs,[y|ys])}||_{MAP}$

$\geq ||\mathtt{map(xs,[y|ys])}||_{MAP} = ||\mathtt{map(xs,ys)}||_{MAP}.$

By Theorem 4.9 we conclude that `MAP`$\cup$`P` is recurrent. Moreover, we obtain that a goal $\leftarrow$ `map(xs,ys)` is bounded if `xs` is a list of terms each of which is bounded w.r.t. `f`. (As expected, a term `t` is bounded w.r.t `f` if `f` is bounded on the set of ground instances of `t`.) Thus we obtained a modular proof scheme for the parametric program `MAP` .

Note that there is no relationship between $|\mathtt{map([x|xs],[y|ys])}|_{MAP}$ which equals $|\mathtt{xs}| + 1$ and $|\mathtt{p(x,y)}|_P$ which equals `f(x)`, so with this natural choice of level mappings we cannot apply here Theorem 4.8.

5 A Modular Approach to Left Termination

5.1 Semi-acceptable programs

An analogous modification of the notion of acceptability yields a modular approach to the proofs of left termination.

Definition 5.1 Let P be a program, $|\ |$ a level mapping for P and I a (not necessarily Herbrand) interpretation of P.

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

  (i) $|A| > |B|$ if $rel(A) \simeq rel(B)$,

  (ii) $|A| \geq |B|$ if $rel(A) \sqsupset rel(B)$.

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

Again, the use of the premise $I \models \mathbf{A}$ forms the *only* difference between the concepts of semi-recurrence and semi-acceptability.

The following observations are immediate. The first one is a counterpart of Lemma 3.4.

Lemma 5.2 *A program is semi-recurrent w.r.t. $|\ |$ iff it is semi-acceptable w.r.t. $|\ |$ and B_P.* □

Lemma 5.3 *If a program is acceptable w.r.t. $|\ |$ and I, then it is semi-acceptable w.r.t. $|\ |$ and I.* □

Also, the proof of Lemma 4.4 can be literally viewed as a proof of the following analogous result for semi-acceptable programs.

Lemma 5.4 *If a program is semi-acceptable w.r.t. $|\ |$ and I, then it is acceptable w.r.t. a level mapping $\|\ \|$ and the same interpretation I. Moreover, for each atom A, if A is bounded w.r.t. $|\ |$, then A is bounded w.r.t. $\|\ \|$.* □

The following is a direct consequence of Lemmata 5.3 and 5.4.

Corollary 5.5 *A program is acceptable iff it is semi-acceptable.* □

Let us consider now the issue of modularity. The following is an analogue of Theorem 4.6 for semi-acceptable programs.

Theorem 5.6 *Let P and Q be two programs such that no relation occurs in both of them. Suppose that*

- *Q is semi-acceptable w.r.t. level mapping $|\ |_Q$ and interpretation I_Q,*
- *P is semi-acceptable w.r.t. level mapping $|\ |_P$ and interpretation I_P.*

Then $P \cup Q$ is semi-recurrent w.r.t. $|\ |$ and $I_P \cup I_Q$, where $|\ |$ is defined as follows:

$$|A| = \begin{cases} |A|_P & \text{if } rel(A) \text{ is defined in } P, \\ |A|_Q & \text{if } rel(A) \text{ is defined in } Q. \end{cases}$$

□

Next, note the following analogue of Theorem 4.8 for semi-acceptable programs.

Theorem 5.7 *Let P and Q be two programs such that P extends Q. Suppose that*

1. *Q is semi-acceptable w.r.t. $|\ |_Q$ and $I_P \cap B_Q$,*
2. *P is semi-acceptable w.r.t. $|\ |_P$ and I_P,*
3. *for every ground instance $A \leftarrow \mathbf{A}, B, \mathbf{B}$ of a clause of P such that $I_P \models \mathbf{A}$*

$$|A|_P \geq |B|_Q \text{ if } rel(B) \text{ is defined in } Q.$$

Then $P \cup Q$ is semi-acceptable w.r.t. $|\ |$ and I_P, where $|\ |$ is defined as follows:

$$|A| = \begin{cases} |A|_P & \text{if } rel(A) \text{ is defined in } P, \\ |A|_Q & \text{if } rel(A) \text{ is defined in } Q. \end{cases} \tag{7}$$

Proof. The proof is identical to that of Theorem 4.8. □

As in the case of semi-recurrent programs we cannot always hope that two unrelated level mappings satisfy condition 3 of this theorem. The following analogue of Theorem 4.9 for semi-acceptable programs deals with this difficulty.

Theorem 5.8 *Let P and Q be two programs such that P extends Q, and let I_P be a model of $P \cup Q$. Suppose that*

1. *Q is semi-acceptable w.r.t. $|\ |_Q$ and $I_P \cap B_Q$,*
2. *P is semi-acceptable w.r.t. $|\ |_P$ and I_P,*
3. *there exists a level mapping $||\ ||_P$ such that for every ground instance $A \leftarrow \mathbf{A}, B, \mathbf{B}$ of a clause from P such that $I_P \models \mathbf{A}$*
   (a) $||A||_P \geq ||B||_P$ if $rel(B)$ is defined in P,
   (b) $||A||_P \geq |B|_Q$ if $rel(B)$ is defined in Q.

Then $P \cup Q$ is semi-acceptable w.r.t. $|\ |$ and I_P, where $|\ |$ is defined as follows:

$$|A| = \begin{cases} |A|_P + ||A||_P & \text{if } rel(A) \text{ is defined in } P, \\ |A|_Q & \text{if } rel(A) \text{ is defined in } Q. \end{cases}$$

Proof. The proof is identical to that of Theorem 4.9. □

5.2 Examples

We now present some applications of the modular method for proving left termination. In the following we adopt proof outlines also as a proof format for the verification of assumption 3 of Theorems 5.7 and 5.8. We refer to such proof outlines with the qualification *weak*, and assume that for weak proof outlines condition 4 of Section 3.3 is amended as follows, by replacing $>$ by $\geq$:

$$4'. \text{ For } i \in [1,n]: \ f_1 \wedge \ldots \wedge f_{i-1} \Rightarrow t_0 \geq t_i.$$

Permutation

Reconsider the program PERMUTATION:

```
perm(Xs, Ys) ← Ys is a permutation of the list Xs.

perm(Xs, [X | Ys]) ←
   app(X1s, [X | X2s], Xs),
   app(X1s, X2s, Zs),
   perm(Zs, Ys).
perm([], []).
```

augmented by the APPEND program.

Denote the program defining the PERMUTATION relation by PERM. Clearly, PERM extends APPEND, and perm $\sqsupset$ app. Recall that APPEND is recurrent w.r.t. |app(xs, ys, zs)| = min(|xs|, |zs|). Observe the following:

- the relation perm is defined by induction on the length of its first argument. Indeed, the program PERM is semi-acceptable w.r.t. $|\ |$ and I_{PERM} defined by:

$$|\mathtt{perm(xs,\ ys)}| = |\mathtt{xs}|,$$

$$I_{PERM} = [\mathtt{perm(Xs, Ys)}] \cup \{\mathtt{app(xs, ys, zs)} \mid |\mathtt{xs}| + |\mathtt{ys}| = |\mathtt{zs}|\}.$$

The proof that I_{PERM} is a model of APPEND is as in Section 3.3. The following is a proof outline for the semi-acceptability of the recursive clause for perm w.r.t. $|\ |$ and I_{PERM}.

$$\begin{array}{lll} \mathtt{perm(xs,[x|ys])} & \leftarrow & \{|\mathtt{xs}|\} \\ & \mathtt{app(x1s,[x|x2s],xs)}, & \\ & \quad \{|\mathtt{xs}| = |\mathtt{x1s}| + |\mathtt{x2s}| + 1\} & \\ & \mathtt{app(x1s,x2s,zs)}, & \\ & \quad \{|\mathtt{zs}| = |\mathtt{x1s}| + |\mathtt{x2s}|\} & \\ & \mathtt{perm(zs,ys)}. & \{|\mathtt{zs}|\} \end{array}$$

- Assumption 3 of Theorem 5.7 is satisfied as the following weak proof outline shows:

$$\begin{array}{lll} \mathtt{perm(xs,[x|ys])} & \leftarrow & \{|\mathtt{xs}|\} \\ & \mathtt{app(x1s,[x|x2s],xs)}, & \{\mathtt{min(x1s,xs)}\} \\ & \quad \{|\mathtt{xs}| = |\mathtt{x1s}| + |\mathtt{x2s}| + 1\} & \\ & \mathtt{app(x1s,x2s,zs)}, & \{\mathtt{min(x1s,zs)}\} \\ & \quad \{|\mathtt{zs}| = |\mathtt{x1s}| + |\mathtt{x2s}|\} & \\ & \mathtt{perm(zs,ys)}. & \end{array}$$

Hence, by Theorem 5.7 and Lemma 5.4 we conclude that PERMUTATION = PERM $\cup$ APPEND is acceptable w.r.t. $|\ |$ and I_{PERM}. We thus achieved the same result of Section 3.3, but in a modular way, and using a more natural level mapping for perm.

A divide & conquer scheme

Reconsider the *divide and conquer* schema DC which is parametric with respect to the base, conquer, divide and merge relations:

```
dcsolve(X, Y) ←
   base(X),
   conquer(X, Y).
dcsolve(X, Y) ←
   divide(X, X0, X1, X2),
   dcsolve(X1, Y1),
   dcsolve(X2, Y2),
   merge(X0, Y1, Y2, Y).
```

Let P be a program defining the relations base, conquer, divide and merge. Clearly, DC extends P, and dcsolve $\sqsupset$ base, conquer, divide, merge. Assume that P is acceptable w.r.t. $|\ |_P$ and I_P defined as follows:

$$\begin{array}{rcl} |\mathtt{base(x)}|_P & = & ||\mathtt{x}||, \\ |\mathtt{conquer(x,y)}|_P & = & ||\mathtt{x}||, \\ |\mathtt{divide(x,x0,x1,x2)}|_P & = & ||\mathtt{x}||, \\ |\mathtt{merge(x0,y1,y2,y)}|_P & = & ||\mathtt{x0}|| + ||\mathtt{y1}|| + ||\mathtt{y2}||, \end{array}$$

$$\begin{array}{rll} I_P = & [\mathtt{base(X)}] & \\ \cup & \{\mathtt{conquer(x,y)} \mid ||\mathtt{x}|| \geq ||\mathtt{y}||\} & \\ \cup & \{\mathtt{divide(x,x0,x1,x2)} \mid ||\mathtt{x}|| \geq ||\mathtt{x0}|| + ||\mathtt{x1}|| + ||\mathtt{x2}|| \wedge ||\mathtt{x}|| > ||\mathtt{x1}|| \wedge ||\mathtt{x}|| > ||\mathtt{x2}||\} & \\ \cup & \{\mathtt{merge(x0,y1,y2,y)} \mid ||\mathtt{y}|| \leq ||\mathtt{x0}|| + ||\mathtt{y1}|| + ||\mathtt{y2}||\}, & \end{array}$$

where $||\ ||$ denotes some function assigning natural numbers to ground terms.

Notice that these assumptions are quite natural for a large class of programs following the divide and conquer paradigm.

We observe the following:

- the program DC is acceptable w.r.t. $|\ |_{DC}$ and I_{DC} defined by:

$$|\mathtt{dcsolve(x,y)}|_{DC} \;=\; ||\mathtt{x}||,$$

$$I_{DC} \;=\; I_P \cup \{\mathtt{dcsolve(x,y)} \mid ||\mathtt{x}|| \geq ||\mathtt{y}||\}.$$

The proof outline for the non-recursive clause of DC is obvious. For the recursive clause take the following proof outline:

$$\begin{array}{lll}
\{||\mathtt{x}|| \geq ||\mathtt{y}||\} & & \\
\mathtt{dcsolve(x,y)} & \leftarrow & \{||\mathtt{x}||\} \\
 & \mathtt{divide(x,x0,x1,x2)}, & \\
 & \quad \{||\mathtt{x}|| \geq ||\mathtt{x0}|| + ||\mathtt{x1}|| + ||\mathtt{x2}|| & \\
 & \quad\ \wedge\ ||\mathtt{x}|| > ||\mathtt{x1}|| \wedge ||\mathtt{x}|| > ||\mathtt{x2}||\} & \\
 & \mathtt{dcsolve(x1,y1)}, & \{||\mathtt{x1}||\} \\
 & \quad \{||\mathtt{x1}|| \geq ||\mathtt{y1}||\} & \\
 & \mathtt{dcsolve(x2,y2)}, & \{||\mathtt{x2}||\} \\
 & \quad \{||\mathtt{x2}|| \geq ||\mathtt{y2}||\} & \\
 & \mathtt{merge(x0,y1,y2,y)}. & \\
 & \quad \{||\mathtt{x0}|| + ||\mathtt{y1}|| + ||\mathtt{y2}|| \geq ||\mathtt{y}||\} &
\end{array}$$

- Assumption 3 of Theorem 5.7 is satisfied as the following weak proof outlines show:

$$\begin{array}{lll}
\mathtt{dcsolve(x,y)} & \leftarrow & \{||\mathtt{x}||\} \\
 & \mathtt{base(x)}, & \{||\mathtt{x}||\} \\
 & \mathtt{conquer(x,y)}. & \{||\mathtt{x}||\}
\end{array}$$

$$\begin{array}{lll}
\mathtt{dcsolve(x,y)} & \leftarrow & \{||\mathtt{x}||\} \\
 & \mathtt{divide(x,x0,x1,x2)}, & \{||\mathtt{x}||\} \\
 & \quad \{||\mathtt{x}|| \geq ||\mathtt{x0}|| + ||\mathtt{x1}|| + ||\mathtt{x2}|| & \\
 & \quad\ \wedge\ ||\mathtt{x}|| > ||\mathtt{x1}|| \wedge ||\mathtt{x}|| > ||\mathtt{x2}||\} & \\
 & \mathtt{dcsolve(x1,y1)}, & \\
 & \quad \{||\mathtt{x1}|| \geq ||\mathtt{y1}||\} & \\
 & \mathtt{dcsolve(x2,y2)}, & \\
 & \quad \{||\mathtt{x2}|| \geq ||\mathtt{y2}||\} & \\
 & \mathtt{merge(x0,y1,y2,y)}. & \{||\mathtt{x0}|| + ||\mathtt{y1}|| + ||\mathtt{y2}||\}
\end{array}$$

Using Theorem 5.7 and Lemma 5.4 we conclude that DC ∪ P is acceptable w.r.t. $|\ |$ and I_{DC}, where

$$|A| \;=\; \begin{cases} |A|_{DC} & \text{if } rel(A) = \mathtt{dcsolve}, \\ |A|_P & \text{otherwise.} \end{cases}$$

Moreover, we obtain that a goal $\leftarrow \mathtt{dcsolve(x,y)}$ is bounded if $\mathtt{dcsolve(x,y)}$ rigid, so in particular if x is ground. Thus we obtained a modular proof scheme for divide and conquer programs.

As a direct application, note that the program QUICKSORT can be defined as QUICKSORT = DC ∪ P by putting

$$\mathtt{qs} \equiv \mathtt{dcsolve},$$

and defining P as follows:

```
base([]).

conquer([], []).

divide([X|Xs], [X], Littles, Bigs) ←
   part(X, Xs, Littles, Bigs).

merge([X], Ls, Bs, Ys) ←
   app(Ls, [X|Bs], Ys).
```

It is easy to check that P satisfies the conditions of the presented proof scheme for DC, and thus we can directly conclude that QUICKSORT is left terminating, and that for a list s, all LD-derivations of QUICKSORT $\cup$ { $\leftarrow$ qs(s, t)} are finite. (To be more precise, we obtain QUICKSORT from the above program by unfolding in the sense of Tamaki and Sato [TS84].)

A generate & test scheme

Consider the following one-clause program GT, representing a *generate and test* schema; it is parametric with respect to the generate and test relations.

```
gtsolve(X, Y) ←
   generate(X, Y),
   test(Y).
```

Let P be a program defining the relations generate and test. Clearly, GT extends P, and gtsolve $\sqsupset$ generate, test. Assume that P is acceptable w.r.t. $|\ |_P$ and I_P defined as follows, where, as before, $||\ ||$ denotes a function assigning natural numbers to ground terms.

$$\begin{aligned} |\mathtt{generate(x,y)}|_P &= ||\mathtt{x}||, \\ |\mathtt{test(y)}|_P &= ||\mathtt{y}||, \end{aligned}$$

$$\begin{aligned} I_P|\{\mathtt{test}, \mathtt{generate}\} = \quad & [\mathtt{test(Y)}] \\ \cup \quad & \{\mathtt{generate(x,y)} \mid ||\mathtt{x}|| \geq ||\mathtt{y}||\}. \end{aligned}$$

Here for a Herbrand interpretation I and a set of relations R, we denote by $I|R$ the restriction of I to the relations belonging to R.

We observe the following:

- the program GT is trivially semi-recurrent w.r.t. any level mapping. In fact, the only clause of GT is non-recursive.
- Define $|\ |_{GT}$ and I_{GT} as follows:

$$\begin{aligned} |\mathtt{gtsolve(x,y)}|_{GT} &= ||\mathtt{x}||, \\ I_{GT} &= I_P \cup [\mathtt{gtsolve(x,y)}]. \end{aligned}$$

  Assumption 3 of Theorem 5.7 is satisfied by $|\ |_P$, $|\ |_{GT}$ and I_{GT}, as the following weak proof outline shows.

$$\begin{array}{lll} \mathtt{gtsolve(x,y)} & \leftarrow & \{||\mathtt{x}||\} \\ & \mathtt{generate(x,y)}, & \{||\mathtt{x}||\} \\ & \quad \{||\mathtt{x}|| \geq ||\mathtt{y}||\} & \\ & \mathtt{test(y)}. & \{||\mathtt{y}||\} \end{array}$$

By Theorem 5.7 and Lemma 5.4 we conclude that GT $\cup$ P is acceptable w.r.t. $|\ |$ and I_{GT}, where

$$|A| = \begin{cases} |A|_{GT} & \text{if } rel(A) = \mathtt{gtsolve}, \\ |A|_P & \text{otherwise.} \end{cases}$$

Moreover, we obtain that a goal $\leftarrow$ gtsolve(x, y) is bounded if gtsolve(x, y) is rigid, so in particular if x is ground. Thus we obtained a modular proof scheme for generate and test programs.

As a direct application, consider the program SLOWSORT = GT $\cup$ P, obtained by putting

$$\begin{aligned} \mathtt{ss} &\equiv \mathtt{gtsolve}, \\ \mathtt{generate} &\equiv \mathtt{perm}, \\ \mathtt{test} &\equiv \mathtt{ordered}, \end{aligned}$$

and

$$\mathtt{P} = \mathtt{PERMUTATION} \cup \mathtt{ORDERED},$$

where ORDERED is defined by

`ordered(Xs)` $\leftarrow$ `Xs` is an $\leq$-ordered list of natural numbers.

```
ordered([]).
ordered([X]).
ordered([X, Y | Xs]) ← X ≤ Y, ordered([Y| Xs]).
```

ORDERED is clearly recurrent w.r.t. the level mapping $|\texttt{ordered(ys)}| = |\texttt{ys}|$, so acceptable w.r.t. $|\ |$ and `[ordered(XS)]`. By Theorem 5.6 PERMUTATION $\cup$ ORDERED is acceptable w.r.t. $|\ |$ defined by

$$\begin{aligned} |\texttt{perm(xs,ys)}|_P &= |\texttt{xs}|, \\ |\texttt{ordered(ys)}|_P &= |\texttt{ys}|, \end{aligned}$$

and the model `[ordered(XS)]` $\cup$ I_{PERM}.

Thus we can directly conclude that SLOWSORT is left terminating and that for a list `s`, all LD-derivations of SLOWSORT $\cup$ $\{\leftarrow \texttt{ss(s,t)}\}$ are finite.

A relational `fold` program

Consider the following program FOLD which implements a relational equivalent of the higher-order combinator *fold-left* of functional programming. The program FOLD is parametric w.r.t. relation `op`. We assume that `op` is the relational equivalent of a binary operator *op*, in the sense that `op(x,y,z)` holds iff `z` = (`x` *op* `y`).

```
fold(X, [Y1, ..., Yn], Z) ← Z = ( ... ((X  op  Y1)  op  Y2)  ... op  Yn)

fold(X, [Y | Ys], Z) ← op(X, Y, V), fold(V, Ys, Z).
fold(X, [ ], X).
```

Let OP be a program defining the relation `op`, such that FOLD extends OP (hence: `fold` $\sqsupset$ `op`.) Assume that OP is acceptable w.r.t. $|\ |_{OP}$ and I_{OP} satisfying the following properties:

$$|\texttt{op(x,y,z)}|_{OP} = \texttt{f(x)} + \texttt{g(y)}$$

$$I_{OP}|\{\texttt{op}\} = \{\texttt{op(x,y,z)} \mid \texttt{f(x)} + \texttt{g(y)} \geq \texttt{f(z)}\},$$

where `f`, `g` denote some functions assigning natural numbers to ground terms.

We observe the following:

- the program FOLD is trivially recurrent w.r.t. $|\ |_{FOLD}$ defined by

  $$|\texttt{fold(x,ys,z)}|_{FOLD} = |\texttt{ys}|.$$

  By Lemma 3.4 FOLD is acceptable w.r.t. $|\ |_{FOLD}$ and $I_{FOLD} = I_{OP} \cup [\texttt{fold(x,ys,z)}]$.

- Define a function $||\ ||$ assigning natural numbers to ground terms by recursion as follows:

  $$\begin{aligned} ||[\texttt{y}|\texttt{ys}]|| &= \texttt{g(y)} + ||\texttt{ys}||, \\ ||\texttt{x}|| &= 0 \text{ otherwise.} \end{aligned}$$

  Assumption 3 of Theorem 5.8 is satisfied by putting:

  $$||\texttt{fold(x,ys,z)}||_{FOLD} = \texttt{f(x)} + ||\texttt{ys}||,$$

  and using I_{FOLD} defined before. Indeed, the weak proof outline for the non-recursive clause of FOLD is obvious and for the recursive clause we have the following weak proof outline:

  $$\begin{array}{lll} \texttt{fold(x,[y|ys],z)} & \leftarrow & \{\texttt{f(x)} + \texttt{g(y)} + ||\texttt{ys}||\} \\ & \texttt{op(x,y,v)}, & \{\texttt{f(x)} + \texttt{g(y)}\} \\ & \quad \{\texttt{f(x)} + \texttt{g(y)} \geq \texttt{f(v)}\} & \\ & \texttt{fold(v,ys,z)}. & \{\texttt{f(v)} + ||\texttt{ys}||\} \end{array}$$

By Theorem 5.8 we conclude that FOLD ∪ OP is acceptable. Moreover, we obtain that a goal

$$\leftarrow \texttt{fold}(\texttt{x}, \texttt{ys}, \texttt{z})$$

is bounded if x is bounded w.r.t. f, and ys is a list of terms each of which is bounded w.r.t. g. Thus we obtained a modular proof scheme for the parametric program FOLD. Notice that with the above choice of the level mappings we cannot apply here Theorem 5.7, since $|\ |_{FOLD}$ is unrelated to $|\ |_{OP}$, whereas $||\ ||_{FOLD}$ does not need to decrease in recursive calls.

As a direct application, consider the program SUMLIST = FOLD ∪ OP, obtained by putting OP = SUM, where SUM is defined as in Section 2.3, and

$$\texttt{op} \equiv \texttt{sum}.$$

It is easy to check that OP satisfies the conditions of the presented proof scheme for FOLD, by putting:

$$\begin{aligned} \texttt{f}(\texttt{x}) &= 0, \\ \texttt{g}(\texttt{x}) &= \text{size}(\texttt{x}). \end{aligned}$$

Thus we can directly conclude that SUMLIST is acceptable, and that, for a ground ys, all LD-derivations of a goal $\leftarrow \texttt{fold}(\texttt{x}, \texttt{ys}, \texttt{z})$ w.r.t. SUMLIST are finite. Note that the goal $\leftarrow \texttt{fold}(0, \texttt{ys}, \texttt{z})$ computes the sum of the elements of the list ys.

The MAP program revisited

Reconsider the program MAP:

```
map([X1, ..., Xn], [Y1, ..., Yn]) ← p(Xi, Yi) holds for i ∈ [1,n].

map([X|Xs], [Y|Ys]) ← p(X, Y), map(Xs, Ys).
map([], []).
```

Relax the assumptions made in Section 4.4 on P by assuming that P is acceptable w.r.t. $|\ |_P$ defined as in Section 4.4, and *any* model I of P. It is immediate to observe that the proof outlines of Section 4.4 remain valid with the new assumptions. Hence, by Theorem 5.8 we conclude that MAP ∪ P is acceptable; moreover, we obtain the same class of bounded goals as in Section 4.4. Again, we cannot apply here Theorem 5.7, since the level mappings for map and p are unrelated.

A map coloring program

Finally, consider a jewel of Prolog – the following MAP_COLOR program from Sterling and Shapiro [SS86, page 212] which generates a coloring of a map in such a way that no two neighbors have the same color. Below we call such a coloring *correct*. The map is represented as a list of regions and colors as a list of available colors. In turn, each region is determined by its name, color and the colors of its neighbors, so it is represented as a term region(name, color, neighbors), where neighbors is a list of colors of the neighboring regions.

```
color_map(Map, Colors) ← Map is correctly colored using Colors.

color_map([Region | Regions], Colors) ←
   color_region(Region, Colors),
   color_map(Regions, Colors).
color_map([], Colors).

color_region(Region, Colors) ← Region and its neighbors are correctly colored using Colors.

color_region(region(Name, Color, Neighbors), Colors) ←
   select(Color, Colors, Colors1),
   subset(Neighbors, Colors1).
```

augmented by the SELECT program.

augmented by the SUBSET program.

Denote by CM the program consisting of the two clauses defining the relation color_map, and by CR the program consisting of the clause defining the relation color_region. Clearly, CM extends CR, and CR extends SELECT and SUBSET. Moreover, color_map $\sqsupset$ color_region $\sqsupset$ select, subset in the program MAP_COLOR = CM $\cup$ CR $\cup$ SELECT $\cup$ SUBSET.

First we deal with the program CR $\cup$ SELECT $\cup$ SUBSET. To this end Theorems 5.6 and 5.7 will be of help. Recall that SELECT is recurrent w.r.t. $|\texttt{select(x, xs, ys)}| = |\texttt{xs}|$, and that SUBSET is recurrent w.r.t. $|\texttt{subset(xs, ys)}| = |\texttt{xs}| + |\texttt{ys}|$ and $|\texttt{member(x, xs)}| = |\texttt{xs}|$. Observe the following:

- the program CR is trivially semi-recurrent w.r.t. any level mapping.
- The Herbrand interpretation $I_S = \{\texttt{select(x,xs,ys)} \mid |\texttt{xs}| \geq |\texttt{ys}|\}$ is a model of SELECT, as the following proof outlines show:

$$\begin{array}{l} \{1 + |\texttt{xs}| \geq |\texttt{xs}|\} \\ \texttt{select(x, [x|xs], xs).} \end{array}$$

$$\begin{array}{lll} \{1 + |\texttt{xs}| \geq 1 + |\texttt{ys}|\} & & \\ \texttt{select(x, [y|xs], [y|ys])} & \leftarrow & \\ & & \texttt{select(x,xs,ys).} \\ & & \quad \{|\texttt{xs}| \geq |\texttt{ys}|\} \end{array}$$

  Consequently, by Lemma 3.4 and Theorem 5.6 SELECT $\cup$ SUBSET is semi-acceptable w.r.t. $|\ |$ and $I_S \cup [\texttt{subset(Xs, Ys)}] \cup [\texttt{member(X, Xs)}]$.

- The programs CR and SELECT $\cup$ SUBSET satisfy assumption 3 of Theorem 5.7 by putting:

$$I_{CR} = I_S \cup [\texttt{color\_region(R, Cs)}] \cup [\texttt{subset(Xs, Ys)}] \cup [\texttt{member(X, Xs)}]$$

  and extending $|\ |$ as follows:

$$\begin{array}{rcl} |\texttt{color\_region(region(n, c, ns), cs)}| & = & |\texttt{ns}| + |\texttt{cs}|, \\ |\texttt{color\_region(x, cs)}| & = & 0 \text{ if } \texttt{x} \neq \texttt{region(n, c, ns)}. \end{array}$$

  The associated weak proof outline follows:

$$\begin{array}{lll} \texttt{color\_region(region(n, c, ns), cs)} & \leftarrow & \{|\texttt{ns}| + |\texttt{cs}|\} \\ & \texttt{select(c, cs, c1s),} & \{|\texttt{cs}|\} \\ & \quad \{|\texttt{cs}| \geq |\texttt{c1s}|\} & \\ & \texttt{subset(ns, c1s).} & \{|\texttt{ns}| + |\texttt{c1s}|\} \end{array}$$

  Therefore, by Theorem 5.7, the program CR $\cup$ SELECT $\cup$ SUBSET is semi-acceptable w.r.t. $|\ |$ and I_{CR}.

Now we can deal with the program MAP_COLOR. For this purpose Theorem 5.8 will be of use. Observe the following:

- The program CM is trivially recurrent w.r.t. $|\texttt{color\_map(rs, cs)}| = |\texttt{rs}|$.
- Define a function $||\ ||$ from lists of regions to natural numbers by induction as follows:

$$\begin{array}{rcl} ||[\texttt{region(n, c, ns)|rs}]|| & = & |\texttt{ns}| + ||\texttt{rs}||, \\ ||[\texttt{x|rs}]|| & = & ||\texttt{rs}|| \text{ if } \texttt{x} \neq \texttt{region(n, c, ns)}, \\ ||\texttt{x}|| & = & 0, \text{ otherwise.} \end{array}$$

  The programs CM and CR $\cup$ SELECT $\cup$ SUBSET satisfy assumption 3 of Theorem 5.8 by putting:

$$I_{CM} = I_{CR} \cup [\texttt{color\_map(Rs, Cs)}],$$

$$||\texttt{color\_map(rs, cs)}||_{CM} = ||\texttt{rs}|| + |\texttt{cs}|.$$

Two weak proof outlines covering all ground instances of the recursive clause of color_map follow. We assume that $\mathtt{x} \neq \mathtt{region(n, c, ns)}$.

$$\begin{array}{lll} \mathtt{color\_map([region(n,c,ns)|rs],cs)} & \leftarrow & \{|\mathtt{ns}| + ||\mathtt{rs}|| + |\mathtt{cs}|\} \\ & \mathtt{color\_region(region(n,c,ns),cs)}, & \{|\mathtt{ns}| + |\mathtt{cs}|\} \\ & \mathtt{color\_map(rs,cs)}. & \{||\mathtt{rs}|| + |\mathtt{cs}|\} \end{array}$$

$$\begin{array}{lll} \mathtt{color\_map([x|rs],cs)} & \leftarrow & \{||\mathtt{rs}|| + |\mathtt{cs}|\} \\ & \mathtt{color\_region(x,cs)}, & \\ & \mathtt{color\_map(rs,cs)}. & \{||\mathtt{rs}|| + |\mathtt{cs}|\} \end{array}$$

Consequently, by Theorem 5.8 we conclude that the program MAP_COLOR = CM ∪ CR ∪ SELECT ∪ SUBSET is semi-acceptable. Moreover, we obtain that a goal $\leftarrow \mathtt{color\_map(rs,cs)}$ is bounded if cs is a list and rs is a list of regions $[\mathtt{region}(\mathtt{n}_1, \mathtt{c}_1, \mathtt{ns}_1), \ldots, \mathtt{region}(\mathtt{n}_k, \mathtt{c}_k, \mathtt{ns}_k)]$, where each $\mathtt{ns}_i$ ($\mathtt{i} \in [1,k]$) is a list. Thus, MAP_COLOR terminates for the desired class of goals.

Acknowledgements

We thank Antonio Brogi, Augusto Ciuffoletti and Paolo Mancarella for useful discussions on the final version and Andrea Schaerf for helpful comments.

References

[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.

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

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

[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.

[Bez93] M.A. Bezem. Strong termination of logic programs. *Journal of Logic Programming*, 15(1 & 2):79–98, 1993.

[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.

[Kön27] D. König. Über eine Schlußweise aus dem Endlichen ins Unendliche. *Acta Litt. Ac. Sci.*, 3:121–130, 1927.

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

[Red86] U.S. Reddy. On the relationship between logic and functional languages. In D. DeGroot and G. Lindstrom, editors, *Functional and Logic Programming*, pages 3–36. Prentice-Hall, 1986.

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

[SVB92] D. De Schreye, K. Verschaetse, and M. Bruynooghe. A framework for analyzing the termination of definite logic programs with respect to call patterns. In *Proceedings of the International Conference on Fifth Generation Computer Systems 1992*, pages 481–488. Institute for New Generation Computer Technology, 1992.

[TS84] H. Tamaki and T. Sato. Unfold/Fold Transformations of Logic Programs. In Sten-Åke Tärnlund, editor, *Proc. Second International Conference on Logic Programming*, pages 127–139, 1984.

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