Proving Termination of Logic Programs by Exploiting Term Properties *

A. Bossi, N. Cocco, M. Fabris
Dip. di Matematica Pura e Applicata, Università di Padova
via Belzoni, 7 - 35131 - PADOVA - ITALY
e.mail: mat010@ipdunivx.BITNET

Abstract. In this paper *semi-linear norms*, a class of functions to weight the terms occurring in a program, are defined and studied. All the functions in this class have the nice property of allowing a syntactical characterization of *rigid terms*, that is terms whose weight does not change under substitution. Based on these norms, a general proof method for termination of pure Prolog programs can be adapted to deal with a large class of programs in a simple way. The simplified method requires *pre/post specifications well-behaved with respect to substitutions*, quite a general case in practice, and *ordering functions not increasing with respect to substitutions*, which can be based on semi-linear norms, to be associated to program predicates. A few examples of this simplified proof method are given.

Keywords: Prolog programs, termination, norm on terms, pre/post specifications

1. Introduction

In logic programming the problem of determining if a computation terminates is even more relevant than in imperative programming. Prolog interpreters, due to their sequential, depth-first strategy, are basically unfair: a not terminating computation may cause to miss solutions which are logically derivable. Hence, in developing logic programs, we cannot avoid caring about their termination properties. Some applications, such as deductive Data-Bases, strongly require not only that one solution or a failure is reached in finite time (existential termination [Vas86]) but that all solutions are computed in finite time (universal termination [Vas86]), which means that the computation tree associated to the query must be finite. With negated atoms and negation as failure the requirement of having finite tree is also important. Some intuitive, empirical reasoning, generally guides programmers in checking these properties. Since non terminating computations can arise because of recursive predicate definitions or recursive data structures, the attention focuses on them while the programmer mentally simulates the interpreter's search for solutions. For example let us consider the two naive programs

* This work has been partially supported by "Progetto Finalizzato Sistemi Informatici e Calcolo Parallelo" of CNR under Grant n. 89.00026.69

$\mathcal{P}_1$: 1: p([], []).
2: p([X], [X]).
3: p([X,Y|Xs], Zs) :- s_m([X,Y|Xs], Us, Vs), p(Us, Ts), p(Vs, Ws), s_m(Zs, Ts, Ws).
4: s_m([], [], []).
5: s_m([A|Cs], [A|Ds], Es) :- s_m(Cs, Ds, Es).
6: s_m([A|Cs], Ds, [A|Es]) :- s_m(Cs, Ds, Es).

$\mathcal{P}_2$: 1: p([], []).
2: p([X], [X]).
3: p([X,Y|Xs], Zs) :- s_m([X,Y|Xs], Us, Vs), p(Us, Ts), p(Vs, Ws), s_m(Zs, Ts, Ws).
4: s_m([], [], []).
5: s_m([A], [], [A]).
6: s_m([A], [A], []).
7: s_m([A,B|Cs], [A|Ds], [B|Es]) :- s_m(Cs, Ds, Es).
8: s_m([A,B|Cs], [B|Ds], [A|Es]) :- s_m(Cs, Ds, Es).

The predicate p(L1, L2) has the same definition in the two programs and it holds when the lists L1 and L2 are permutations one of each other; the predicate s_m(L1, L2, L3) is defined differently, but in both programs it holds when the list L1 contains all the elements in the other two lists and only those, that is s_m(L1, L2, L3) stands either for split the list L1 into the lists L2 and L3 or for merge the lists L2 and L3 into list L1, depending on the input/output functionality chosen. However there is a difference: in $\mathcal{P}_1$ s_m can produce an unfair splitting, s_m(Xs, Xs, []) holds, while this does not happen in $\mathcal{P}_2$ if the list Xs has more than one element. The two programs are very inefficient because of their nondeterminism, but this is not the point here. It is rather immediate to get convinced that the computation of s_m(L1, L2, L3) terminates with both the definitions if either L1, or at least two of the parameters, are ground, i.e. they do not contain variables. We may infer this by observing that in either cases the lists in input are recursively inspected until the base list is reached. This is equivalent to say that the predicate is recursively applied to smaller terms. But if we consider the predicate p(L1, L2), our analysis becomes more complicated and it is easy to miss that, even when the first parameter is ground, the computation does not terminate in $\mathcal{P}_1$ while it terminates in $\mathcal{P}_2$.
On the other hand an infinite computation can arise also if no structured term is present as in the following program where terms are only variables or constants

$\mathcal{P}_3$: 1: c(X, Y) :- base(X, Y).
2: c(X, Y) :- base(X, Z), c(Z, Y).
3: base(a, b).
4: base(a, c).
5: base(a, d).
6: base(b, c).
7: base(c, d).
8: base(d, b).

where c(X, Y) is the transitive closure of the relation defined by base(Z, W). In this program an infinite computation for the goal :- c(X, Y). and its instances can arise because of a cycle in the flat data domain given by the base relation.

Since we know that the halting-problem is not decidable, we can only expect sufficient criteria for termination. Moreover given the multiple input/output functionalities associated to a logic program, it would be rather difficult, and in most cases not very meaningful, to prove the termination for any goal and for any functionality. An infinite computation most often comes from the possibility of infinite expansions of a term, hence knowing which terms in the goal have a fixed structure is extremely relevant for guaranteeing termination. This not necessarily means that the goal's parameters have to be ground or to become ground during the computation. Let us consider again our examples: s_m([X1, X2, X3, X4], L2, L3) terminates both in $\mathcal{P}_1$ and $\mathcal{P}_2$, so does p([X1, X2, X3], L) in $\mathcal{P}_2$. On the other hand c(X, Y) would terminate in $\mathcal{P}_3$ with a finite set of ground substitutions if clause 8 were eliminated.

Termination of logic programs have been studied with various approaches. Three main directions of research can be singled out, even if there are many connections between them. One deals with characterizing classes of programs and/or classes of goals which have some termination property [Vas86, Apt89, Bez89]. Another one is concerned with the automatic and efficient generation of inequalities among size of terms, whose satisfaction is sufficient for universal termination [Ull88, Wan90, Plü90a, Plü90b]. A third approach, which is also our approach, deals with techniques for verifying termination properties and guiding program development [Fra85, Bau88, Wan89, Apt90, Dev90].

In this work we basically try to formalize the intuitive reasoning on programs which programmers usually do and come to a rather simple verification method, similar to the one used in procedural programming. We are interested in proving universal termination of a class of goals, which means that for every goal G in the class, all computations for G terminate (or equivalently G has a finite computation tree). Since we consider universal termination, the order of clauses, which corresponds to the order of solutions in the search tree, is not relevant, while the order of atoms in the bodies of clauses can affect termination. Obviously ignoring the order of clauses allows us to simplify proofs. We assume the Prolog computation rule, while its search strategy is not relevant. We adapt to logic programming and integrate all what is already known for recursive programming, namely our approach is axiomatic in style [Flo67, Hoa69, Dij76, Cla77, Gri81, Hog84, Dra88, Bos89], it is based on appropriately specifying the program and inductively proving such specifications, and it consists in finding a well-founded ordering on predicate invocations, that is defining a function on the predicate arguments with value in a well-founded set and prove that it decreases at each recursive call. To this purpose we define norms on terms and through them characterize a class of terms which have the property of having fixed structure (rigidity) and not been expandable during the computation in a way which could affect termination. Such class obviously includes ground terms, but it is much more general and it allows us to deal with termination of not ground goals. We axiomatize the properties of such norms and terms in a theory which can be used to prove termination. Obviously our proving technique requires a deep understanding of the program we analyse and it is not automatic. The axiomatic style is not intended to compete with efficient algorithms for analysing already existing programs, but to have a technique for reasoning and understanding programs while developing them. We are convinced that one gets a deep insight from such programs verification, which can guide program development and transformation.

The paper is organized as follows. In the next section we introduce the basic definitions and properties of norms and rigid terms and illustrate them by means of examples. Such definition and properties are then collected in a set of deduction rules. In section 3 our proof technique is described through examples. It consists in determining all the maximal strongly connected subgraphs in the graph associated to the program and a class of goals, defining an ordering function on atoms with a well-founded set as codomain and proving that it decreases on all the elementary circuits. Such function is based on appropriate norms on terms. In order to prove that it decreases, we make use of a pre/post specification of the program and the class of goals which usually describes rigidity properties and relations among terms. Both the specification and the ordering function are "well-behaved" with respect to substitutions. This allows us to give a simplified method for proving termination. Section 4 contains some conclusive remarks.

2. Norms on terms

A natural approach to the problem of defining a well-founded ordering on predicate invocations is: first assign a weight to the terms to which the predicates apply, and then define an ordering function based on it. Hence we focus first on how to weight the terms. Obviously, the success of our proofs strongly depends on the correct choice of this weight. This is not new and there are many examples of different ways of giving a weight to terms. For instance, in [Ull88] a concept of *size* of list arguments is used and in [Plü90a, Plü90b] a *norm*, that is a function weighting terms is introduced. Moreover, a class of norms which allow him to develop an interesting technique for the automatic generation of termination proofs is given. Our work has some analogies with [Plü90a]. We do not concentrate on automatization and efficiency but study a wider class of norms which still have interesting properties. These properties allow us to give a proof method, which is rather simple and largely syntactically driven. We think it is possible to automatize the more tedious parts. In this section we define and study this class of weighting functions.

2.1 Basic definitions and properties

In the following we assume that T is a set of terms built up on a set V of variables and a set F of n-ary ($n \geq 0$) function symbols which we shall call also *constructors*.

We consider two kinds of operations on terms. The usual substitution of variables, which is a set of pairs variable/term, and the *replacement of a term for a subterm*, which is a pair {term $\rightarrow$ subterm}. By referring to a subterm of t we mean to refer to a particular occurrence of the subterm in t, therefore any substitution can be expressed by the set of replacements of each occurrence of the variables. $t\{s \rightarrow r\}$ denotes the result of the replacement in t of the term s for the subterm r of t. For simplicity' sake we also use $t'\{s \rightarrow r\}$ to denote the result of the replacement in t' of the term s for the subterm r whenever either t is a subterm of t', or r is a subterm of t' which in turn is a subterm of t. We call *principal subterms* those subterms to which the outermost constructor applies.

We adopt the following general definition given by Plümer in [Plü90a].

2.1.1 Definition

Let T be a set of terms. *A norm on T* is a function, $|\ldots|_f : T \rightarrow N$, mapping T on natural numbers. *The weight of t with respect to a norm* $|\ldots|_f$ is the value $|t|_f$.

Examples.

Suppose F comprises at least the constructors for lists and for binary trees.

- We can define the norm $|\ldots|_{size}$ as follows:
  1) $|t|_{size} = 0$, if t is a variable or it is not a list;
  2) $|t|_{size} = 0$, if t is [];
  3) $|t|_{size} = 1 + |tail|_{size}$, if t is [head| tail].
  On a ground list this norm measures the length of the list. On a not ground list l it gives the minimum length of ground instantiations of l.
- We can define the norm $|\ldots|_{depth}$ as follows:
  1) $|t|_{depth} = 0$, if t is a variable or it is not a tree;
  2) $|t|_{depth} = 0$, if t is *void* ;
  3) $|t|_{depth} = 1 + \max(|lft|_{depth} + |rgt|_{depth})$, if t is *tree*(r, lft, rgt).
  This norm measures the depth of the tree.
- We can define the norm $|\ldots|_{rpath}$ as follows:
  1) $|t|_{rpath} = 0$, if t is a variable or it is not a tree;
  2) $|t|_{rpath} = 0$, if t is *void* ;
  3) $|t|_{rpath} = 1 + |rgt|_{rpath}$, if t is *tree*(r, lft, rgt).
  This norm measures the length of the rightmost path of the tree.
- We can define the norm $|\ldots|_{odd}$ as follows:
  1) $|t|_{odd} = 0$, if t is a variable or it is not a list;
  2) $|t|_{odd} = 0$, if t is [];
  3) $|t|_{odd} = 1$, if t is [h| X] where X is a variable or the empty list [];
  4) $|t|_{odd} = 1 + |tail|_{odd}$, if t is $[h_1 |[h_2| tail]]$.
  This norm, applied to a list, counts the elements in odd positions.

2.1.2 Definition

Let $|\ldots|_f$ be a norm on a set of terms T. *A term* $t \in T$ *is rigid with respect to* $|\ldots|_f$ if its weight is invariant under substitutions:

$\forall\sigma.\ |t\sigma|_f = |t|_f$.

Examples.

The list [a, X] is rigid with respect to the norm $|\ldots|_{size}$. In fact for any σ, $|[a, X\sigma]|_{size} = 2$.

The list [a| X] is not rigid with respect to the same norm. For instance let us consider $\sigma = \{X/[b,c]\}$. We have $|[a| X\sigma]|_{size} = |[a| [b,c]]|_{size} = 3$, while $|[a| X]|_{size} = 1 + |X|_{size} = 1$.

The concept of rigid term turns out to be extremely important in simplifying termination proofs. If a rigid term is encountered during the computation, its weight will not be modified by further substitutions and the possibility of ignoring substitutions makes proofs much simpler. Clearly a ground term is rigid whichever the norm is: *rigidity is an extension of groundness*. Rigidity depends on the choice of the norm, and finding out the norm which makes some specific terms rigid is one of the main points in a termination proof.

Let $|\ldots|_f$ be a norm on a set T and $t \in T$ a term which is not rigid. Then there must occur some variables in t whose substitution may affect the weight of t. We want to identify each of their occurrences.

2.1.3 Definition

Let $|\ldots|_f$ be a norm on a set of terms T and $t \in T$. *The i-th occurrence, $X_{(i)}$, of a variable X in the term t is relevant with respect to $|\ldots|_f$* whenever there exists a replacement $\{s \rightarrow X_{(i)}\}$ of the term s for the i-th occurrence of X in t such that $|t\{s \rightarrow X_{(i)}\}|_f \neq |t|_f$. We will call $VREL_f(t)$ the set of all the relevant occurrences of variables in t.

Example.

Let us consider the term *tree*(a, *tree*(b, *void*, X), *tree*(c, Y, X)) and the norm $|\ldots|_{rpath}$. Then, $VREL_{rpath}(t) = \{X_{(2)}\}$.

The previous definition can be extended to terms.

2.1.4 Definition

Let $|\ldots|_f$ be a norm on a set of terms T and $t \in T$. *A subterm s of t is relevant with respect to $|\ldots|_f$* whenever there exist a replacement $\{s' \rightarrow s\}$ of the term s' for the term s in t such that $|t\{s' \rightarrow s\}|_f \neq |t|_f$. We will call $SREL_f(t)$ the set of all the relevant subterms of t.

Note that a term t is always a relevant subterm of itself unless the norm is constant.

Example.

The relevant subterms of t =*tree*(a, *tree*(b, *void*, X), *tree*(c, Y, X)) with respect to the norm $|\ldots|_{rpath}$ are: t, *tree*(c, Y, $X_{(2)}$), $X_{(2)}$.

Clearly, since we may always think of the occurrence of a variable in a term t as a subterm, if we denote by Var(t) the set of all the occurrences of variables in the term t: $VREL_f(t) = SREL_f(t) \cap Var(t)$.

In the sequel, to avoid unnecessary subscripts, we shall omit the reference to the specific norm whenever this does not cause ambiguity.

The following definition introduces an interesting class of norms which allow a syntactic characterization of rigid terms.

2.1.5 Definition

Let T be a set of terms. *A norm |../ on T is semi-linear* if it can be recursively defined, for any term t in T, by using the following schema:

if t is a variable then $|t| = 0$;

if $t = f(t_1, \ldots, t_n)$ then $|t| = c_0 + |t_{i1}| + \ldots + |t_{im}|$, where $c_0 \geq 0$, and $\{1, \ldots, n\} \supseteq \{i_1, \ldots, i_m\}$.

Note that, for any term $t=f(t_1, \ldots, t_n)$, the definition of a semi-linear norm selects the subterms $t_{i1}, \ldots, t_{im}$ among the principal ones depending only on the outermost constructors. We call these subterms *selected subterms.*

Examples.

Let us consider the norms given in the previous examples. The norm $|\ldots|_{size}$ is semi-linear: it yields 0 when applied to variables and follows the schema in the other cases. The selected subterm is the tail of the list. Also $|\ldots|_{rpath}$ is semi-linear. On the contrary, $|\ldots|_{depth}$ does not match the definition since max(a, b) cannot be expressed by a linear combination of a and b. Also $|\ldots|_{odd}$ is not semi-linear since the definition does not depends on principal subterms only.

We introduced the class of semi-linear norms as a generalization of the class of linear norms as defined in [Plü90a]. Indeed we can prove that any linear norm is semi-linear. Let us recall the definition of linear norm.

2.1.6 Definition [Plü90a]

A norm |...| on T is linear if for any $t \in T$ and any substitution of variables σ such that $t\sigma$ is ground: $|t\sigma| = |t| + \Sigma_{V \in Var(t)} |V\sigma|$.

The following proposition provides a conceptually simpler, alternative definition of linearity.

2.1.7 Proposition: A norm |...| on T is linear if and only if it can be recursively defined, for any term in T, by means of the following schema:

if t is a variable then $|t| = 0$;

if t is $f(t_1, \ldots, t_n)$ then $|t| = c_0 + |t_1| + \ldots + |t_n|$, where $c_0 \geq 0$.

Proof.

only-if part) Let |...| be a linear norm. Consider $V \in Var$ and a substitution σ such that $V\sigma$ is ground. Since |...| is linear, $|V\sigma| = |V| + |V\sigma|$, hence $|V|=0$. Consider now a structured term: $t=f(t_1, \ldots, t_n)$, then there exists another structured term $t'=f(X_1, \ldots, X_n)$, formed by applying the constructor f to the distinct variables $X_1, \ldots, X_n$, and a substitution $\rho=\{X_1/t_1, \ldots, X_n/t_n\}$ such that $t'\rho=t$. Consider another substitution σ such that $t'\rho\sigma$ is ground. Since |...| is linear, by considering the substitution $\rho\sigma$, we have:

$$|t'\rho\sigma| = |t'| + \Sigma_{X \in Var(t')} |X\rho\sigma| = |t'| + \Sigma_{X \in Var(t')} (|X\rho| + \Sigma_{V \in Var(X\rho)} |V\sigma|)$$
$$= |t'| + \Sigma_{X \in Var(t')} |X\rho| + \Sigma_{X \in Var(t')} \Sigma_{V \in Var(X\rho)} |V\sigma|$$
$$= |t'| + \Sigma_{X \in Var(t')} |X\rho| + \Sigma_{W \in Var(t'\rho)} |W\sigma|$$

On the other hand, by considering only the substitution σ applied to $t'\rho = t$, we have:

$|t'\rho\sigma| = |t'\rho| + \sum_{W \in Var(t'\rho)} |W\sigma|$

Hence $|t| = |t'\rho| = |t'| + \sum_{X \in Var(t')} |X\rho| = |t'| + |t_1| + \ldots + |t_n|$, where $|t'|$ depends only on the constructor f.

if part) Let us prove now, by induction on the structure of the terms, that a norm $|\ldots|$ defined by means of the given schema is linear.

- If t is a variable then $|t| = 0$ and the condition of linearity is satisfied.
- If t is a constructor of arity 0 then it does not contain variables and the condition of linearity is trivially satisfied.
- If $t=f(t_1, \ldots, t_n)$, for any substitution σ it is $f(t_1, \ldots, t_n)\sigma = f(t_1\sigma, \ldots, t_n\sigma)$, hence, $|f(t_1, \ldots, t_n)\sigma| = c_0 + |t_1\sigma| + \ldots + |t_n\sigma|$. By inductive hypothesis $|\ldots|$ behaves linearly on subterms, then $|t_j\sigma| = |t_j| + \sum_{V \in Var(t_j)} |V\sigma|$, $1 \leq j \leq n$. Hence $|t\sigma| = c_0 + |t_1\sigma| + \ldots + |t_n\sigma| = c_0 + |t_1| + \ldots + |t_n| + \sum_{V \in Var(t1)} |V\sigma|) + \ldots + \sum_{V \in Var(tn)} |V\sigma|) = |t| + \sum_{V \in Var(t1)} |V\sigma|) + \ldots + \sum_{V \in Var(tn)} |V\sigma|)$ and the claim follows, since $Var(t) = \cup_j Var(t_j)$.

The major advantage in semi-linear norms is that they characterize relevant subterms in a pure syntactic way as shown by the next proposition.

2.1.8 Proposition: For any given definition of a semi-linear norm on T, for any term t in T, the following properties hold

i) if the norm is constant, then $SREL(t) = \varnothing$;

ii) if the norm is not constant,

$SREL(t) = \{t\}$, if t is a variable

$SREL(t) = \{t\} \cup (\cup_{j=1,\ldots,m} SREL(t_{ij}))$, if $t=f(t_1, \ldots, t_n)$ and t_{ij}, $1 \leq j \leq m$, are the selected subterms of t.

Proof.

(i) It is trivial.

(ii) If t is a variable, then the proof comes trivially from the assumption that the norm is not constant.

Let us consider $t=f(t_1, \ldots, t_n)$ where t_{ij}, $1 \leq j \leq m$, are the selected subterms of t.

1) $SREL(t) \supseteq \{t\} \cup (\cup_{j=1,\ldots,m} SREL(t_{ij}))$.

From the assumption that the norm is not constant $t \in SREL(t)$.

Let us consider $s \in SREL(t_{ik})$, $k \in \{1, \ldots, m\}$. Then there exists a replacement $\{s' \rightarrow s\}$ in t_{ik}, such that

$|t_{ik}\{s' \rightarrow s\}| \neq |t_{ik}|$.

Since the norm is semi-linear

$|t| = c_0 + |t_{i1}| + \ldots + |t_{ik}| + \ldots + |t_{im}|$

and, since the replacement influences only t_{ik},

$|t\{s' \rightarrow s\}| = c_0 + |t_{i1}| + \ldots + |t_{ik}\{s' \rightarrow s\}| + \ldots + |t_{im}|$.

Hence $|t| \neq |t\{s' \rightarrow s\}|$ which means that $s \in SREL(t)$.

2) $\{t\} \cup (\cup_{j=1,\ldots,m} SREL(t_{ij})) \supseteq SREL(t)$.

The proof proceeds by contradiction. Suppose $s \in SREL(t)$, $s \neq t$ and, ad absurdum, $\forall j=1,\ldots,m$ $s \notin SREL(t_{ij})$. Since $s \in SREL(t)$, there exists a replacement $\{s' \rightarrow s\}$ in t such that $|t| \neq |t\{s' \rightarrow s\}|$.

$t=f(t_1, \ldots, t_n)$ and $s \neq t$, then there exists a principal subterm t_h, $h \in \{1, \ldots, n\}$, such that $t\{s' \rightarrow s\} = f(t_1, \ldots, t_h\{s' \rightarrow s\}, \ldots, t_n)$. t_h must be a selected subterm of t, otherwise $|t\{s' \rightarrow s\}| = |f(t_1, \ldots, t_h\{s' \rightarrow s\}, \ldots, t_n)| = c_0 + |t_{i1}| + \ldots + |t_{im}| = |t|$. Therefore there exists $k \in \{1, \ldots, m\}$ such that $h=ik$. Hence $|t\{s' \rightarrow s\}| = c_0 +$

$|t_{i1}| + \ldots + |t_{ik}\{s' \rightarrow s\}| + \ldots + |t_{im}| \neq c_0 + |t_{i1}| + \ldots + |t_{ik}| + \ldots + |t_{im}| = |t|$, and then $|t_{ik}\{s' \rightarrow s\}| \neq |t_{ik}|$ which contradicts the hypothesis $s \notin SREL(t_{ik})$.

2.1.9 Corollary. For any given definition of a not constant semi-linear norm on T, for any term t in T, SREL(t) is the minimal set which contains t and is closed with respect to the operation of selecting subterms.

Proof.

Immediate by induction and the previous proposition 2.1.8.

2.1.10 Corollary. For any given definition of a semi-linear norm, the property of being relevant is transitive, that is any relevant subterm of a relevant subterm of a term t is a relevant subterm of t.

Proof.

By induction on the complexity of t. Let $s \in SREL(t)$ and $w \in SREL(s)$. The only interesting case is $s \neq t$, then, by proposition 2.1.8, there exists a selected subterm t_{ik} of t such that $s \in SREL(t_{ik})$. By inductive hypothesis, $w \in SREL(t_{ik})$ and, by the same proposition 2.1.8, $w \in SREL(t)$.

2.1.11 Proposition. VR-linearity condition : for any term t in T and any substitution of variables σ:

$$|t\sigma|_f = |t|_f + \Sigma_{V \in VREL(t)} |V\sigma|_f$$

holds for any semi-linear norm $|\ldots|_f$ on T.

Proof.

Let $|\ldots|_f$ be a semi-linear norm. We proceed by induction on the structure of the terms.

i) t is a variable X.

Either $VREL(t)=\emptyset$ and then $|X\sigma|=|X|=0$ or $VREL(t)=\{X\}$ and then $|X\sigma|=|X|+|X\sigma|$ since, by definition, $|X|=0$.

ii) $t=f(t_1, \ldots, t_n)$.

For any substitution of variables σ: $|t\sigma|_f=|f(t_1\sigma, \ldots, t_n\sigma)|_f=c_0+|t_{i1}\sigma|_f+\ldots+|t_{im}\sigma|_f$. By inductive hypothesis: $|t_{ij}\sigma|_f = |t_{ij}|_f + \Sigma_{V \in VREL(tij)} |V\sigma|_f$, $1 \leq j \leq m$. Then,

$$\begin{aligned} |t\sigma|_f &= |f(t_1\sigma, \ldots, t_n\sigma)|_f \\ &= c_0 + (|t_{i1}|_f + \Sigma_{V \in VREL(ti1)} |V\sigma|_f) + \ldots + (|t_{im}|_f + \Sigma_{V \in VREL(tim)} |V\sigma|_f) \\ &= |t|_f + \Sigma_{V \in VREL(t)} |V\sigma|_f \end{aligned}$$

since $VREL(t) = \cup_j VREL(t_{ij})$ by proposition 2.1.8.

For non-constant norms VR-linearity condition characterizes semi-linear norms.

2.1.12 Proposition: Any non-constant norm which satisfies VR-linearity condition is semi-linear.

Proof.

Let $|\ldots|_f$ be a not constant norm which satisfies the VR-linearity condition. We distinguish two cases.

i) t is the variable X. We know that $|X\sigma|_f = |X|_f + \Sigma_{V \in VREL(t)} |V\sigma|_f$. Moreover X must be relevant, since we are considering non-constant norms. Hence $|X\sigma|_f = |X|_f + |X\sigma|_f$, which implies $|X|_f=0$.

ii) t is $t=f(t_1, \ldots, t_n)$. Let $X_1, \ldots, X_n$ be new variables and $\sigma=\{X_1/t_1, \ldots, X_n/t_n\}$. Then $t=f(X_1, \ldots, X_n)\sigma$. Let $\{X_{i1}, \ldots, X_{im}\}=VREL_f(f(X_1, \ldots, X_n))$, since $|\ldots|_f$ is VR-linear,

$|t|_f = |f(X_1, \ldots, X_n)\sigma|_f = |f(X_1, \ldots, X_n)|_f + \Sigma_{V \in VREL(f(X1, \ldots, Xn))} |V\sigma|_f = |f(X_1, \ldots, X_n)|_f + |X_{i1}\sigma|_f + \ldots + |X_{im}\sigma|_f = |f(X_1, \ldots, X_n)|_f + |t_{i1}|_f + \ldots + |t_{im}|_f$.

2.1.13 Proposition: Let $|\ldots|_f$ be a semi-linear norm on T. A term t in T is rigid with respect to $|\ldots|_f$ if and only if $VREL_f(t)=\emptyset$.

Proof.

The if part is immediate from the definition of rigid term.

The only if part easily follows from the proposition 2.1.11. In fact, since $|\ldots|_f$ is VR-linear, by proposition 2.1.11, $|t\sigma|_f = |t|_f + \Sigma_{V \in VREL(t)} |V\sigma|_f$, but t is rigid and then $\Sigma_{V \in VREL(t)} |V\sigma|_f = 0$ for any substitution of variables σ. Then either $VREL(t)=\emptyset$ or $VREL(t)\neq\emptyset$ and there exists a variable $V \in VREL(t)$ such that $|V\sigma|=0$ for any substitution σ. But this second case is contradictory since $|V\sigma|=0$ for any substitution σ implies $V \notin VREL(t)$.

It is worth noting that, given a semi-linear norm, the check $VREL(t)=\emptyset$ can be done syntactically, in fact the relevant occurrences of variables in t are in SREL(t) which is the minimal set containing t and closed with respect to the operation of selecting subterms.

Note also that VR-linearity is essential for the validity of the previous theorem. This may be not so evident but consider, as an example, the following norm $|\ldots|_{bal}$

$|t|_{bal} = 0$, if t is *void* or it is not a tree;

$|t|_{bal} = 0$, if t is *tree* (a, lft, rgt) and (lft ≡ rgt);

$|t|_{bal} = 1$, if t is *tree* (a, lft, rgt) and not(lft ≡ rgt).

where ≡ means syntactical equality.

The norm $|\ldots|_{bal}$ is not VR-linear nor semi-linear. The term *tree*(a, X, X) is rigid with respect to the given norm since for any substitution of the variable, the two subtrees remain equal. Nevertheless VREL(*tree*(a, X, X))=$\{X_{(1)}, X_{(2)}\}$ since by substituting one single occurrence of the variable X we may obtain different subterms. Therefore, semi-linearity guarantees a behaviour which is in some sense the expected one.

Another important property of semi-linear norms is stated by the following proposition.

2.1.14 Proposition: For any semi-linear norm, the relevant subterms of a rigid term are rigid.

Proof.

A variable can be rigid only if the norm is constant, but in this case there are no relevant subterms. Hence, let $t=f(t_1, \ldots, t_n)$ be rigid and s be any relevant subterm of t. Suppose s is not rigid. By proposition 2.1.13, $VREL(s)\neq\emptyset$, and since $VREL(t)\supseteq VREL(s)$ by corollary 2.1.10, $VREL(t)\neq\emptyset$ which leads to contradiction, by proposition 2.1.13.

This last property is not satisfied by every norm. To see this consider again the norm $|\ldots|_{bal}$ and the rigid term *tree*(a, X, X). The subterm $X_{(1)}$ is relevant for *tree*(a, X, X) but it is not rigid.

The following proposition is a sort of inverse of the previous one.

2.1.15 Proposition: Let $s_1, \ldots, s_k$ be subterms of t such that $\cup_{j=1,\ldots,k} VREL(s_j) = VREL(t)$. Then, if all the terms $s_1, \ldots, s_k$ are rigid, t is rigid too.

Proof.

Since $s_1, \ldots, s_k$ are rigid terms, by proposition 2.1.13, $VREL(s_j)=\emptyset$, $1 \leq j \leq k$. Then $VREL(t)=\emptyset$ and, by the same proposition, t is rigid.

2.2 Rules

The definitions and propositions developed in the previous subsection can be collected in a set of inference rules $\mathfrak{R}$ characterizing the predicate "rigid" in case of semi-linear norms. These rules are used in the termination proofs. The first two rules correspond to the definition of rigid terms while the succeeding two are direct consequences of the definition. The rules 3.a and 3.b correspond to proposition 2.1.13 and are basic to establish rigidity. The rules 4 and 5 correspond to propositions 2.1.14 and 2.1.15, they allow one to infer rigidity of subterms from rigidity of terms and vice versa.

(definition)

$$1.a)\ \frac{rigid(t)}{\forall\sigma(|t\sigma|=|t|)} \qquad 1.b)\ \frac{\forall\sigma(|t\sigma|=|t|)}{rigid(t)}$$

(persistency)

$$2.a)\ \frac{rigid(t)}{\forall\sigma(rigid(t\sigma))} \qquad 2.b)\ \frac{\forall\sigma(rigid(t\sigma))}{rigid(t)}$$

(syntactic characterization)

$$3.a)\ \frac{VREL(t)=\emptyset}{rigid(t)} \qquad 3.b)\ \frac{rigid(t)}{VREL(t)=\emptyset}$$

(term decomposition)

$$4)\ \frac{rigid(t) \qquad s \in SREL(t)}{rigid(s)}$$

(term composition)

$$5)\ \frac{\cup_{j=1,\ldots n} VREL(s_j) \supseteq VREL(t) \qquad rigid(s_1) \wedge \ldots \wedge rigid(s_n)}{rigid(t)}$$

Rules 1.a and 2.a ensure that rigidity properties are not modified during the computation.

Rule 3.a is useful for deducing rigidity properties from the syntactic structure of terms. For example let us consider the predicate p([X], [X]) and the norm $|\ldots|_{size}$. Since $VREL_{size}$ ([X]) =∅, by applying rule 3.a we can deduce rigid([X]).

Rule 4 is useful for deducing rigidity properties of subterms of a rigid term. Let us consider the predicate p([X,Y|L], Zs) and a substitution σ. Since $L\sigma \in SREL_{size}([X,Y|L]\sigma)$, if we know that rigid([X,Y|L]σ) holds, by applying rule 4 we can deduce rigid(Lσ).

Rule 5 is useful for deducing rigidity properties of terms from rigidity properties of subterms. Let us consider the predicate p([X,Y|L], Zs) and a substitution σ. Since $VREL_{size}([X,Y|L]\sigma) = VREL_{size}(L\sigma)$, if we know that rigid(Lσ) holds, by applying rule 5 we can deduce rigid([X,Y|L]σ).

3. The verification method

In this section the method we propose for verifying the termination of a pure Prolog program is described by means of a few examples, a complete description can be found in [Fab90, Bos91].

In the following we assume that a logic program $\mathcal{P}$, that is a set of definite clauses, is written in a first-order language $\mathcal{L}_\mathcal{P}$. We denote by $\mathcal{Pred}_\mathcal{P}$ and $\mathcal{Term}_\mathcal{P}$, respectively, the sets of n-ary (n>0) predicate symbols and the set of terms of $\mathcal{L}_\mathcal{P}$. Let p be a predicate symbol of arity n. We denote by $p(x_1, \ldots, x_n)$ its general form and by $p(t_1, \ldots, t_n)$ the general instance of an atom with predicate symbol p. Note that both $x_1, \ldots, x_n$ and $t_1, \ldots, t_n$ are metavariables representing the n arguments of p. We use the metavariables t_i when we want to stress the fact that the arguments of p are terms in $\mathcal{Term}_\mathcal{P}$ and we are considering an instance of the general form. We use the abbreviation $\underline{y}$ for the tuple $y_1, \ldots, y_n$. When considering a goal G for a specific program $\mathcal{P}$, we always assume that G is written in the language $\mathcal{L}_\mathcal{P}$. A general goal $G=:- q_1(\underline{x}_1), \ldots, q_n(\underline{x}_n)$. is a finite sequence of atoms in general form.

3.1. Annotating the program

A termination proof generally requires a certain amount of information about terms such as rigidity properties and relations among terms in the same atom. We find very natural to annotate the program and the goal with such information by means of pre/post specification [Dra88, Bos89]. In the sequel we introduce these concepts.

3.1.1 Definition

Let $\mathcal{L}_\mathcal{P}$ be the language associated to a logic program $\mathcal{P}$, and $p \in \mathcal{Pred}_\mathcal{P}$. *A pre/post specification of the predicate* $p(x_1, \ldots, x_n)$ is an expression of the form

$\{Pre_p(x_1, \ldots, x_n)\}\ p(x_1, \ldots, x_n)\{Post_p(x_1, \ldots, x_n)\}$

where $Pre_p(x_1, \ldots, x_n)$ and $Post_p(x_1, \ldots, x_n)$ are formulas of a first order language $\mathcal{L}$ whose set of constants contains $\mathcal{Term}_\mathcal{P}$.

A predicate symbol p in $\mathbf{Pred_P}$ *is asserted* if a pre/post specification is associated to it.
An asserted program $\mathbf{P}$ is a program such that each p$\in\mathbf{Pred_P}$ is asserted.

Example.
Let us consider the program defining the permutations of a list we introduced in the first section

```
P2: 1:  p([], []).
    2:  p([X], [X]).
    3:  p([X,Y|Xs], Zs) :-  s_m([X,Y|Xs], Us, Vs), p(Us, Ts), p(Vs, Ws), s_m(Zs, Ts, Ws).
    4:  s_m([], [], []).
    5:  s_m([A], [], [A]).
    6:  s_m([A], [A], []).
    7:  s_m([A,B|Cs], [A|Ds], [B|Es]) :- s_m(Cs, Ds, Es).
    8:  s_m([A,B|Cs], [B|Ds], [A|Es]) :- s_m(Cs, Ds, Es).
```

If we consider the norm *size* defined in section 2, $\mathbf{P}_2$ can be asserted in this way:

$\{rigid(x_1)\}\ p(x_1, x_2)\ \{rigid(x_1, x_2)\}$

$\{rigid(x_1) \vee rigid(x_2, x_3)\}\ s\_m(x_1, x_2, x_3)\ \{rigid(x_1, x_2, x_3) \wedge (|x_1| = |x_2| + |x_3|) \wedge$
$((|x_1| > 1) \rightarrow ((|x_1| > |x_2|) \wedge (|x_1| > |x_3|)))\}$

where we omit the subscript *size* and adopt the abbreviation $rigid(x_1, \ldots, x_n)$ for $rigid(x_1) \wedge \ldots \wedge rigid(x_n)$.

Let $p(x_1, \ldots, x_n)$ be a n-ary predicate symbol, $\{Pre_p(x_1, \ldots, x_n)\}p(x_1, \ldots, x_n)\{Post_p(x_1, \ldots, x_n)\}$ be a pre/post specification for it, and let $P=p(t_1, \ldots, t_n)$ be an instance of $p(x_1, \ldots, x_n)$. We write either $Pre_p(t_1, \ldots, t_n)$ $(Post_p(t_1, \ldots, t_n))$ or Pre_P $(Post_P)$ to indicate the corresponding instance of $Pre_p(x_1, \ldots, x_n)$ $(Post_p(x_1, \ldots, x_n))$.

A pre/post specification is meant to relate some properties (the precondition) which hold before the computation of the corresponding predicate to other properties (the postcondition) which hold after its computation. Hence, we say that a predicate $p(\underline{x})$ is partially correct with respect to its specification when for all its instances $p(\underline{t})$ which satisfy the precondition, $Pre_p(\underline{t})$, and for all the computed answer substitutions, σ, of $p(\underline{t})$, the postcondition, $Post_p(\underline{t}\sigma)$, holds.

3.1.2 Definition

A theory $\mathbf{T}$ *can be associated to an asserted program* $\mathbf{P}$ if

1) it is written in the language $\mathbf{L}$ of the pre/post specification;
2) it describes functions and predicates of $\mathbf{L}$;
3) it contains an arithmetic theory and a theory of term equality.

3.1.3 Definition

Let $\mathbf{P}$ be an asserted program and $\mathbf{T}$ a theory associated to its specification. *The asserted program* $\mathbf{P}$ *is (partially) correct with respect to its specification in* $\mathbf{T}$ iff
for any p$\in\mathbf{Pred_P}$, for any tuple $\underline{t}$ and for any computed answer substitution, σ, of $p(\underline{t})$ in $\mathbf{P}$:

if $\mathbf{T} \vdash Pre_p(\underline{t})$ then $\mathbf{T} \vdash Post_p(\underline{t}\,\sigma)$.

Practice suggests that the specifications which are useful for proving termination have the following interesting property which we named "well behaviour with respect to substitutions".

3.1.4 Definition

Let p be a n-ary predicate symbol, let $\{Pre_p(x_1, \ldots, x_n)\}p(x_1, \ldots, x_n)\{Post_p(x_1, \ldots, x_n)\}$ be a pre/post specification for it and $\mathcal{T}$ an associated theory. Such *a predicate specification is well-behaved in $\mathcal{T}$ with respect to substitutions* (shortly, well-behaved) if, for all the instances $p(t_1, \ldots, t_n)$ of $p(x_1, \ldots, x_n)$ and for all the substitutions of the variables $z_1, \ldots, z_m$ in $t_1, \ldots, t_n$, both the precondition and the postcondition imply their corresponding instances

$\mathcal{T} \vdash \forall \underline{t} \forall \sigma.\ (Pre(\underline{t}) \rightarrow Pre(\underline{t}\sigma)) \wedge (Post(\underline{t}) \rightarrow Post(\underline{t}\sigma))$.

An asserted program $\mathcal{P}$ has a well-behaved specification (with respect to substitutions) if all the specifications of the predicate symbols in $\mathbf{Pred}_{\mathcal{P}}$ are well-behaved with respect to substitutions.

Intuitively, if a predicate specification, which is well-behaved with respect to substitutions, holds for an instance $p(t_1, \ldots, t_n)$ of the predicate $p(x_1, \ldots, x_n)$, it can be further instantiated but not falsified through unification. This corresponds also to say that the variables $z_1, \ldots, z_m$ in $t_1, \ldots, t_n$ are universally quantified. This is obviously true for the specifications of data properties related to the declarative semantics of the predicate, as the ones we considered in [Bos89], since these are basically properties of ground terms. This is also true in the specifications useful for proving termination, since they deal with persistent structural properties of the data. Rigidity has been defined exactly to characterize this persistency property.

Examples.

The specification given in our example is well-behaved with respect to substitutions since both the preconditions and the postconditions are based on the predicate rigid which, by rules 1.a and 2.a (definition and persistency) in $\mathcal{R}$, is well-behaved. But this is not the only well-behaved predicate dealing with term properties. For example, let $\equiv$ be the identity relation, st be the subterm relation and let not-unifiable be defined as

$not\text{-}unifiable(x_1, \ldots, x_n) =_{def} \forall\, x_i, x_j \in (x_1, \ldots, x_n). \forall\, \sigma.\ (i \neq j) \rightarrow \neg\, (x_i\sigma \equiv x_j\sigma)$.

Then the following specifications for a predicate symbol q are also well-behaved with respect to substitutions:

$\{x \equiv y\}\ q(x, y, z)\ \{x \equiv y \wedge not\text{-}unifiable(z, x, y)\}$

$\{x\ st\ y\}\ q(x, y, z)\ \{x\ st\ y \wedge ground(z)\}$

$\{rigid(x) \wedge |x| \leq |y|\}\ q(x, y, z)\ \{rigid(x, z) \wedge |x| \leq |y|\}$

On the other hand a specification containing information such as var(x) is obviously not well-behaved with respect to substitutions.

Note that if $Pre_q(\underline{x})$ is well-behaved, $q(\underline{x})$ is always correct with respect to the specification $\{Pre_q(\underline{x})\}\ q(\underline{x})\ \{Pre_q(\underline{x})\}$.

Formulating a specification does not guarantee that it correctly describes properties of $\mathcal{P}$, and so we have to prove it. This proof can be done by an inductive method [Dra88, Bos89]. If the specification is well-behaved with respect to substitutions, as it is the case in most termination proofs, then we can apply

the simplified inductive method proposed in [Bos89]. The method is based on the following sufficient criterion. The proof of its correctness can be found in [Bos89, Bos91].

3.1.5 Sufficient Criterion for correctness of a program with respect to a well-behaved specification.

Let $\mathcal{P}$ be an asserted program with a well-behaved specification and let $\mathcal{T}$ be its associated theory. $\mathcal{P}$ is correct with respect to its specification in $\mathcal{T}$ if, for each clause $a_0(\underline{t}_0)$:- $a_1(\underline{t}_1)$, ..., $a_n(\underline{t}_n)$. in $\mathcal{P}$, the following two conditions are satisfied:

1) $\mathcal{T}$ |- $\forall \underline{x}.\ (Pre_{a0}(\underline{t}_0) \wedge (\wedge_{i=1}^{k-1} Post_{ai}(\underline{t}_i))) \rightarrow Pre_{ak}(\underline{t}_k)$, for all k in 1, ..., n;

2) $\mathcal{T}$ |- $\forall \underline{x}.\ (Pre_{a0}(\underline{t}_0) \wedge (\wedge_{i=1}^{n} Post_{ai}(\underline{t}_i))) \rightarrow Post_{a0}(\underline{t}_0)$.

where $\underline{x}$ corresponds to the tuple of variables in $\underline{t}_0$, ..., $\underline{t}_n$ and the universal quantification is in $\mathcal{Term}_{\mathcal{P}}$.

Example.

We prove the correctness of the specification for $\mathcal{P}_2$. The associated theory $\mathcal{T}$ contains the definition of the norm *size* and the theory $\mathcal{R}$ of section 2. Since the specification is well-behaved, we can apply the sufficient criterion 3.1.5.

For clarity' sake we keep program variables instead of using metavariables.

Let us first consider the clauses defining the predicate $s\_m(x_1, x_2, x_3)$.

Sufficient criterion 3.1.5 applied to facts, with n=0, requires to prove the implication:

$\mathcal{T}$ |- $\forall \underline{x}.\ (Pre_{a0}(\underline{t}_0) \rightarrow Post_{a0}(\underline{t}_0))$.

Then, for clauses 4, 5 and 6, we have to prove:

4) $(rigid([]) \vee rigid([], [])) \rightarrow (rigid([],[],[]) \wedge (|[]|=|[]|+|[]|) \wedge ((|[]|>1) \rightarrow ((|[]|>|[]|) \wedge (|[]|>|[]|))))$

5) $\forall A.\ (rigid([A]) \vee rigid([], [A])) \rightarrow (rigid([A],[],[A]) \wedge (|[A]|=|[]|+|[A]|)$
$\wedge ((|[A]|>1) \rightarrow ((|[A]|>|[]|) \wedge (|[A]|>|[A]|))))$

6) $\forall A.\ (rigid([A]) \vee rigid([A], [])) \rightarrow (rigid([A],[A],[]) \wedge (|[A]|=|[A]|+|[]|)$
$\wedge ((|[A]|>1) \rightarrow ((|[A]|>|[A]|) \wedge (|[A]|>|[]|))))$

which are immediately derivable by using rule 3.a in $\mathcal{R}$ (syntactic characterization), arithmetics and the definition of the norm *size*.

For clause 7, since n=1, we have to prove the two formulas

7.1) $\forall$ A,B,Cs,Es,Ds. (rigid([A,B|Cs]) $\vee$ rigid([A|Ds], [B|Es])) $\rightarrow$ (rigid(Cs) $\vee$ rigid(Ds, Es));

7.2) $\forall$ A,B,Cs,Es,Ds. ((rigid([A,B|Cs]) $\vee$ rigid([A|Ds], [B|Es])) $\wedge$ rigid(Cs, Ds, Es) $\wedge$
(|Cs|=|Ds|+|Es|) $\wedge$ ((|Cs|>1) $\rightarrow$ ((|Cs|>|Ds|) $\wedge$ (|Cs|>|Es|))))
$\rightarrow$
(rigid([A,B|Cs], [A|Ds], [B|Es]) $\wedge$ (|[A,B|Cs]|=|[A|Ds]|+|[B|Es]|) $\wedge$
((|[A,B|Cs]|>1) $\rightarrow$ ((|[A,B|Cs]|>|[A|Ds]|) $\wedge$ (|[A,B|Cs]|>|[B|Es]|))))

The proof of 7.1 immediately follows from rule 4 (term decomposition).

To prove 7.2 we note that: rigid([A,B|Cs], [A|Ds], [B|Es]) follows from rigid(Cs, Ds, Es) by rule 5 (term composition). The second conjunct of the consequent, (|[A,B|Cs]| = |[A|Ds]| + |[B|Es]|), is implied by the second conjunct of the antecedent, (|Cs| = |Ds| + |Es|) and the definition of the norm *size*. Finally, as regards the third conjunct of the consequent, since |[A,B|Cs]| >1 is true, we have to prove ((|[A,B|Cs]| > |[A|Ds]|) $\wedge$ (|[A,B|Cs]| > |[B|Es]|)). This follows from (|Cs| = |Ds| + |Es|), the second conjunct of the antecedent, since this implies that both |Ds| and |Es| are less or equal to |Cs|.

For clause 8 the proof is analogous.

As regards the other clauses which define the predicate p, the verification is very easy. Indeed, it could be done automatically. In fact, since rigidity properties are very easy to deal with in the theory $\mathfrak{R}$, if the specification concerns only rigidity of terms, we can apply the following simple procedure to each clause. This exactly corresponds to apply the sufficient criterion 3.1.5 in $\mathfrak{R}$.

3.1.6 Sufficient criterion for rigidity properties:

a) mark all the terms in the clause which can be stated rigid from their syntax; this corresponds to apply the rule 3.a in $\mathfrak{R}$ (syntactic characterization);

b) mark all the terms in the head of the clause which are declared rigid by its precondition;

c) apply the rules of theory $\mathfrak{R}$ to the terms in the clause in order to get as much further information as possible on rigidity of terms;

d) for each atom in the body of the clause, from left to right, do
if its precondition has been satisfied then
- mark the terms of the atom which in its postcondition are specified to be rigid;
- apply the rules of theory $\mathfrak{R}$ to the terms in the clause in order to get as much further information as possible on rigidity of terms;

else the program is not precondition persistent or it is not correct with respect to the specification;

e) if step (d) terminates successfully and the postcondition of the head is satisfied then
the verification terminates successfully for that clause
else the program is not precondition persistent or it is not correct with respect to the specification.

Example.

Let us apply 3.1.6 to clauses 1, 2 and 3 which define the predicate p in our example. We use the notation t^r to mark a term t which is rigid.

1) after step (a) p([]^r, []^r).

Thus, from step (e), this rule satisfies the sufficient criterion.

2) after steps (a) p([A]^r, [A]^r).

Thus, from step (e), this rule satisfies the sufficient criterion.

3) after steps (a, b)
p([X, Y|Xs]^r, Zs) :- s_m([X,Y|Xs], Us, Vs), p(Us, Ts), p(Vs, Ws), s_m(Zs, Ts, Ws).
after step (c)
p([X, Y|Xs]^r, Zs) :- s_m([X,Y|Xs]^r, Us, Vs), p(Us, Ts), p(Vs, Ws), s_m(Zs, Ts, Ws).

step (d), for each atom in the body, consists in checking that the precondition of the atom has been satisfied and, since this is true, marking the terms which the postcondition specifies to be rigid

p([X, Y|Xs]^r, Zs) :- s_m([X,Y|Xs]^r, Us^r, Vs^r), p(Us^r, Ts), p(Vs^r, Ws), s_m(Zs, Ts, Ws).
p([X, Y|Xs]^r, Zs) :- s_m([X,Y|Xs]^r, Us^r, Vs^r), p(Us^r, Ts^r), p(Vs^r, Ws), s_m(Zs, Ts^r, Ws).
p([X, Y|Xs]^r, Zs) :- s_m([X,Y|Xs]^r, Us^r, Vs^r), p(Us^r, Ts^r), p(Vs^r, Ws^r), s_m(Zs, Ts^r, Ws^r).
p([X, Y|Xs]^r, Zs^r) :- s_m([X,Y|Xs]^r, Us^r, Vs^r), p(Us^r, Ts^r), p(Vs^r, Ws^r), s_m(Zs^r, Ts^r, Ws^r).

Thus, from step (e), we obtain that also this clause satisfies the sufficient criterion.

We are interested in proving universal termination of a class of goals in a logic program. The class can be characterized by the common sequence of predicate symbols of the atoms in the goals and by a class description which is a first order formula describing some properties of the goals in the class. The termination proof clearly depends on these properties.

3.1.7 Definition

Let G = :- $q_1(\underline{x}_1)$, ..., $q_m(\underline{x}_m)$. be a general goal, $\mathcal{L}$ a first order language which contains $Term_P$, $Des_G(\underline{x}_1, ..., \underline{x}_m)$ any formula of $\mathcal{L}$ and $\mathcal{T}$ a theory.

The class of goals $\mathcal{G} = (G, Des_G(\underline{x}_1, ..., \underline{x}_m))$ is the class which contains all the instances, $G_0 = q_1(\underline{t}_1), ..., q_m(\underline{t}_m)$., of G such that Des_{G0} holds in $\mathcal{T}$:

$\mathcal{T}$ |- $Des_G(\underline{t}_1, ..., \underline{t}_m)$.

A class of goals $\mathcal{G} = (G, Des_G(\underline{x}_1, ..., \underline{x}_m))$ *is well-behaved in* $\mathcal{T}$ *with respect to substitutions* (shortly, well-behaved) if for all instances :- $q_1(\underline{t}_1), ..., q_m(\underline{t}_m)$. of G, for all substitutions σ of the variables $z_1, ..., z_k$ in $\underline{t}_1, ..., \underline{t}_m$, the class description implies its corresponding instance

$\mathcal{T}$ |- $\forall \underline{t}_1, ..., \underline{t}_m \forall \sigma$. $Des_G(\underline{t}_1, ..., \underline{t}_m) \rightarrow Des_G(\underline{t}_1\sigma, ..., \underline{t}_m\sigma)$.

Let $\mathcal{P}$ be a program asserted with a well-behaved specification $\mathcal{S}$, $\mathcal{T}$ the associated theory and $\mathcal{G} = (G, Des_G(\underline{x}_1, ..., \underline{x}_m))$ a well-behaved class of goals.

$\mathcal{G}$ fits with the specification $\mathcal{S}$ in $\mathcal{T}$ if

i) $\mathcal{T}$ |- $Des_G(\underline{x}_1, ..., \underline{x}_m) \rightarrow Pre_{q1}(\underline{x}_1)$ and

ii) $\mathcal{T}$ |- $(Des_G(\underline{x}_1, ..., \underline{x}_m) \wedge (\wedge_{i=1}^{k-1} Post_{qi}(\underline{x}_i))) \rightarrow Pre_{qk}(\underline{x}_k)$, for all k, $2 \leq k \leq m$.

An asserted queried program $(\mathcal{P}, \mathcal{S}, \mathcal{G})$ consists of a program $\mathcal{P}$, a well-behaved specification $\mathcal{S}$ for $\mathcal{P}$, and a well-behaved class of goals $\mathcal{G}$.

3.1.8 Definition

The asserted queried program $(\mathcal{P}, \mathcal{S}, \mathcal{G})$ *is correct in* $\mathcal{T}$, if

1) $\mathcal{P}$ is correct with respect to its specification in $\mathcal{T}$;
2) $\mathcal{G}$ fits with the specification $\mathcal{S}$ in $\mathcal{T}$.

Examples.

By looking at the heads of the clauses defining $p(x_1, x_2)$ in $\mathcal{P}_2$, our intuition suggests that the computation of a goal G= :- $p(t_1, t_2)$. terminates when the first parameter t_1 is no more expandable, namely when it is either a ground list or, more in general, it is not ground but it has a fixed length. Then, by considering the norm *size* defined in the previous section, we would like to verify that any goal in the class $\mathcal{G}$ = (:-$p(x_1, x_2)$., rigid(x_1)), universally terminates in $\mathcal{P}_2$. Note that the class is well-behaved. In practice it is exactly such well-behaviour which allows us to prove termination. Both G_1 = :- p([a, b, c], Xs). and G_2 = :- p([X, a, Y], Xs). belong to this class of goals. $\mathcal{G}$ obviously fits with the specification since $p(x_1, x_2)$ is atomic and rigid(x_1)=Pre_p. Hence $(\mathcal{P}, \mathcal{S}, \mathcal{G})$ is correct in $\mathcal{T}$ since we proved that $\mathcal{P}$ is correct with respect to the given specification.

Not always the verification that the class $\mathcal{G}$ fits with the specification is so simple. Let us consider the goal G = :- $s\_m(x_1, x_2, x_3)$, $p(x_4, x_5)$. and $Des_G(x_1, x_2, x_3, x_4, x_5)$ be rigid(x_1) $\wedge$ (x_1=[y, y | z]) $\wedge$ (x_3=x_4). $\mathcal{G}$ contains all the instances of :- $s\_m(x_1, x_2, x_3)$, $p(x_4, x_5)$. with the third parameter of s_m equal to the first

of p, a rigid list (of fixed length) as the first parameter of s_m and where such rigid list has the first two elements which are equal. Therefore, $(\mathcal{P}_2, \mathcal{G})$ represents a class of computations in $\mathcal{P}_2$. The class of goals is well-behaved since only the predicates rigid and equality are used. $\mathcal{G}$ fits with the specification given in our first example. In fact

i) $(\text{rigid}(x_1) \wedge (x_1=[y, y \mid z]) \wedge (x_3=x_4)) \rightarrow \text{rigid}(x_1) \vee \text{rigid}(x_2, x_3)$ $\quad (\text{Des}_G \rightarrow \text{Pre}_{s\_m})$

ii) $(\text{rigid}(x_1) \wedge (x_1=[y, y \mid z]) \wedge (x_3=x_4) \wedge$ $\quad ((\text{Des}_G \wedge$

$\text{rigid}(x_1, x_2, x_3) \wedge (|x_1|=|x_2|+|x_3|) \wedge ((|x_1|>1) \rightarrow ((|x_1|>|x_2|) \wedge (|x_1|>|x_3|))))$ $\quad \text{Post}_{s\_m})$

$\rightarrow \text{rigid}(x_4)$ $\quad \rightarrow \text{Pre}_p)$

3.1.9 Definition

Let $(\mathcal{P}, \mathcal{S}, \mathcal{G})$ be an asserted queried program correct in $\mathcal{T}$. *$(\mathcal{P}, \mathcal{S}, \mathcal{G})$ is precondition persistent in $\mathcal{T}$* if, for every SLD-derivation of a goal in $\mathcal{G}$, and for any goal $G_i = :- b_1(\underline{s}_1), \ldots, b_n(\underline{s}_n).$, $n \geq 0$, in the derivation,

$\mathcal{T} \mid\!- \text{Pre}_{b1}(\underline{s}_1)$.

3.1.10 Proposition: Let $(\mathcal{P}, \mathcal{S}, \mathcal{G})$ be an asserted queried program correct in $\mathcal{T}$. If the correctness of $\mathcal{P}$ has been proved by the sufficient criterion 3.1.5, then $(\mathcal{P}, \mathcal{S}, \mathcal{G})$ is precondition persistent in $\mathcal{T}$.

Proof.

See [Bos91].

3.2 Proving universal termination

We are interested in proving universal termination of a class of goals, which means that for every goal G_0 in the given class, all computations for G_0 terminate (G_0 has a finite computation tree). Hence, if we assume SLD-resolution and the Prolog selection rule, the order of clauses, which corresponds to the order of solutions, is not relevant, while the order of atoms in the bodies of clauses can affect termination.

As in imperative programming, the universal termination proof consists in

i) finding all the loops;

ii) proving that these loops actually terminate.

In order to identify all the loops, we can consider *the specific graph associated with the program $\mathcal{P}$ and the class of goals $\mathcal{G}$*. This is similar to what has been called U-graph in [Wa89] and it is a modification of the dependency graph (here we adopt the general terminology for graph of [Ber73]). The specific graph represents the computations of $\mathcal{G}$ in $\mathcal{P}$ by making clear both the dependencies and the unification relations among some atoms in the program and by showing the maximal strongly connected subgraphs (shortly: m.s.c.s.) where the program can loop.

3.2.1 Definition

Let $\mathcal{P}$ be a program and $\mathcal{G} = (G, \text{Des}_G)$ a class of goals. *$g(\mathcal{P})$ is the directed graph associated to the program $\mathcal{P}$* and it is defined as follows:

1) the vertices of $\mathcal{G}(\mathcal{P})$ are the atoms occurring in $\mathcal{P}$: multiple occurrences of the same atom are considered as distinct vertices;
2) (A, B) is an arc of $\mathcal{G}(\mathcal{P})$ iff
   i) A is the head of a clause c and B is in the body of c; this is a *clause arc*;
   ii) A is in the body of a clause c_1 and, after renaming all the variables, it unifies, with the head B of a clause c_2; this is an *unifier arc*.

The specific graph of $\mathcal{P}$ and $\mathcal{G}$, $\mathcal{SG}(\mathcal{P}, \mathcal{G})$, is obtained from $\mathcal{G}(\mathcal{P})$ by

1) marking all the vertices in $\mathcal{G}(\mathcal{P})$ whose predicate symbol occurs in G;
2) deleting all the vertices and arcs which do not lay on a path connecting a marked atom with a not trivial maximal strongly connected subgraph of $\mathcal{G}(\mathcal{P})$.

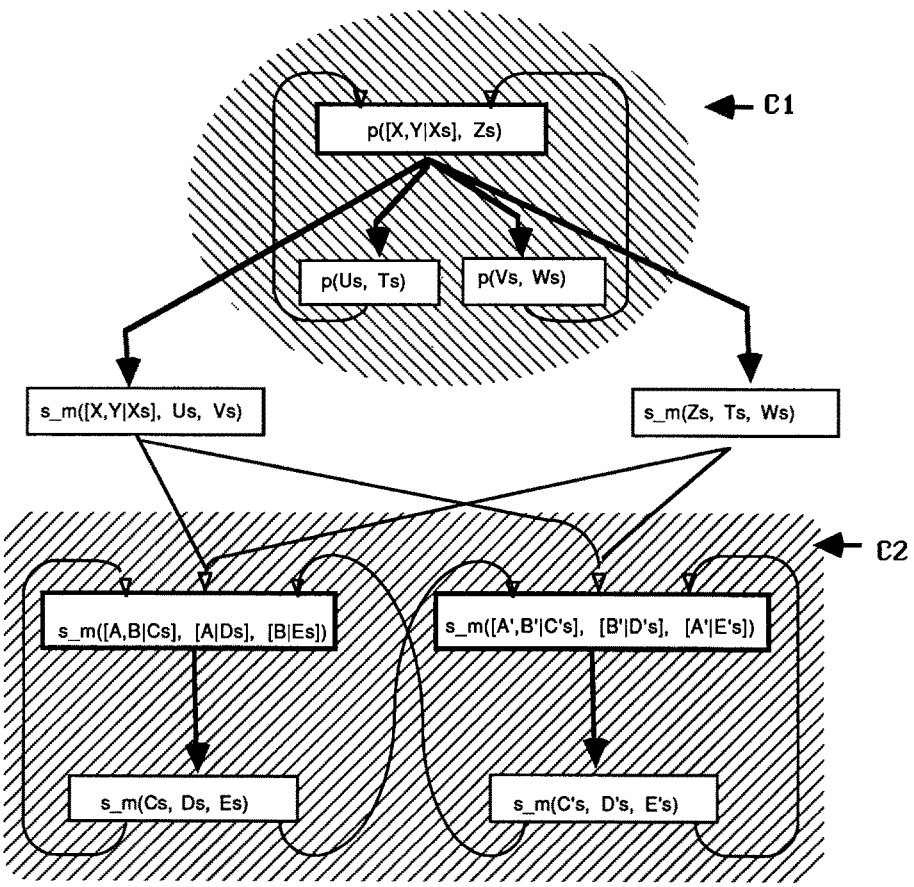

Fig. 1 The specific graph of the program $\mathcal{P}_2$ and the class (:- $p(x_1, x_2)$., rigid(x_1))

Note that the specific graph $\mathcal{SG}(\mathcal{P}, \mathcal{G})$ does not depend on the class description Des_G, but it depends only on $\mathcal{P}$ and the common sequence of predicate symbols of the atoms in $\mathcal{G}$.

Example.

The specific graph of $\mathcal{P}_2$ and (:- p(x_1, x_2)., rigid(x_1)). is shown in Fig. 1. Clause arcs are represented by thicker lines and unifier arcs by thinner ones. The (not trivial) maximal strongly connected subgraph (m.s.c.s.) are $\mathcal{C}_1$, $\mathcal{C}_2$ and they are displayed inside dashed areas. $\mathcal{C}_1$ contains two elementary circuits, $\mathcal{C}_2$ three. In order not to generate confusion in the termination proof, the variables in clause 8 have been renamed; this is obviously always possible, since variables are implicitly universally quantified in each clause.

Each infinite SLD-derivation in $\mathcal{P}$ of a goal in $\mathcal{G}$ has a corresponding infinite path in the specific graph [Fab90, Bos91]. On the other hand an infinite path in a finite graph can be obtained if and only if it traverses an elementary circuit an infinite number of times. Every elementary circuit belongs to one m.s.c.s. in $\mathcal{Sg}(\mathcal{P}, \mathcal{G})$. Hence, analogously to what is done in imperative programming, *in order to prove universal termination of $\mathcal{G}$ in $\mathcal{P}$ it is necessary and sufficient to prove that for all the elementary circuits in the specific graph, there is a function on the instances of atoms in the not trivial maximal strongly connected subgraph (m.s.c.s), with values in a well-founded set, which decreases at each traversal of the circuits during the computation of any goal in the class.* A characterization of such a function **can be** found in [Fab90, Bos91], we briefly recall here some interesting concepts.

Let $\mathcal{C}$ be a not trivial m.s.c.s. in $\mathcal{Sg}(\mathcal{P}, \mathcal{G})$, we denote by $\mathcal{A}_{(\mathcal{P}, \mathcal{G})}(\mathcal{C})$ the set of all the instances of atoms in $\mathcal{C}$.

3.2.2 Definition

Let $\mathcal{C}$ be a not trivial m.s.c.s. in $\mathcal{Sg}(\mathcal{P}, \mathcal{G})$. *An ordering function of $\mathcal{C}$, $f_{\mathcal{C}}$,* is a total function which maps $\mathcal{A}_{(\mathcal{P}, \mathcal{G})}(\mathcal{C})$ into $\mathcal{N}$:

$$f_{\mathcal{C}} : \mathcal{A}_{(\mathcal{P}, \mathcal{G})}(\mathcal{C}) \rightarrow \mathcal{N}.$$

Example.

For $\mathcal{P}_2$ and (:- p(x_1, x_2)., rigid(x_1)). we can define the following ordering functions:

$f_{\mathcal{C}1}$: p(x_1, x_2) $\rightarrow |x_1|_{size}$

$f_{\mathcal{C}2}$(s_m(x_1, x_2, x_3)) = $|x_1|$ if rigid(x_1) and not (rigid(x_2, x_3));

$|x_2| + |x_3|$ if rigid(x_2, x_3) and not (rigid(x_1));

$\min(|x_1|, |x_2| + |x_3|)$ otherwise.

The following definition is meant to characterize which are the atoms actually computed in $\mathcal{A}_{(\mathcal{P}, \mathcal{G})}(\mathcal{C})$.

3.2.3 Definition

Let A be an atom in a m.s.c.s. $\mathcal{C}$ in $\mathcal{Sg}(\mathcal{P}, \mathcal{G})$. An instance A' of A is *a calling instance of A in $\mathcal{C}$* if one of the following two conditions are satisfied

1) A is an atom in the body of a clause and A' is the selected atom (i.e. the first) of a resolvent R in a SLD-derivation in $\mathcal{P}$ of any goal G_0 in $\mathcal{G}$;

2) A is the head of a clause and there is a resolvent R in a SLD-derivation in $\mathcal{P}$ of any goal G_0 in $\mathcal{G}$ and a substitution σ such that :

$R=B_1, \ldots, B_m, \sigma=mgu(B_1, A)$ and $A'=A\sigma$.

We denote by $\mathcal{ACI}_{(\mathcal{P}, \mathcal{G})}(\mathcal{C})$ the set of the calling instances of the atoms in $\mathcal{C}$.

The ordering functions which are more interesting for our proofs satisfy a strong property, namely their value cannot increase if a calling instance of an atom in the circuit is further instantiated. We call such a property "not increasing with respect to substitutions".

3.2.4 Definition

Let $\mathcal{C}$ be a m.s.c.s. in $\mathcal{Sg}(\mathcal{P}, \mathcal{G})$, and $f_{\mathcal{C}}$ an ordering function mapping $\mathcal{A}_{(\mathcal{P}, \mathcal{G})}(\mathcal{C})$ into $\mathcal{N}$. *The ordering function is not increasing with respect to substitutions* if for all the atoms A in $\mathcal{ACI}_{(\mathcal{P}, \mathcal{G})}(\mathcal{C})$ and for all the substitutions σ,

$f_{\mathcal{C}}(A) \geq f_{\mathcal{C}}(A\sigma)$.

The property of being non increasing with respect to substitutions is essential in order to simplify the proof that the ordering function decreases at each traversal of each elementary circuit during any computation. This property strictly depends in general on terms rigidity.

Example.

We prove that the ordering functions we defined are not increasing with respect to substitutions.

1) $f_{\mathcal{C}1} : p(x_1, x_2) \rightarrow |x_1|_{size}$.

For all the calling instances, $p(t_1, t_2)$, of the atoms in $\mathcal{C}_1$ the precondition $\{rigid(t_1)\}$ is satisfied because of the precondition persistency of $(\mathcal{P}_2, \mathcal{S}, \mathcal{G})$ and the well-behaviour of the pre/post specification. Then $f_{\mathcal{C}1}(p(t_1, t_2))$ is even invariant under substitutions.

2) $f_{\mathcal{C}2}(s\_m(x_1, x_2, x_3)) =$

$\|x_1\|$	if $rigid(x_1)$ and not $(rigid(x_2, x_3))$;
$\|x_2\| + \|x_3\|$	if $rigid(x_2, x_3)$ and not $(rigid(x_1))$;
$\min(\|x_1\|, \|x_2\| + \|x_3\|)$	otherwise.

For all the calling instances $s\_m(t_1, t_2, t_3)$ of the atoms in $\mathcal{C}_2$ the precondition $\{rigid(t_1) \vee rigid(t_2, t_3)\}$ is satisfied because of the precondition persistency of $(\mathcal{P}_2, \mathcal{S}, \mathcal{G})$ and the well-behaviour of the pre/post specification. We can consider separately the three cases.

i) $rigid(t_1)$ and not $(rigid(t_2, t_3))$ and then $f_{\mathcal{C}2}(s\_m(t_1, t_2, t_3)) = |t_1|$.

By applying a substitution σ to $s\_m(t_1, t_2, t_3)$, $t_1\sigma$ is still rigid and both $t_2\sigma$ and $t_3\sigma$ might become rigid too. Then either $f_{\mathcal{C}2}(s\_m(t_1, t_2, t_3)\sigma) = |t_1\sigma| = |t_1|$, or $f_{\mathcal{C}2}(s\_m(t_1, t_2, t_3)\sigma) = \min(|t_1\sigma|, |t_2\sigma| + |t_3\sigma|) \leq |t_1\sigma| = |t_1|$. Hence in both cases $f_{\mathcal{C}2}(s\_m(t_1, t_2, t_3))$ has not increased with respect to the substitution.

ii) $rigid(t_2, t_3)$ and not $(rigid(t_1))$.

Analogous to the previous one.

iii) $rigid(t_1, t_2, t_3)$.

Trivial because of 1.a and 2.a in $\mathcal{R}$ (definition and persistency).

We can now state the sufficient criterion for universal termination of a well-behaved class of goals in a program we propose to adopt in practice and whose correctness is proven in [Fab90, Bos91]. By considering only well-behaved specifications and ordering functions which are not increasing with respect to substitutions, the proof method becomes rather simple since it formalizes our usual reasoning on program termination, based only on the program text and on what we know about the goal, without simulating real execution.

3.2.5 Sufficient criterion for universal termination of a well-behaved class of goals in a program $\mathcal{P}$.

Let $\mathcal{P}$ be a program, $\mathcal{G}$ a well-behaved class of goals and $\mathcal{T}$ a first order theory. Every goal G_0 in $\mathcal{G}$ universally terminates in $\mathcal{P}$ if

1) there exists a well-behaved specification $\mathcal{S}$ such that the asserted queried program $(\mathcal{P}, \mathcal{S}, \mathcal{G})$ is correct and precondition persistent in $\mathcal{T}$;
2) for every m.s.c.s. $\mathcal{C}$ in $\mathcal{Sg}(\mathcal{P}, \mathcal{G})$, there exists an ordering function, $f_{\mathcal{C}}$, not increasing with respect to substitutions, which is defined in $\mathcal{T}$;
3) for every m.s.c.s. $\mathcal{C}$,
   for every elementary circuit in $\mathcal{C}$, $B_0, H_1, B_1, \ldots, H_n, B_n$,
   where $B_n=B_0$, $1 \le n$, B_{i-1} unifies with H_i,
   H_i is the head and B_i is the k-th atom in the body of the clause
   $h_i(t_0) :- a_1(t_1), \ldots, a_k(t_k), \ldots, a_m(t_m).$,
   the following conditions hold:
   i) for every i, $1 \le i \le n$,
   $\mathcal{T} \vdash \forall \underline{x}.\ (Pre_{hi}(t_0) \wedge (\wedge_{j=1}^{k-1} Post_{aj}(t_j)) \rightarrow (f_{\mathcal{C}}(B_i) \le f_{\mathcal{C}}(H_i)))$,
   where $\underline{x}$ corresponds to the variables in $t_0, t_1, \ldots, t_k$ and the universal quantification is in $\mathcal{T}erm_{\mathcal{P}}$;
   ii) at least one such inequality is strict.

In every elementary circuit in $\mathcal{C}$, for each head atom, H_i, the antecedent in condition (3.i) represents all the information given by the specification before executing the corresponding body atom in the circuit, B_i, with the Prolog computation rule. Therefore condition (3.i) just says that these information can be used for proving that the ordering function cannot increase while passing from H_i to B_i. The restriction to well-behaved specifications and not increasing ordering functions makes condition (3.i) simple to use. Without these restrictions, with a general specification and general ordering functions, in order to prove that a general ordering function cannot increase at each traversal of an elementary circuit, we should consider each substitution actually computed while running along the circuit.

Example.

In our example we have already verified conditions (1) and (2) of 3.2.5. Let us prove condition (3). There are two m.s.c.s. $\mathcal{C}_1$ and $\mathcal{C}_2$, corresponding to the recursive definition of the predicate p and s_m respectively, and they contain five elementary circuits.

$\mathcal{C}_1$) The m.s.c.s. contains two elementary circuits c_1 and c_2. The atoms in c_1 are p([X,Y|Xs], Zs) and p(Us, Ts), the ones in c_2 are p([X,Y|Xs], Zs) and p(Vs, Ws). Let us prove the sufficient condition for termination in c_1. We have to prove

$\mathcal{T}$ |- ∀ (rigid([X,Y|Xs]) ∧ rigid([X,Y|Xs], Us, Ts) ∧ (|[X,Y|Xs]| = |Us| + |Vs| ∧
((|[X,Y|Xs]| >1) → ((|[X,Y|Xs]| > |Us|) ∧ (|[X,Y|Xs]| > |Vs|)))) → (|Us| ≤ |[X,Y|Xs]|).

Since (|[X,Y|Xs]| >1) holds by definition of the norm *size*, the forth conjunct of the antecedent

((|[X,Y|Xs]| >1) → ((|[X,Y|Xs]| > |Us|) ∧ (|[X,Y|Xs]| > |Vs|)))

implies the consequent. Furthermore it allows us to conclude that the inequality is strict and then also the second condition for termination is verified.

The proof for the elementary circuit c_2 is similar.

c_2) The m.s.c.s. contains three elementary circuits c_3, c_4 and c_5.The atoms in c_3 are s_m([A, B|Cs], [A|Ds], [B|Es]) and s_m(Cs, Ds, Es), the ones in c_4 are s_m([A', B'|Cs'], [B'|Ds'], [A'|Es']) and s_m(Cs', Ds', Es'), the ones in c_5 are s_m([A, B|Cs], [A|Ds], [B|Es]), s_m(Cs, Ds, Es), s_m([A', B'|Cs'], [B'|Ds'], [A'|Es']) and s_m(Cs', Ds', Es').

Let us prove the sufficient conditions for termination in c_3:

$\mathcal{T}$ |- ∀ (rigid([A, B|Cs]) ∨ rigid([A|Ds], [B|Es])) →
(f_{c2} (s_m(Cs, Ds, Es)) ≤ f_{c2} (s_m([A, B|Cs], [A|Ds], [B|Es])).

Let us assume rigid([A, B|Cs]) and not rigid([A|Ds], [B|Es]). By rule 4 in $\mathcal{R}$ (term decomposition), rigid([A, B|Cs]) implies rigid(Cs). Because of the definition of the norm *size* we can also prove, by contradiction and by rules 3.a and 3.b in $\mathcal{R}$ (syntactic characterization), that not rigid([A|Ds], [B|Es]) implies not rigid(Ds, Es). Hence the consequent is

(|Cs| ≤ |[A, B|Cs]|)

which holds by definition of *size*. The other two cases, (rigid([A|Ds], [B|Es]) ∧ ¬ rigid([A, B|Cs])) and rigid([A, B|Cs], [A|Ds], [B|Es]), lead to the two consequents

((|Ds| + |Es|) ≤ ([A|Ds]| + |[B|Es]|))

(min(|Cs|, (|Ds| + |Es|)) ≤ min(|[A, B|Cs]|, ([A|Ds]| + |[B|Es]|))

which are also immediate consequence of the definition of the norm *size*.

Since strict inequalities hold in all the three cases, condition (3.ii) is also satisfied.

The proofs for the elementary circuits c_4 and c_5 are similar.

Note that the conjunct ((|x_1| >1) → ((|x_1| > |x_2|) ∧ (|x_1| > |x_3|))) in the postcondition of the predicate s_m(x_1, x_2, x_3) is necessary in order to insure that the ordering function on c_1 and c_2 actually decreases. Let us consider now the other program we gave in the first section:

$\mathcal{P}_1$: 1: p([], []).
2: p([X], [X]).
3: p([X,Y|Xs], Zs) :- s_m([X,Y|Xs], Us, Vs), p(Us, Ts), p(Vs, Ws), s_m(Zs, Ts, Ws).
4: s_m([], [], []).
5: s_m([A|Cs], [A|Ds], Es) :- s_m(Cs, Ds, Es).
6: s_m([A|Cs], Ds, [A|Es]) :- s_m(Cs, Ds, Es).

The program looks similar but simpler with respect to the previous one. The program is still correct with respect to the previous specification, except for ((|x_1| >1) → ((|x_1| > |x_2|) ∧ (|x_1| > |x_3|))) which does not hold any more as postcondition of s_m(x_1, x_2, x_3). This should make us suspect that a goal :- p(t_1, t_2). does not terminate in $\mathcal{P}_1$ even when rigid(t_1) holds, and this is the case! Note that with respect to a goal :-

$s\_m(t_1, t_2, t_3)$. which belongs to any class of goals which fits with the specification, the computation does terminate, and the proof can be obtained by simplifying the corresponding one in $\mathbf{P}_2$.

A few other examples of termination proofs are sketched in the following.

1) Let us consider an example which has been proposed in the literature, [Ull88, Plü90a, Apt90], as particularly interesting:

```
1:  mergesort([], []).
2:  mergesort([E], [E]).
3:  mergesort([E,F|U], V) :-  s([E,F|U], W, Y), mergesort(W, X), mergesort(Y, Z), m(X, Z, V).
4:  m(X, [], X).
5:  m([], X, X).
6:  m([A|X], [B|Y], [A|Z]) :- A≤B, m(X, [B|Y], Z).
7:  m([A|X], [B|Y], [B|Z]) :- A>B, m([A|X], Y, Z).
8:  s([], [], []).
9:  s([E|U], [E|V], W) :- s(U, W, V).
```

where m stands for merge and s for split. Let us consider the well behaved class of goals (:- mergesort(x_1, x_2)., ground(x_1)). The problem is in clause 9 because of the switching of the last two parameters in the recursive invocation s(U, W, V). In fact the methods proposed in [Ull88, Plü90a] can automatically generate for the predicate s([E,F|U], W, Y) only the inequality ($|[E,F|U]| \leq |W| + |Y|$), but this is not enough for guaranteeing that mergesort actually terminates. In fact we have to prove that the recursive invocations of mergesort are applied to smaller lists and for this we need the stronger information ($|[E,F|U]| > |W|$) $\wedge$ ($|[E,F|U]| > |Y|$). By unfolding s([E,F|U], W, Y) in clause 3, the automatic proof would become possible. Since we do not deal with automatic proofs, we do not need any program transformation, we have just to choose the proper specification and ordering function. In this example we can define

- the specification:

  $\{rigid(x_1)\}$ mergesort(x_1, x_2) $\{rigid(x_1, x_2)\}$

  $\{rigid(x_1, x_2)\}$ m(x_1, x_2, x_3) $\{rigid(x_1, x_2, x_3)\}$

  $\{rigid(x_1)\}$ s(x_1, x_2, x_3) $\{rigid(x_1, x_2, x_3) \wedge (|x_1| = |x_2| + |x_3|) \wedge$
  $(|x_1| = 1) \rightarrow (|x_2| = 1) \wedge ((|x_1| > 1) \rightarrow ((|x_2| \neq 0) \wedge (|x_3| \neq 0)))\}$

  $\{true\}$ A≤B $\{true\}$

  $\{true\}$ A>B $\{true\}$;

- the ordering functions:

on the m.s.c.s. $\mathbf{C}_1$ produced by the recursive definition of mergesort(x_1, x_2)

f_{C1} : mergesort(x_1, x_2) $\rightarrow |x_1|_{size}$;

on the m.s.c.s. $\mathbf{C}_2$ produced by the recursive definition of m(x_1, x_2, x_3)

f_{C2} : m(x_1, x_2, x_3) $\rightarrow |x_1|_{size} + |x_2|_{size}$

and on the m.s.c.s. $\mathbf{C}_3$ produced by the recursive definition of s(x_1, x_2, x_3)

f_{C3} : s(x_1, x_2, x_3) $\rightarrow |x_1|_{size}$.

Note how the interesting part of the termination proof consists in the formulation of the proper norms, specifications and ordering functions. The actual verification is rather simple and boring, just checking that the previous choices were the proper ones. Obviously the less trivial is the recursive structure of the program, the more involved the proof becomes.

2) Now let us consider the simple example of the transitive closure of a base relation we gave in the introduction, without clause 8 which was source of non termination.

$\mathcal{P}_3$: 1: c(X, Y) :- base(X, Y).
2: c(X, Y) :- base(X, Z), c(Z, Y).
3: base(a, b).
4: base(a, c).
5: base(a, d).
6: base(b, c).
7: base(c, d).

In this case the data domain is flat, terms are constant without structure, and the termination for any goal, $\mathcal{G}$ = (:- c(x_1, x_2)., true)), depends on the absence of cycles in the data domain. Our method applies also to this cases. We have to choose a norm $|\ldots|_f$ corresponding to the order defined by the base relation. The following specification collects the information related to termination

{true} c(x, y) {true}
{true} base(x, y) {rigid(x, y) $\wedge$ ($|y|_f > |x|_f$)}.

The specification is clearly well-behaved with respect to substitutions. Note that rigidity in this case is equivalent to groundness.

We could choose the norm

$|t|_f = 0$, if t is a variable
$|a|_f = 1$, $|b|_f = 2$, $|c|_f = 3$, $|d|_f = 4$.

and the ordering function

$f_C(c(x, y)) = (4 - |x|_f)$

on the only m.s.c.s. C, determined by the recursive definition of c(x, y).

This ordering function is not increasing with respect to substitutions.

3) Finally we give an example where the class description, essential to insure termination, does not regard only moding or rigidity of terms.

$\mathcal{P}_4$: 1: turn([a|Xs], [a|Xs]).
2: turn([X|Xs], Zs) :- (X≠a), append(Xs, [X], Ts), turn(Ts, Zs).

The program $\mathcal{P}_4$ is meant to accept goals G_0= :- turn(t_1, t_2). with a ground list as the first argument and a variable as the second one. The first list is rotated until the constant a becomes the first element of the list and such permutation of t_1 is supplied as a result in t_2. Then G= :- turn(x_1, x_2). universally terminates in $\mathcal{P}_4$ only if the constant a occurs as an element in the first list argument. We can prove the termination in $\mathcal{P}_4$ of any goal in the well-behaved class (:-turn(x_1, x_2)., (ground_list(x_1) $\wedge$ a$\in x_1$)). A pre/post specification useful for the termination proof is:

{ground_list(x_1) $\wedge$ a$\in x_1$} turn(x_1, x_2) {true}

{ground_list(x_1, x_2)} append(x_1, x_2, x_3) {ground_list(x_1, x_2, x_3) $\wedge$ x_3=x_1·x_2}

{ground(x)} x≠a{ground(x) $\wedge$ (x≠a)}.

The specification is clearly well-behaved since the terms are ground. The only m.s.c.s. $\mathbf{C}$ is the one produced by the recursive definition of the predicate turn and we could associate to it the ordering function

$f_{\mathbf{C}}$ (turn(x_1, x_2)) = $|x_1|_{size}$ - $|x_2'|_{size}$

where $x_1 = x_1'\cdot[a|x_2']$, with x_1' prefix which does not contain a.

The function $f_{\mathbf{C}}$ is not increasing with respect to substitutions since the length of the minimal prefix containing a cannot be modified by substitutions.

In practice it is often the case that the termination of the computation of a goal strictly depends on some information we have about the domain of application of such a goal, that is on its class. In fact, it is common either to reuse already available definitions of predicates or to give definitions of subgoals, as turn(x_1, x_2) in our last example, that are more general than their particular intended use. In our proof method any information about the domain of application of a goal which cannot be modified by the computation itself (well-behaved with respect to substitutions) can be used. Hence well-behaved class descriptions and pre/post specifications are a very powerful tool: they allow one to deal with termination in a modular style, by characterizing the context of invocation.

4. Conclusions

In this paper we defined and studied a class of functions, semi-linear norms, to weight the terms occurring in a pure Prolog program. All the functions in the class have the nice feature of allowing a syntactical characterization of rigid terms, that is terms whose weight does not change under substitution. By exploiting this class of functions, a general proof method for termination of pure Prolog programs with respect to a class of goals can be adapted to deal with most programs in a simple way. The proof method is similar to that suggested in [Wan89] and it consists of

(a) finding all the maximal strongly connected subgraphs (m.s.c.s.) which are reachable from the goal in the directed graph associated to the program;

(b) associating to each m.s.c.s. an ordering function which maps the calling instances of atoms in the m.s.c.s. into a well-founded set;

(c) associating to each predicate in the program a pre/post specification which states terms properties;

(d) proving the correctness of such a specification;

(e) proving, for each m.s.c.s., that the associated ordering function decreases at each traversal of each elementary circuit in the m.s.c.s.

The last two steps of the method can be extremely complex to carry on. Therefore we have been looking for cases where they can be handled in a simple way. In [Bos91] we show that when the pre/post specifications well-behave with respect to substitutions a method can be used to perform step (d) which is much simpler of what is proposed in [Wan89] or [Dra88]. Another simplification with respect to the

general case can be applied at point (e) when also the ordering functions are not increasing under substitution. In practice the class of terminating goals are generally well-behaved and a well-behaved specification as well as not increasing ordering functions can be found. Moreover when the specification states only rigidity of terms the simplification of (d) is drastic and the proof has a strong similarity to mode analysis. The simplified proof method is explained through examples.

The idea of linear norms originates from [Plü90a, Plü90b] but we notice that, in most practical cases, only some particular subterms of a term may affect termination. This leads to the definition of semi-linear norms and to the concept of rigid term as a generalization of ground term. It had been rather surprising for us to see how strong analogies the concept of rigid term presents with that of *bounded goals* proposed in [Apt90]. Indeed, even if coming from completely different theoretical approaches (they study and characterize the class of *left terminating* programs, that is pure Prolog programs which terminate on all ground goals, by means of level mappings [Bez89]), the practical examples they give have a lot in common with ours. In both proposals an inductive proof is used and the decreasing of similar functions has to be proved. Their proofs, based on models, are simple and elegant. Clearly, since they characterize only left terminating programs, there are practical situations where their method does not apply. Our last example, $\mathcal{P}_4$ with the class of goals (:- turn(x_1, x_2)., ground_list(x_1) $\wedge$ (a$\in$ x_1)), is one simple case of termination they cannot capture. On the contrary, all the examples given in [Apt90] can be handled also with our simplified method. Crucial points of their approach are, in our opinion, the questions of how to determine the set of bounded goals and how to find out the suitable level mappings. We guess that the concepts of semi-linear norms and rigid terms could be useful for these questions too.

References

[Apt89] Apt K.R., Bol R.N., Klop J.W., On the safe termination of prolog programs, Proc. ICLP'89, MIT Press, (1989).

[Apt90] Apt K.R., Pedreschi D., Studies in Pure Prolog: Termination, Proc. Symp. on Computational Logic, LNCS (1990).

[Bau88] Baudinet M., Proving Termination Properties of Prolog Programs: A Semantic Approach, Proc. IEEE Symp. on Logic in Computer Science, pp. 336-347 (1988).

[Bez89] Bezem M., Characterizing Termination of Logic Programs, Proc. NACLP'89, pp. 69-80 (1989).

[Ber73] Berge C., Graphs and Hypergraphs, North-Holland Mathematical Library, North-Holland, 1973.

[Bos89] Bossi A., Cocco N., Verifying Correctness of Logic Programs, Proc. TAPSOFT'89, Vol. 2, Springer-Verlag, pp. 96-110 (1989).

[Bos91] Bossi A., Cocco N., Fabris M., Norms on Terms and Their Use in Proving Universal Termination of Logic Programs, C.N.R. Technical Report on Project "Sistemi Informatici e Calcolo Parallelo" (1991).

[Cla77] K.L. Clark, S. Tarnlund, A first order theory of data and programs, Proc. IFIP 77, pp. 939-944 (1977).

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

[Dij76] Dijkstra E.W., A Discipline of Programming, Prentice-Hall, 1976.

[Dra88] Drabent W., Maluszynski J., Inductive Assertion Method for Logic Programs, Theoretical Computer Science, 59, pp. 133-155 (1988).

[Fab90] Fabris M., Sulla Terminazione dei Programmi Logici, tesi di laurea, Dip. di Matematica Pura e Applicata, Università di Padova (1990).

[Flo67] Floyd R.M., Assigning meaning to programs, Proc. AMS Symp. on Applied Mathematics, Amer. Math. Soc., pp. 19-31 (1967).

[Fra85] Frances N., Grumberg O., Katz S., Pnueli A., Proving Termination of Prolog Programs, Proc. Logics of Programs Conf., LNCS 193, pp. 89-105 (1985).

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

[Hoa69] Hoare C.A.R., An axiomatic approach to computer programming, Comm. ACM 12, pp. 576-580, 583 (1969).

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

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

[Plü90a] Plümer L., Termination Proofs for Logic Programs based on Predicate Inequalities, Proc. ICLP'90, pp. 634-648 (1990).

[Plü90b] Plümer L., Termination Proofs for Logic Programs, Lecture Notes in Artificial intelligence 446, Springer-Verlag, 1990.

[Ull88] Ullman J.D., Van Gelder A., Efficient Tests for Top-Down Termination of Logical Rules, JACM, Vol. 35, No. 2, pp. 345-373 (1988).

[Vas86] Vasak T., Potter J., Characterisation of Terminating Logic Programs, Int. Symp. on Logic Programming '86, IEEE, pp. 140-147 (1986).

[Wan89] Wang B., Shryamasunder R.K., Proving Termination of Logic Programs, Internal Report, CS Dept., Penn State University (1989).

[Wan90] Wang B., Shryamasunder R.K., Towards a Characterization of Termination of Logic Programs, Proc. II Int. Workshop on Programming Language Implementation and Logic Programming '90, LNCS 456, pp. 204-221 (1990).