Typed Norms *

A. Bossi, N. Cocco, M. Fabris
Dip. di Matematica Pura e Applicata
Università di Padova
via Belzoni, 7 - 35131 - PADOVA - ITALY
e.mail: Bossi, Cocco@pdmat1.unipd.it

Abstract

In this paper *typed norms*, a class of functions to evaluate terms which have a specified structure, are defined and studied. The required structure is described by means of a *type schema* which has to be *well formed*, that is well defined and unambiguous. By interpreting the type schema in a semantic domain, we associate a set of typed norms to it. Such norms can distinguish among: (1) terms which have the required structure and hence are evaluated in the semantic domain, (2) terms which do not have it and hence are evaluated to "false", (3) terms which could have instances with the required structure and hence are evaluated to "may be". In the first two cases the term cannot change its "main" structure by applying a substitution to it, and hence its value is fixed. This is an interesting property allowing us to describe terms properties which cannot be affected by further computation, once they have been reached. The applications of typed norms can be determined by choosing the language of the type schema and the semantic domain. In the examples we show how a simple proof method for universal termination of pure Prolog programs, we proposed in [BCF91a, BCF91b], is extended to deal with a larger class of programs.

1. Introduction

In [BCF91a, BCF91b] we have studied a rather simple and general technique for proving universal termination of a pure Prolog program (without extra-logical features) with respect to a class of goals. Universal termination is a strong property requiring that all the derivations of a query in the class are finite and hence that all its solutions are reached. The basic idea in our approach was to keep the verification technique as simple as possible and to single out the parts of the proof which can be automatized while retaining the maximum of generality. The

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

generality of the method is due to the fact that it is just an adaptation of the technique which is used in traditional procedural programming: a computation is proved to terminate by associating a partial function to each cycle in it. This function maps the computation states into a well-founded set, and its value has to decrease at each iteration. The simplicity of our proposal is related to an important observation: *when we reason informally about termination of a given query in a program, we consider terms with a structure which is fixed at least in the part traversed by the program.* In this way we get convinced that the traversal is actually finite. We never simulate the actual computation by considering the composition of unifiers in the derivation! We reason only at a syntactic level. This observation leads us to the definition of *rigid terms*, namely terms with the property of having an associated measure which is fixed, independent from substitutions and hence from the computation. We define also *a class of norms, semi-linear ones*, which have the nice property of allowing a syntactic characterization of rigid terms. Our termination proofs make use of predicates annotations in order to express useful information such as rigidity of terms or relations among terms in the same atom. Such annotations, in the form of *pre/post specifications*, must be *well-behaved with respect to substitutions*, which means that, if they hold at a certain point, then they hold through the following computation since they cannot be falsified by unification. Also the *ordering functions* we associate to cycles, are *not increasing with respect to substitutions*. In this way it is possible to reason at a syntactic level, on the program text, thus simplifying the termination proof. Moreover it is possible to automatize to a certain extent rigidity analysis which is basic to the methodology.

Semi-linear norms are characterized by a simple recursive schema which is at the same time their strength and their weakness. The strength lays obviously on the simplicity of use and clarity of definition. A semi-linear norm associates a weight to every term and for every term it can be decided if it is rigid or not with respect to such a norm. A term which is not rigid can become rigid by substitution. However in the definition of semi-linear norms there are some restrictions. The recursive schema of a semi-linear norm gets into the term structure by only one level. Moreover so far it is not defined how different semi-linear norms can be linked to work together. The definition of a semi-linear norm is recursively based only onto itself and it is easy to understand that this is a severe restriction. Then semi-linear norms can be considered as an interesting, but limited, tool for analyzing terms. Furthermore, while evaluating the weight of a term, since we traverse the term into depth, we could also at the same time check or collect other useful information, such as the type of the term. For all these reasons we extended the definition of semi-linear norms thus allowing the application of our verification technique for universal termination to a larger class of programs.

In this paper *typed norms*, a class of functions assigning a value in a semantic domain to terms which have a specified structure, are defined and studied. The required structure is described by means of a *type schema* which has to be *well formed*. The interpretation of a well

formed type schema produces a set of *typed norms*. Such norms are still simple but more powerful than semi-linear ones, while retaining all the properties which make semi-linear norms useful for termination proofs. Typed norms assign a value to a term if and only if it has a precise structure, the one described by the type schema. If the term has such a structure, then this structure and the associated value cannot be modified by further instantiating the term. If the term has not such a structure, it is possible to distinguish between the possibility and the impossibility of gaining it. Both having a fixed structure and the impossibility of gaining it, are well-behaved properties for a term, namely properties which cannot be falsified by applying further substitutions. Type schemata and typed norms give the possibility of performing also some type checking. This may be very interesting if we want to deal with properties like unifiability, which seems fundamental for verifying more complex properties such as existential termination, finite failure or universal termination without finite failure.

The paper is structured in the following way. In section 2 the notation for terms and the language for analyzing them are introduced. In section 3 type schemata are defined in order to describe and verify structural properties of terms. In section 4 type schemata are interpreted, thus associating to them a set of typed norms. Such norms can be used to evaluate the terms satisfying the structural properties defined by the corresponding schema. A few examples of using such typed norms for universal termination proofs are shown in section 5. It is not possible to deal with these examples with semi-linear norms as defined in [BCF91a, BCF91b]. In section 6 the conclusions follow. An example of a different application of typed norms, namely verifying correct typing in a program, is also given.

2. A language for analyzing terms

We use the standard notation and terminology of [Llo87] or [Apt90] for logic programs. Term(F, V) denotes the set of terms built up in the standard way on a set F of n-ary function symbols and on a set V of variables. Var(t) denotes the set of variables of a term t. If $\rho = \{X_1/t_1, ..., X_n/t_n\}$ is a substitution, then we indicate respectively with $Dom(\rho) = \{X_1, ..., X_n\}$ and $Cod(\rho) = \{t_1, ..., t_n\}$ its domain and codomain.

We distinguish between the language of the program, that is the object language, and a language for analyzing it.

2.1 Definition. Let L = <VAR, FUN> be our object language, where VAR ={X, Y, Z, ...} is the set of variables and FUN = $\{f_1, ..., f_m\}$ is the set of n-ary, n≥0, function symbols. A term analyzer language, L*, on L is given by:

L* = <VAR, FUN, MVAR, TYPE, EVAL>

where MVAR = {x, y, z, ...} is a set of new metavariables such that VAR ∩ MVAR = ∅;

TYPE = $\{A_1, ..., A_n\}$ and EVAL = $\{g_1, ..., g_m\}$ are two sets of function symbols, the type function symbols and the evaluable function symbols respectively, such that FUN, TYPE and EVAL are disjoint sets ($TYPE \cap FUN = \emptyset$, $EVAL \cap FUN = \emptyset$ and $TYPE \cap EVAL = \emptyset$).

The type functions, TYPE, are used for checking the membership of a term into a specific set of terms. If the checked term is in the set, the type functions rewrite it into a ground term with only evaluable function symbols. This new term can be evaluated in a semantic domain.

Example. Let us consider L* such that FUN = {o, succ(_), nil, list(_, _)}, TYPE = {Nat, Loa}, EVAL = {0, 1, +(_, _)}.
Nat can be defined so that it selects some terms, representing natural numbers such as succ(succ(o)), and it maps them into other ground terms evaluable in a semantic domain, in our example +(1, +(1, 0)) which can be evaluated to 2.
Similarly Loa (list of anything) could select well formed lists, such as list(X, list(o, nil)), and map them into a rewriting, +(1, +(1, 0)) in our case, which can be interpreted as the length of the list.

2.2 Definition. Let L* = <VAR, FUN, MVAR, TYPE, EVAL> be a term analyzer language.
$\mathbb{T}^*$ = Term(FUN, MVAR $\cup$ VAR).
$\mathbb{T}$ = Term(FUN, VAR) is the set of terms in the object language.
The set of tests on L* is the set of all the terms built up on FUN and MVAR which are not metavariables: $TEST_{L^*}$ = (Term(FUN, MVAR) - MVAR).
Let t' be a term in $\mathbb{T}^*$. The closure of t' in $\mathbb{T}$ is the set of all the terms in $\mathbb{T}$ that are obtained from t' by substituting its variables with terms in $\mathbb{T}$: $Close(t') = \{t \mid t \in \mathbb{T}, \exists \rho. (t = t'\rho)\}$.

Note that tests cannot be metavariables since they are meant to represent a particular term structure.

2.3 Definition. Let t be a term in $\mathbb{T}$ and test in $TEST_{L^*}$
t satisfies test iff $t \in Close(test)$.
t could satisfy test if t does not satisfy test but some of the proper instances of t do.
t cannot satisfy test iff neither t nor any instance of t satisfies test.
ρ is a clean mgu of t and test if $t = test\rho$ and then $Dom(\rho) = Var(test)$ and $\mathbb{T} \supset Cod(\rho)$.

Unification among t and test can be used to determine when t satisfies test. We can state the following propositions:
1) t satisfies test iff t and test have a clean mgu, such a clean mgu is unique;

2) t could satisfy test iff t and test are unifiable but they have no clean unifier;
3) t cannot satisfy test iff t and test are not unifiable.

These three cases are mutually exclusive and decidable [BCF91c].
An mgu as in (2) is called a dirty mgu of t and test. It is dirty in the sense that it changes t. Note that tρ might not be in $\mathbb{T}$.

Examples.
- list(succ(X), nil) satisfies the test list(x, y) and a clean mgu for them is ρ={x/succ(X), y/nil}.
- list(Y, Y) satisfies list(x, x) too and a clean mgu for them is ρ={x/ Y}. Also σ = {Y/ x} is an mgu for them, but it is not clean.
- list(succ(X), Y) could satisfy list(x, x). A dirty mgu for them is ρ={x/succ(X), Y/succ(X)}.
- list(X, X) could satisfy list(x, succ(y)). A dirty mgu for them is ρ={x/succ(y), X/succ(y)}.
- list(o, X) cannot satisfy the test list(succ(o), x) since they are not unifiable.

The following proposition states some trivial consequences of definition 2.3.

2.5 Proposition. Let test ∈ $\mathbb{TEST}_{\mathbb{L}^*}$ and t ∈ $\mathbb{T}$.
1) t satisfies test iff Close(test) ⊇ Close(t).
2) t could satisfy test iff
   i) Close(test) ∩ Close(t) ≠ ∅; ii) Close(test) ⊉ Close(t).
3) t cannot satisfy test iff Close(test) ∩ Close(t) = ∅.

2.6 Definition. Two tests, $test_1$, $test_2$ ∈ $\mathbb{TEST}_{\mathbb{L}^*}$, are independent iff Close($test_1$) and Close($test_2$) are disjoint sets: Close($test_1$) ∩ Close($test_2$) = ∅.
A is a set of independent tests if any two tests in A are independent.
Let A, B be sets of tests. A and B are independent if whenever $test_a$ ∈ A and $test_b$ ∈ B, then $test_a$ and $test_b$ are independent.

2.7 Proposition.
1) $test_1$, $test_2$ ∈ $\mathbb{TEST}_{\mathbb{L}^*}$ are independent iff they cannot be unified in $\mathbb{T}^*$ even after renaming variables.
2) Let A be a set of independent tests and t be in $\mathbb{T}$. Three mutually exclusive cases are possible:
   i) t cannot satisfy any test in A;
   ii) t satisfies one test in A and it cannot satisfy the others;
   iii) t could satisfy one or more tests in A and it cannot satisfy the others.

Examples.

- The set A = {nil, o, list(x, succ(y)), list(o, nil)} is a set of independent tests.
- B = {succ(x), succ(succ(x))} is not a set of independent tests since the two tests can be unified in $\mathbb{T}^*$ after renaming one of them.
- succ(o) cannot satisfy any test in A.
- list(X, nil) satisfies list(o, nil) in A and it cannot satisfy the remaining tests in A.
- list(X, Y) could satisfy the last two tests in A, but it cannot satisfy nil or o.

3. Type schemata

In this section schemata describing structural properties of terms are introduced. They are expressed in the language defined in the previous section. The schemata have been restricted in order to insure simple definitions and at the same time maintaining the maximum of generality. Such schemata can be applied to terms of the object language in order to verify if they have the described structural properties. The verification is a rewriting process: each schema defines a way of traversing the terms and then it allows one to collect information during the traversal. When we apply a schema to a term three cases are possible:

a) the rewriting ends, in this case the term has the structure specified by the type schema and some useful information has been computed while traversing it;
b) the rewriting is not possible, in this case the term cannot have the specified structure;
c) the rewriting is suspended which means that the term could be instantiated so that the specified property might hold for the instance.

We distinguish two kinds of expressions on L*: type expressions and analysis expressions.

3.1 Definition. Let L* = <FUN, VAR, MVAR, TYPE, EVAL> be a term analyzer language and $\mathbb{T}$ = Term(FUN, VAR) the terms in the object language.

A type expression is defined inductively as follows:

i) if g is an evaluable function of arity 0 in EVAL, then g is a type expression;
ii) if A is a type function in TYPE and x is a metavariable in MVAR, then A(x) is a type expression;
iii) if g is an evaluable function of arity h in EVAL and $t_1, \ldots, t_h$ are type expressions, then $g(t_1, \ldots, t_h)$ is a type expression.

$\mathbb{T}\_EXP_{L^*}$ is the set of type expressions on L*.

An analysis expression can be obtained from a type expression by substituting each metavariable x in MVAR with a term t in $\mathbb{T}$.

$A\_EXP_{L^*}$ is the set of analysis expressions on L*.
An evaluable expression is an analysis expression containing no type symbol.
$E\_EXP_{L^*}$ is the set of evaluable expressions on L*.

Note that in type expressions type function symbols are applied to metavariables only and this is the only place where metavariables can occur. As a consequence, in analysis expressions the variables of the object language can appear only inside arguments of type function symbols and evaluable expressions are necessarily ground expressions.

3.2 Definition. Let L* be a term analyzer language.
A type schema, TS_{L^*}, on L* is a finite set of equalities in the following form:

A(test) = typexp

where A is in TYPE, test is in $TEST_{L^*}$, typexp is in $T\_EXP_{L^*}$ and each metavariable in typexp occurs only once and it occurs also in test.
Def(A) is the set of equalities associated to the type function symbol A in TS_{L^*}:

$A(test_1) = typexp_1, ..., A(test_k) = typexp_k$,

the set of tests associated to A in TS_{L^*} is $Test(A) = \{test_1, ..., test_k\}$ and
the type expression associated to $test_i$ is $typexp_i$.

In the following we assume that L* is fixed and so we simply write TEST, T_EXP, A_EXP, ... Moreover, in order to improve readability, we write type expressions and analysis expressions using infix notation for the functions in EVAL and assume properties (commutativity, associativity, ...) of the intended interpretation of these functors when this interpretation is clear.

Examples. Let FUN = {o, nil, empty, succ(_), list(_, _), tree(_, _, _)}.

1) $TYPE_1$= {Nat, Lon}. $EVAL_1$ = {ntrue, ltrue, ∧(_, _)}.

TS_1: {Nat = natural number, Lon = list of naturals}

Nat(o) = ntrue
Nat(succ(x)) = Nat(x)
Lon(nil) = ltrue
Lon(list(x, y)) = Nat(x) ∧ Lon(y).

2) TYPE = {Nat, Lon, Nln, Stree}. $EVAL_2$ = {0, 1, +(_, _)}.

TS_2: {Nat = natural number, Lon = list of naturals, Nln = nested list of naturals, Stree = symmetric tree}

Nat(o) = 0
Nat(succ(x)) = 1 + Nat(x)
Nln(nil) = 0
Nln(list(x, y)) = 1+ Nat(x) + Nln(y)
Nln(list(x, y)) = 1+ Nln(x) + Nln(y)
Lon(nil) = 0
Lon(list(x, y)) = 1 + Nat(x) + Lon(y).
Stree(empty) = 0
Stree(tree(x, y, y)) = 1+ Stree(y).

3.3 Definition. Let TS be a type schema.
The set of not empty types in TS, NEMPTY(TS), is recursively defined as the closure

NEMPTY(TS) = {A | A ∈ TYPE and there exists an equality A(test) = typexp in TS such that for all B in typexp, B ∈ NEMPTY(TS)}.

It is reasonable to require from a type schema to have a correct recursive definition, namely it must actually define something and it must give a unique definition. In order to express the second property, we need to define the concept of "separate" types. The intuition is the following: if two types are separate, a term cannot belong to both of them.

3.4 Definition. Let A, B, C, D be type function symbols in a type schema TS. Let x ∈ MVAR.
A and B are separate in TS iff every pair of equalities in Def(A) × Def(B) is separable.
A pair of equalities, $A(test_1) = typexp_1$ and $B(test_2) = typexp_2$, in TS is separable iff
i) either $test_1$ and $test_2$ are independent or
ii) $test_1$ and $test_2$ are equal and for some variable x there exist a subexpression C(x) of $typexp_1$ and a subexpression D(x) of $typexp_2$, such that C and D are separate in TS.

Examples. In TS_2, Test(Nat) and Test(Nln) are independent, hence Nat and Nln are separate. On the contrary, Lon and Nln are not separate since Test(Lon) and Test(Nln) are not independent and Lon(nil) = 0 and Nln(nil) = 0, thus condition (ii) is not satisfied.

3.5 Definition. A type schema TS is well formed iff
1) TYPE = NEMPTY(TS);
2) for every A ∈ TYPE, every pair of (distinct) equalities in Def(A) are separable.
The first condition is meant to ensure a basis for each recursive type definition. The second condition is meant to guarantee a unique result when applying the type schema to a term.

Examples. TS_1 and TS_2 are well formed schemata. Regarding condition (2), the only not trivial case is given by the two recursive definitions of Nln in TS_2. list(x, y) is a common test while Nat and Nln are separate.

A well formed type schema represents sets of terms in the object language which have the structures specified by the schema itself. Such sets correspond to the closures of the equations associated to the type functions with respect to instantiation in T.

3.6 Definition. Let A be a type function symbol in a well formed type schema TS and let t be a term in T.

1) t belongs to A in TS if
   i) in TS there exists A(test(x_1, ..., x_m)) = typexp, such that t satisfies test: t = testρ, with ρ = {x_1/ s_1, ..., x_m/ s_m}, and
   ii) for all subexpressions B(s_j) of typexpρ s_j belongs to B.
2) t cannot belong to A in TS if
   either for all test ∈ Test(A), t cannot satisfy test,
   or there exists test(x_1, ..., x_m) ∈ Test(A) such that
   t satisfies test: t = testρ, ρ = {x_1/ s_1, ..., x_m/ s_m}, and
   for every equation A(test(x_1, ..., x_m)) = typexp in Def(A),
   there exists a subexpressions B(s_j) of typexpρ such that s_j cannot belong to B.
3) t could belong to A in TS if neither (1) nor (2) holds.

From definitions 3.4, 3.5 and 3.6 we have the following proposition.

3.7 Proposition. Let TS be a well formed type schema.
1) For any type function A in TYPE, at least one term belongs to A.
2) For any term t in T, one and only one of the three cases of definition 3.6 holds.
3) Let A and B be two separate type function symbols, then
   if t belongs to A, then t cannot belong to B.

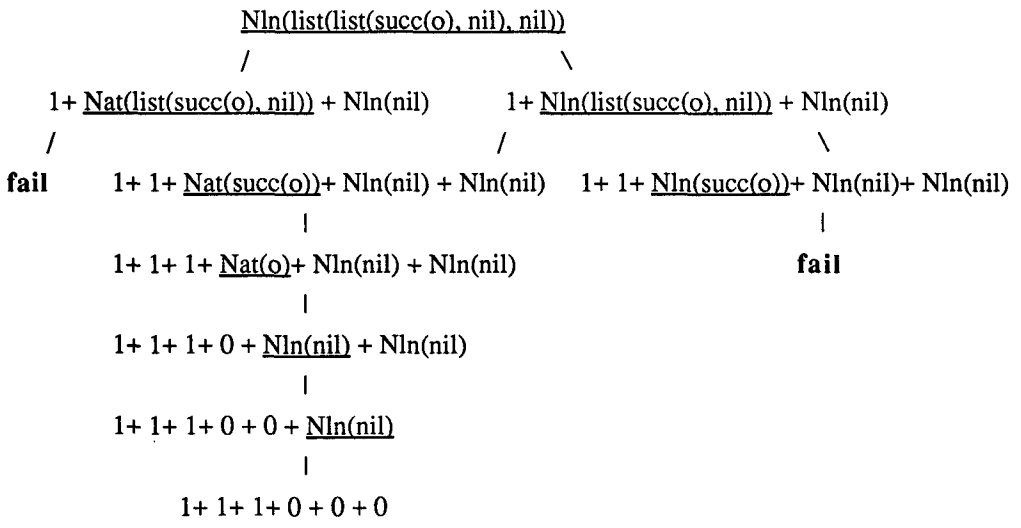
The rewriting process described by a well formed type schema can be represented by a derivation tree. Building such a tree for a type function applied to a term corresponds to verifying if the term belongs to the type.

3.8 Definition. Let TS be a well formed type schema, let exp be an analysis expression. The derivation tree of exp in TS, TREE(exp), is a tree whose nodes are labelled either by an analysis expression or by **fail**. TREE(exp) is built up in the following way:

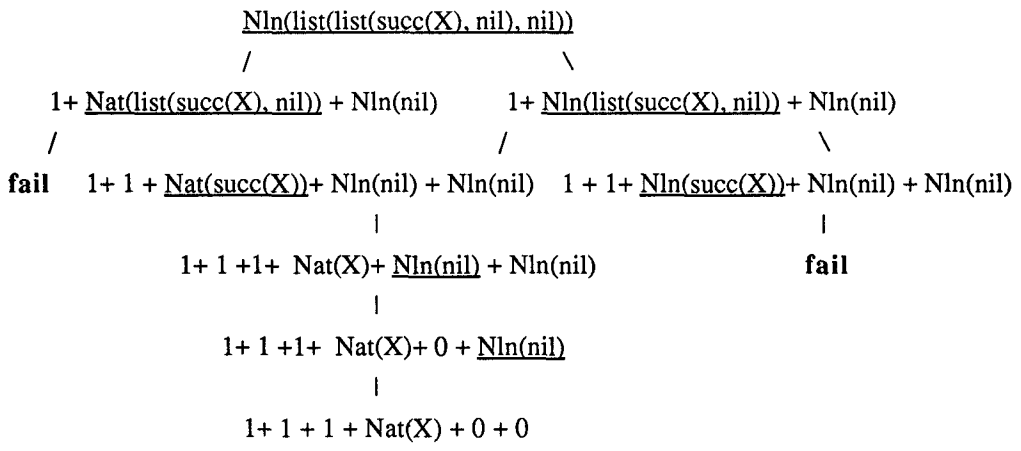
label the root with exp;
repeat
 choose and mark a not marked leaf node, exp';
 if exp' has a subexpression B(s) such that s does not unify with any test in Test(B)
  then add a single marked leaf: **fail**
  else **if** exp' has at least one subexpression B(s) such that s satisfies
   test$_1$ = ... = test$_n$ in Test(B) with the clean mgu ρ,
   then choose the leftmost B(s);
    add n leaves to exp': exp'σ_j,
    where σ_j = [B(s)/typexp$_j$ ρ], with $1 \leq j \leq n$,
until all the leaves have been marked.

Examples.

1) Let us consider the type schema $\mathbb{TS}_2$ and let us build the derivation tree of Nln(list(list(succ(o), nil), nil)).



2) Let us consider again the type schema $\mathbb{TS}_2$ and the derivation tree of Nln(list(list(succ(X), nil), nil)).



3.9 Lemma. Let $\mathbb{TS}$ be a well formed type schema. For every exp in $\mathbb{A\_EXP}$, TREE(exp) is finite.

The lemma is a consequence of the fact that in the left side of the equations in $\mathbb{TS}$ a type can never be applied to a metavariable; therefore each equation corresponds to term decomposition. A complete proof is given in [BCF91c].

3.10 Lemma. Let $\mathbb{TS}$ be a well formed type schema and let exp be in A_EXP.
If exp contains at least one type function symbol, then one of the following three exclusive cases holds:

1) (failed tree) all the leaves in TREE(exp) are labelled by **fail** and there exists at least one subexpression of exp, B(s), such that s cannot belong to B;
2) (success tree) all the leaves in TREE(exp) are labelled by **fail** except one which is labelled by an evaluable expression (ground and containing no type function symbols) and for any subexpression of exp, B(s), s belongs to B;
3) (expansible tree) the leaves in TREE(exp) are partitioned into two sets:
   i) a possibly empty set of leaves labelled by **fail**;
   ii) a not empty set of leaves labelled by expressions which contain only subexpressions B(s) such that s could belong to B.

The proof is by induction on the depth of TREE(exp) and it is given in [BCF91c].

3.11 Corollary. Let $\mathbb{TS}$ be a well formed type schema, A be in TYPE and t in $\mathbb{T}$.

1) TREE(A(t)) is failed iff t cannot belong to A;
2) TREE(A(t)) is successful iff t belongs to A;
3) TREE(A(t)) is expansible iff t could belong to A.

3.12 Proposition. Let $\mathbb{TS}$ be a well formed type schema, A be in TYPE and t in $\mathbb{T}$.

1) If t belongs to A, then for every substitution σ such that $\mathbb{T} \supset Cod(\sigma)$, $t\sigma$ belongs to A and TREE(A($t\sigma$)) has the same evaluable leaf as TREE(A(t)).
2) If t cannot belong to A, then for every substitution σ such that $\mathbb{T} \supset Cod(\sigma)$, $t\sigma$ cannot belong to A.

The proof is given in [BCF91c] by induction on the complexity of the term t.

Both belonging to a type and the impossibility to belong to it are properties which are invariant wrt substitution. Moreover, by Lemma 3.10 and its Corollary, it is possible to transfer information about the structure from a term to its subterms and viceversa. These are exactly the properties we need for verifying universal termination of logic programs with our method, as we will see in the next section.

4. Typed Norms

The evaluable expressions we defined in section 3 can be interpreted. Their interpretation depends on the analysis we want to perform on terms. A particular interpretation is determined by giving a semantic domain in order to associate a meaning to the evaluable function symbols already introduced. We need to distinguish three cases: when a term has the required structural property and hence it can be rewritten and interpreted, when it cannot have the required structural property and when it could have it, if properly instantiated. In order to represent the last two cases we introduce two special semantic values: **false** and **maybe**.

4.1 Definition. Let $\mathbb{I}$ be a not empty set and EVAL be a set of evaluable function symbols. An interpretation of EVAL in $\mathbb{I}$, [ ], is defined in this way:

1) $[f] = c \in \mathbb{I}$, if f has arity 0;

2) $[f] = f_{\mathbb{I}} : \mathbb{I}^n \rightarrow \mathbb{I}$, if f has arity n>0.

$\mathbb{I}$ is a semantic domain.

The interpretation can be naturally extended to all evaluable expressions:

$[f(t_1, \ldots, t_h)] = f_{\mathbb{I}}([t_1], \ldots, [t_h])$.

4.2 Definition. Let $\mathbb{TS}$ be a type schema, $\mathbb{I}$ a not empty set and [ ] an interpretation of EVAL in $\mathbb{I}$. Let $\mathbb{T}$ be the set of terms in the object language and A be in TYPE.

The typed norm $|\ |_{\mathbb{I},A}$ associated to A by $\mathbb{TS}$ and [ ], is the function

$$|\ |_{\mathbb{I},A} : \mathbb{T} \rightarrow \mathbb{I} \cup \{\textbf{maybe}, \textbf{false}\}$$

defined in the following way:

1) $|t|_{\mathbb{I},A} = \textbf{false}$ iff t cannot belong to A;

2) $|t|_{\mathbb{I},A} = \textbf{maybe}$ iff t could belong to A;

3) $|t|_{\mathbb{I},A} = [exp] \in \mathbb{I}$ iff t belongs to A and exp is the label of the only evaluable leaf in TREE(A(t)).

Example. Let us consider $\mathbb{TS}_1$ with the interpretation

1) $\mathbb{I} = \{\textbf{true}\}$ and [ntrue] = **true**, [ltrue] = **true**, [$\wedge$] = $\wedge$ (the usual logical "and"). It induces the typed norms $|\ |_{\mathbb{I},Nat}$ and $|\ |_{\mathbb{I},Lon}$ such that

$|list(succ(o), nil)|_{\mathbb{I},Lon}$ = **true**. In fact TREE(Lon(list(succ(o), nil))) is successful and its only evaluable leaf is (ntrue $\wedge$ ltrue);

$|list(succ(o), X)|_{\mathbb{I},Lon}$ = **maybe**. In fact TREE(Lon(list(succ(o), X))) is expansible. Its only suspended leaf is (ntrue $\wedge$ Lon(X));

Such norms determine if a term is either a natural number or a list containing only natural numbers.

2) $\mathbb{I} = \mathbf{N}$ and [ltrue] = **0**, [ntrue] = 1, [$\wedge$] = +, where + is the usual sum of natural numbers. It induces the typed norms $|\ |_{\mathbb{I},Nat}$ and $|\ |_{\mathbb{I},Lon}$ such that
$|list(o, list(succ(o), list(succ(succ(o)), nil)))|_{\mathbb{I},Nat} = 3$. In fact TREE(Lon(list(o, list(succ(o), list(succ(succ(o)), nil)))) is successful. Its only evaluable leaf is (1+1+1+0).
$|list(o, X)|_{\mathbb{I},Nat}$ = **maybe**. In fact TREE(Lon(list(o, X))) is expansible. Its only suspended leaf is (1+Lon(X)).
Such norms determine the length of a list of natural numbers.

Let A be a type function in a well formed type schema $\mathbb{TS}$ and $\mathbb{I}$ a not empty set. Let t be in $\mathbb{T}$ and $|\ |_{\mathbb{I},A}$ be a typed norm. From 4.2 and the properties of TREE(A(t)) stated in section 3, we have that:

1) for every substitution σ such that $\mathbb{T} \supset Cod(\sigma)$,
   i) if $|t|_{\mathbb{I},A} \in \mathbb{I}$, then $|t\sigma|_{\mathbb{I},A} = |t|_{\mathbb{I},A}$;
   ii) if $|t|_{\mathbb{I},A}$ = **false**, then $|t\sigma|_{\mathbb{I},A}$ = **false**;
2) $|t|_{\mathbb{I},A} \in \mathbb{I}$ iff
   there is one and only one equation in $\mathbb{TS}$, $A(test) = typexp(B_1(x_1), ..., B_n(x_n))$, such that
   i) t satisfies test, t=testρ;
   ii) $|t|_{\mathbb{I},A} = typexp(|x_1\rho|_{\mathbb{I},B1}, ..., |x_n\rho|_{\mathbb{I},Bn})$;
3) $|\ |_{\mathbb{I},A}$ is a total function;
4) there exists $t \in \mathbb{T}$ such that $|t|_{\mathbb{I},A} \in \mathbb{I}$.

Typed norms include semi-linear norms [BCF91a, BCF91b]. Let |...| be a semi-linear norm on T defined as follow:

for all functions f in FUN,
$|f(t_1, ..., t_n)| = c_f + |t_{i1}| + ... + |t_{im}|$, where $c_f \in \mathbf{N}$, and $\{1, ..., n\} \supseteq \{i_1, ..., i_m\}$.

Let TYPE = {Semil} and EVAL = $\{\underline{c}_f \mid f \text{ in FUN}\} \cup \{+(\_, \_)\}$. The type schema, $\mathbb{TS}$, corresponding to |...| is defined as follows:

1) for all functions f in FUN the schema contains an equality
   $Semilin(f(x_1, ..., x_n)) = \underline{c}_f + Semilin(x_{i1}) + ... + Semilin(x_{im})$,
   where $x_1, ..., x_n$, are distinct variables;
2) there are not other equalities.

It is clear that, for all t in $\mathbb{T}$, TREE(Semilin(t)) has no branching.

4.3 Proposition. Let |...| be a semi-linear norm on $\mathbb{T}$, $\mathbb{TS}$ the corresponding type schema and [] an interpretation in **N** given by $[\underline{c}_f] = c_f$ and $[+(\_, \_)] = +$. Let $|\ |_{N,Semilin}$ be the typed norm induced by $\mathbb{TS}$.
For every t in $\mathbb{T}$:

1) $|t| = n$ and rigid(t) $\Leftrightarrow$ $|t|_{N,Semilin} = n$.

2) $|t| = n$ and not-rigid(t) $\Leftrightarrow$ $|t|_{N,Semilin}$ = **maybe**.

$VREL_{|\ |}(t)$ are the variables in the label of the only leaf node in TREE(Semilin(t)).

5. Applications

In this section we give a few examples of universal termination proofs for logic programs, which are not feasible by using semi-linear norms and which are feasible by using typed norms. The verification methodology is the one we defined in [BCF91a, BCF91b], hence the proof is still reasonably simple. The only novelty is the possibility of using typed norms to analyze also programs which traverse the terms in a complex way. The class of goals for which universal termination is ensured does still include not ground goals, only the part of the terms which is traversed by the program need to be fixed. We briefly recall our verification technique in the first example, the other verification examples are only sketched for brevity. From now on we adopt the notation for lists which is usual in logic programming and we use underlined symbols, such as $\underline{x}$ or $\underline{t}$, to denote tuples. Moreover, for simplicity's sake, we shall often write A(t) instead of "t belongs to A" and $\neg$A(t) instead of "t cannot belong to A".

1) Let us consider the following program.

```
P1:  1:  check([X| Xs]) :- check(Xs).
     2:  check([X]) :- nat(X).
     3:  nat(s(X)) :-  nat(X).
     4:  nat(o).
```

check(t) holds if t is a not empty list with a natural number as its last element.

If t is a list with a variable as tail or if it is a list of fixed length whose last element is a variable X or $s^n(X)$, then the program loops. We would like to prove that the class of goals described by {:- p(t).}, where t is a non-empty list whose last element belongs to **N**, universally terminates in $\mathbf{P_1}$. A semi-linear norm cannot be used since it cannot distinguish the last element of a list from the other ones. But we can very naturally define a typed norm for this purpose. In $\mathbf{P_1}$, FUN = {o, s(_), [], [_ | _]} is the set of program language functors.

Let TYPE = {Length, Nat, Empty} and EVAL = {0, 1, +(_, _)} and consider the following type schema:

TS: Lastn([x| y]) = 1 + Lastn(y) Nat(s(x)) = 1 + Nat(x)

Lastn([x| y]) = Nat(x) + Empty(y) Nat(o) = 0

Empty([]) = 0

The type schema is well formed. Consider the interpretation $\{|\ |_{N,Lastn}, |\ |_{N,Nat}, |\ |_{N,Empty}\}$ of TS determined by the usual interpretation of EVAL in **N**.

$|\ t\ |_{N,Lastn} \in \mathbf{N}$ means that that t belongs to Lastn. The description of the class of goals we are considering becomes: {:- check(t).; Lastn(t)}.

Our termination proof technique [BCF91a, BCF91b] is similar to the one used in procedural programming. When considering a program without mutual recursion, the general method can be simplified and it consists in

(a) associating to each predicate symbol p in the program *a pre/post specification* {Pre($\underline{x}$)} p($\underline{x}$) {Post($\underline{x}$)}, *well-behaved with respect to substitutions*, in order to state terms properties which can be useful for proving termination. Well-behaved with respect to substitutions means that if an instance p($\underline{t}$) of p satisfies its precondition (postcondition) then every further instantiation p($\underline{t}\sigma$) also satisfies it;

(b) proving the correctness of such a specification. Following the criterion proposed in [BC89, BCF91b] this can be done by proving, for each clause $a_0(\underline{t}_0)$:- $a_1(\underline{t}_1), \ldots, a_n(\underline{t}_n)$. in the program, that the following two conditions are satisfied:
   1) $\forall \underline{x}.\ (Pre_0(\underline{t}_0) \wedge (\wedge_{i=1}^{k-1} Post_i(\underline{t}_i))) \rightarrow Pre_k(\underline{t}_k)$, for all k in 1, ..., n;
   2) $\forall \underline{x}.\ (Pre_0(\underline{t}_0) \wedge (\wedge_{i=1}^{n} Post_i(\underline{t}_i))) \rightarrow Post_0(\underline{t}_0)$;

(c) finding *an ordering function* f, *not increasing with respect to substitutions*, which maps the calling instances of each recursive predicate into **N**. A calling instance of a predicate p is an invocation of p in the computation. Not increasing with respect to substitutions means that if p($\underline{t}$) is a calling instance of p, then for every substitution σ: $f(p(\underline{t}\sigma)) \leq f(p(\underline{t}))$.

(d) proving, for every recursive clause $a_0(\underline{t}_0)$:- $a_1(\underline{t}_1), \ldots, a_m(\underline{t}_m)$. and every $a_k(\underline{t}_k)$, which is a recursive call of $a_0(\underline{t}_0)$, that the following condition holds:
   $\forall \underline{x}.\ (Pre_0(\underline{t}_0) \wedge (\wedge_{j=1}^{k-1} Post_j(\underline{t}_j)) \rightarrow (f(a_k(t_k)) < f(a_0(t_0)))$,
   namely the information given in the specification ensures that the value of the ordering function on the head is greater than the value on the recursive call.

Steps (b) and (d) of the method can be handled in a simple way since the pre/post specifications and the ordering functions "well-behave" with respect to substitutions and this allows us to ignore the actual computation (that is real unification) and to reason at a syntactic level. For further details see [BCF91a, BCF91b].

We now apply the technique to our example. Due to space limitations, we describe only points (a) and (c). The proofs required in points (b) and (d) are rather simple.

a) The pre/post specification we associate to each predicate is
   {Lastn(t) $\vee$ Empty(t)} check(t) {true}
   {Nat(t)} nat(t) {true}.

The specification is rather trivial since the clauses have no local variables and moreover each predicate has arity one. Hence no relation among terms need to be known for proving

termination. The meaning of the specification is the following: the predicate check is called with an argument in Lastn and nat is called with an argument in Nat. Such a specification is clearly well-behaved wrt substitutions since it deals only with the structure of input terms.

c) The ordering function is intended to measure the part of the term which is traversed by the program. As an ordering function we can associate the weight of the input terms defined by the typed norm induced by TS:

f:	check(t)	→	$\lvert t \rvert_{N,Lastn}$	if Lastn(t)
			$\lvert t \rvert_{N,Empty}$	if Empty(t)
			0	otherwise
	nat(t)	→	$\lvert t \rvert_{N,Nat}$	if Nat(t)
			0	otherwise

The ordering function is not increasing wrt substitutions. In fact t belongs to the type at every invocation of the predicate and then the weight of the input term t does not change by instantiating it as shown in section 4.

2) Let us now consider the following program.

P_2:
```
1:  change([], []).
2:  change([X, Y| Xs], [Z, Y| Zs]) :- q(X, Z), p(Y), change(Xs, Zs).
3:  q([], []).
4:  q([X| Xs], [a| Zs]) :- q(Xs, Zs).
5:  p([]).
6:  p([X| Xs]) :- nat(X), p(Xs).
7:  nat(0).
8:  nat(s(X)) :- nat(X).
```

The predicate change(t_1, t_2) holds when

i) t_1 is a list of lists whose length is even and its elements in even positions are lists of natural numbers;

ii) t_2 is equal to t_1 but with the elements in odd positions substituted by lists of "a" of the same length of the original elements.

Let us consider for example the class of goals {:- change(t_1, t_2).}, where t_1 is a finite list whose length is even (no variables in tail position) and whose elements in even positions are lists of natural numbers while the elements in odd positions are finite lists. We want to prove that all the goals in the class universally terminate in **P_2**. Semi-linear norms cannot be used in this example too. Since we need to associate different measures to the same functor list. Element lists in odd positions should have a weight corresponding to their length and elements lists in even positions should be weighted by the sum of the natural numbers which are their elements. Moreover the list of lists, given in input to the predicate change, should have a weight given by

the sum of the weights of its elements. This corresponds to consider the following type schema with the standard interpretation on N.

TYPE = {N, Length, Sum, Nat} EVAL = {0, 1, +}

TS: N([]) = 0 Length([]) = 0

N([x, y| z]) = Length(x) + Sum(y) + N(z) Length([x| y]) = 1 + Length(y)

Sum([]) = 0 Nat(o) = 0

Sum([x| y]) = Nat(x) + Sum(y) Nat(s(x)) = 1 + Nat(x)

The description of the class of goals we are interested in becomes: {:- change(t_1, t_2).; N(t_1)}. The termination proof then follows the usual path.

3) As a last example, let us consider the following program:

split([b| Xs], [b| Ys], Zs) :- split(Xs, Ys, Zs).

split([a| Xs], Ys, [a| Zs]) :- split(Xs, Ys, Zs).

split([], [], []).

split(t_1, t_2, t_3) is true if t_1 is a list of b's and a's, t_2 a list of b's, t_3 a list of a's and the number of a's in t_1 is equal to the length of t_3, while the number of b's is equal to the length of t_2.

Let us consider the following type schema (with its obvious interpretation):

Finitel([]) = 0

Finitel([x| y]) = 1 + Finitel(y)

and the specification

{Finitel(t_1)} split(t_1, t_2, t_3) {$|t_1|_{N,Finitel} = |t_2|_{N,Finitel} + |t_3|_{N,Finitel}$}.

With this specification, we can easily prove that all the goals in the class {:- split(t_1, t_2, t_3).; Finitel(t_1)} universally terminate. But in this termination proof we do not distinguish between termination due to a finite failure of the computation, for example when the first list does not contain only a's and b's, and successful termination. This distinction can be sometimes very useful or even necessary. To this purpose we can further specialize both the specification and the typed norms in this way:

Alist([]) = 0 Blist([]) = 0

Alist([a| y]) = 1 + Alist(y) Blist([b| y]) = 1 + Blist(y)

ABlist([]) = 0

ABlist([b| y]) = 1 + ABlist(y)

ABlist([a| y]) = 1 + ABlist(y)

and

{ABlist(t_1)} split(t_1, t_2, t_3) {$|t_1|_{N,ABlist} = |t_2|_{N,Blist} + |t_3|_{N,Alist}$}.

The information in the postcondition can help us to detect some goals which terminate by finite failure. For example the goals:

1) split([a, b, a, a], [a, b], [a, a]) 2) split([a, c, a, a], X, Y).

3) split([X, Y, Z], [X], [Z, c]) 4) split([a, a, b], [b, b], [a, a])

all terminate by finite failure. They are in the class we are interested in, but they all have a property, invariant wrt substitutions, which falsifies the postcondition. We can prove that the program is correct wrt the specification and then, when split is invoked in a way that satisfies the precondition and it terminates successfully, its postcondition must hold. Hence all the previous goals must fail. In our examples:

1) $|\ t_2\ |_{N,Blist}$ = **false**; 2) $|\ t_1\ |_{N,ABlist}$ = **false**;

3) $|\ t_3\ |_{N,Alist}$ = **false**; 4) $|\ t_1\ |_{N,ABlist} \neq |\ t_2\ |_{N,Blist} + |\ t_3\ |_{N,Alist}$.

In this way we can identify a class of finite failing goals: the ones which satisfy the preconditions and falsify (in a three valued logic {**true**, **maybe**, **false**}) the postcondition. Note that we cannot say anything about failure or success of other goals such as:
split([a, b, a, b], [b, b], [a, a]); split([a, b, a, a], X, Y); split([X, Y, Z], [X, a], [Z]);
split([X, Y, Z], [X, Y], [Z]); split([X, X, X, Z], [b, b], [a, a]).

6. Conclusions

In this paper we defined a language for describing *type schemata* and studied a class of functions, the *typed norms*, which can be associated to such schemata by means of an interpretation in a semantic domain. Typed norms can check if a term has the structure described by the schema. There are three possibilities: the term has the required structure, the term cannot have the required structure and the term could be instantiated in a way to have such a structure. In the first case, when there is a positive answer, the typed norm can associate a value in the semantic domain to the term. Both having the required structure and the impossibility to gain it, are *well behaved properties* for a term, that is properties which cannot be falsified by applying further substitutions to the term. Typed norms still have all the interesting characteristics of *semi-linear norms*, we studied in [BCF91a, BCF91b] for verifying universal termination of pure Prolog programs. With typed norms we can then apply our simple verification method to a much larger class of programs. A few examples of such verifications, which were not possible by using only semi-linear norms, are also given.

We are also considering different fields of application for typed norms and pre/post specifications. One could be a simple (when compared with less abstract ones, such as in [CM91]) characterization of unifiability. This seems to be fundamental for analyzing more complex properties of logic programs such as existential termination (the existence of one finite successful derivation), finite failure and for distinguishing successful and failing computations in universal termination. Also some type checking could be feasible by using these norms. For example let us consider the type schema $\mathbb{TS}_1$ defined in the examples, the usual interpretation on **N** and let us take the simple program:

sum(o, X, X).

sum(s(X), Y, s(Z)) :- sum(X, Y, Z).

This program correctly defines the sum over natural numbers if we know that its domain of application is restricted to triples of natural numbers. In fact only the first term in sum(t_1, t_2, t_3) is strictly typed by the program. By using preconditions, we can express the application domain and then verify that the typing is correct with respect to such a domain and to the meaning we intend for the program. In this example we can use the well-behaved specification

{Nat(t_1), Nat(t_2))} sum(t_1, t_2, t_3) {Nat(t_3)},

and our simple inductive method [BC89, BCF91b] and verify that the program is correct with respect to the specification. This corresponds to a type checking.

Our work is strongly related to many other works dealing with the verification or the synthesis of program properties. On one hand there are the works on logic program termination such as [VP86, Bez89, AP90, AP91] which characterize classes of programs with interesting termination properties, or [UG88, Plü90a, Plü90b, VS91] where automatic termination proofs are based on systems of inequalities among term sizes, or [FG85, Bau88, WS89, WS91, AP90, AP91, Dev90, BCF91a, BCF91b, CM91] where more general techniques for verifying termination are proposed. On the other hand there are the works on Abstract Interpretation for moding and for groundness analysis and for type checking and type inference [BJCD87, Deb89, JB90, MS90, FS90, CFW91]. The invariance of typed norms with respect to substitutions corresponds to the requirement of closure under instantiation in Abstract Interpretation, this allows us to hope for a partial automatization of the verification process.

References

[AP90] Apt K.R., Pedreschi D., Studies in Pure Prolog: Termination, in *Proceedings Symposium on Computational Logic*, J. W. Lloyd Ed., Basic Research Series 1, Springer-Verlag (1990), 150-176.

[Apt90] Apt K.R., Introduction to Logic Programming, in *Handbook of Theoretical Computer Science*, J. van Leeuwen Ed., Elsevier Science Publishers 1990.

[AP91] Apt K.R., Pedreschi D., Proving Termination of General Prolog Programs. Technical Report, CWI, Amsterdam, 1991.

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

[Bez89] Bezem M., Characterizing Termination of Logic Programs, in *Proceedings NACLP'89*, E. L. Lusk, R. A. Overbeek, Eds., The MIT Press (1989), 69-80.

[BC89] Bossi A., Cocco N., Verifying Correctness of Logic Programs, in *Proceedings TAPSOFT'89*, Vol. 2, J. Diaz, F. Orejas, Eds., LNCS 352, Springer-Verlag, (1989), 96-110.

[BCF91a] Bossi A., Cocco N., Fabris M., Proving Termination of Logic Programs by Exploiting Term Properties, in *Proceedings CCPSD-TAPSOFT '91*, S. Abramsky, T.S.E. Maibaum, Eds., LNCS 494, Springer-Verlag, (1991), 153-180.

[BCF91b] Bossi A., Cocco N., Fabris M., Norms on terms and their use in proving universal termination of a logic program, CNR Technical Report "Progetto Finalizzato Sistemi Informatici e Calcolo Parallelo", n. 4/29 (March 1991).

[BCF91c] Bossi A., Cocco N., Fabris M., Typed Norms for Logic Programs. Technical Report Dip. Matematica Pura e Applicata, Università di Padova, Italy, (December 1991).

[BJCD87] Bruynooghe M., Janssens G., Callebaut A., Demoen B., Abstract Interpretation: towards the global optimization of Prolog programs, In *Proceedings Symp. on Logic Programmming*, IEEE Society Press, (1987), 192-204.

[CFW91] Cortesi A., Filè G. Winsborough W., Prop Revisited: Propositional Formula as Abstract Domain for Groundness Analysis. In *Proceedings of LICS 91.* Amsterdam, The Netherlands (July 1991), IEEE Computer Society Press, 322-327.

[CM91] Colussi L., Marchiori E, Proving Correctness of Logic Programs Using Axiomatic Semantics, in *Proceedings ICLP'91*, K. Furukawa, Ed., The MIT Press (1991) 629-642.

[Deb89] Debray S., Static Inference of Modes and data Dependencies in Logic Programs, *ACM Trans. on Programming Languages and Systems 11*, No. 3, (1989), 418-450.

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

[FGKP85] Francez N., Grumberg O., Katz S., Pnueli A., Proving Termination of Prolog Programs. In *Logics of Programs* LNCS 193, Springer-Verlag, 1985, 89-105.

[FS91] Filè G. and Sottero P., Abstract Interpretation for Type Checking. In *Procceding of PLILP 91*, J. Maluszynski and M. Wirsing (eds), LNCS 528, Springer-Verlag, (1991), 311-322.

[JB90] Janssens G., Bruynooghe M., Deriving Descriptions of Possible Values of Program Variables by means of Abstract Interpretation, Technical Report CW 107, Dept. of Computer Science, K.U. Leuven, (March 1990), to appear in Journal of Logic Programming.

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

[MS90] Marriott K., Søndergaard H., Abstract Interpretation of Logic Programs: the Denotational Approach. In *Proceedings GULP '90,* A. Bossi (ed.), Padova (June 1990), 399-425.

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

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

[UG88] Ullman J.D., Van Gelder A., Efficient Tests for Top-Down Termination of Logical Rules, *JACM 35*, No. 2, (1988), 345-373.

[VP86] Vasak T., Potter J., Characterisation of Terminating Logic Programs, in *Proceedings Int. Symposium on Logic Programming '86*, IEEE, (1986) 140-147.

[VS91] Verschaetse K., De Schreye D., Deriving Termination Proofs for Logic Programs Using Abstract Procedures. In *Proceedings ICLP'91*, Paris, June 1991. The MIT Press, 301-315.

[WS89] Wang B., Shryamasunder R.K., Proving Termination of Logic Programs, In *Perspective in Theoretical Computer Science*, Commemorative Volume, Ed. R. Narasimhan, World Scientific Publishers, Singapore, 380-397 (1989).

[WS91] Wang B., Shryamasunder R.K., Methodology for Proving the Termination of Logic Programs, In *Proceedings STACS'91.* Hamburg, Germany, February 1991.