When Size Does Matter
Termination Analysis for Typed Logic Programs (Extended Abstract)

Wim Vanhoof and Maurice Bruynooghe

Department of Computer Science, K.U.Leuven, Belgium
{wimvh,maurice}@cs.kuleuven.ac.be

Abstract. This extended abstract describes work in progress on termination analysis of strongly typed logic programs. A basic difference with existing work on the topic is that in our approach we use the type of a term to derive a number of basic measures – rather than a single sophisticated measure – by which the size of a term can be measured. The basic idea is to characterise the size of a term by counting the number of subterms of a particular type it has. This characterisation allows to compare the sizes of terms of different but related types and to construct size relations between such terms in a straightforward way. Precisely the construction of such size relations has been a major point of difficulty in other type-based approaches.

1 Introduction and Motivation

Within the context of logic programming, a lot of attention has been paid to automatic termination analysis (see e.g. [12] for an overview). Roughly said, a logic program terminates with respect to a query of interest Q if it can be shown that in any derivation for Q the "size" – measured over some well-founded domain – of two consecutive calls to a same predicate decreases. In what follows, we restrict our attention to directly recursive definite programs, and consider left-to-right termination of such a program with respect to a set of *atomic* queries. To characterise the set of calls that arise during a derivation, one can introduce the concept of a *call set* [7]: given a program P and a set S of atomic queries, the *callset*, denoted $call(P, S)$, is the set of all atoms A such that a variant of A is a selected atom under the left-to-right selection rule in some derivation for (P, Q), for some $Q \in S$. To measure terms and atoms, one usually employs the notions of *norm*, which maps a term to a natural number, and *level mapping* which maps an atom to a natural number. Examples of well-known norms are the *listlength* norm, which counts the number of elements in a list, and the *termsize* norm, which counts the number of functors in a term. In [7] it is shown that a directly recursive program P terminates under the left-to-right selection rule for all atomic queries in a set S if and only if P is *acceptable* with respect to S:

Definition 1. *Let S be a set of atomic queries and P a definite directly recursive program. P is* acceptable *with respect to S if there exists a level mapping $|.|$ such that*

- $\forall A \in call(P,S)$
- $\forall H \leftarrow B_1,\ldots,B_n \in P$ *such that* $\theta = mgu(H,A)$ *exists*
- *for any atom* B_i *having the same predicate symbol as* A *and for any computed answer substitution* θ' *for* $(B_1,\ldots,B_{i-1})\theta$*:*

$$|A| > |B_i\theta\theta'|$$

In practice, the callset $call(P,S)$ can be approximated by a set of abstract calls, that represent which of a call's arguments are *instantiated enough* with respect to a particular norm [5]. A term t is said to be instantiated enough with respect to a norm $\|.\|$ if and only if $\|t\| = \|t\theta\|$ for every substitution θ (this corresponds with the norm being rigid on t). Likewise, the computed answer substitution for the intermediate atoms $B_1,\ldots,B_i$ in Definition 1 can be abstracted by the *interargument relations* (or *size relations*) of the involved atoms. Informally, a size relation is a relation that holds between the norms of the arguments of atoms in the success set of the program. When abstraction is employed, termination of a program is guaranteed if a strict decrease can be shown between the head H and each recursive body atom B_i of every clause using only the size relations of the intermediate atoms and a level mapping that only takes arguments into account which are guaranteed to be instantiated enough for each call.

Whether or not termination can be proven often depends on the particular norm that is chosen. Some techniques for termination analysis use a specific norm [15], while others provide the user with the possibility of specifying a norm that might be tailored to the program to be analysed [10]. In any case, fixing a particular norm imposes a number of restrictions on the termination analysis, as was noted e.g. in [3, 9]. The main problem in this context is that by using a single norm an occurrence of a functor in a term is always measured in the same way, although it might make sense to treat different occurrences of a same functor differently. Consider the following example from [9], in which the predicate `flat/2` can convert a list of lists of terms into a list containing all the terms in the lists.

Example 1.

```
flat([],[]).
flat([[]|T], R):- flat(T,R).
flat([[H|T]|T'], [H|R]):- flat([T|T'],R).
```

Here, it might make sense to use a norm that treats the list constructor [|] differently depending on which argument of `flat/2` it is used in. To measure the size of `flat/2`'s second argument, the *listlength* norm suffices (taking only outermost occurrences of [|] into account) whereas to measure the size of its first argument, it may be worthwhile to consider the innermost occurrences of [|] and count, effectively, the sum of the lengths of the elements of the outermost list.

To overcome these problems, the use of type information has been advocated before. The solution presented in [9] considers so-called *typed norms*. A typed norm measures a term according to its type; hence a single term can be measured differently depending on the type it is considered to belong to. For the above example, [9] would derive exactly the above mentioned norms: $\|.\|_1$ measuring the length of a list and $\|.\|_2$ counting the sum of the lengths of a list's elements.

$$\begin{array}{ll} \|[]\|_1 = 0 & \|[]\|_2 = 0 \\ \|[E|Es\|_1 = 1 + \|Es\|_1 & \|[E|Es]\|_2 = \|E\|_1 + \|Es\|_2 \end{array}$$

However, while the use of different norms opens up a number of possibilities, it also renders the computation of interargument relations far from trivial [8], as it involves computing relations that hold between different norms, like e.g. $\|t_1\|_1 = \|t_2\|_2$ in the `flat/2` example.

In this work, we reconsider the use of types in termination analysis. Contrary to earlier type-based approaches, we do *not* construct a particular norm for a term according to its type, but rather characterise the term's size in several ways, by counting the number of subterms of a particular type it contains. Reconsider Example 1 from above. Instead of characterising the size of terms like $[1, 2, 3]$ and $[[1], [2], [3, 4]]$ by the particularly constructed norms ($\|.\|_1$, respectively $\|.\|_2$), we characterise their sizes by counting the number of integer values occurring in them, the number of lists of integer values and the number of lists of lists of integer values. The first two of these measurements are applicable to both $[1, 2, 3]$ and $[[1], [2], [3, 4]]$ and enable to compare their sizes. This contrast with the difficulties that are encountered when one has to compare their sizes when measured with different norms $\|.\|_1$ and $\|.\|_2$. Our motivation is twofold:

- First of all, we believe the use of type information to be a powerful tool for termination analysis, as it provides a structured way to automatically derive suited norms. The problems encountered in the derivation of relations between typed norms [8] inspired us to explore a different way of representing these relations. This work complements our earlier work on deriving type based instantiation dependencies [4], which are a necessary ingredient for a type based termination analysis.
- In recent work [16], we have developed a binding-time analysis for logic programs that makes heavy use of termination analysis in order to decide what calls can be safely unfolded during specialisation. For this application, we feel that a lightweight termination analysis that allows to check termination quickly by searching in a structured way through a number of possible norms without requiring the user to assist in the construction of these norms may be more suited than a heavy analysis employing dedicated norms.

In Section 2 we recall the basics of the instantiatedness analysis of [4] and develop a type based size abstraction. In section 3 we introduce a type based termination condition and we conclude in Section 4.

2 Characterising Terms

2.1 About Terms and Types

We assume familiarity with logic programming concepts [11, 1]. We use $Term = T(\Sigma, V)$ to denote the set of terms constructed from a set of function symbols

Σ and variables V. Types, like terms are constructed from (type) variables and (type) symbols. We denote by $\mathcal{T} = \mathcal{T}(\Sigma_{\mathcal{T}}, V_{\mathcal{T}})$ the set of types constructed from type variables $V_{\mathcal{T}}$ and type symbols $\Sigma_{\mathcal{T}}$. We assume that the sets of symbols, variables, type symbols and type variables are fixed and do not overlap. We write $t : \tau$ to specify that term t has type τ and f/n to specify that (function or type) symbol f has arity n. Besides the usual substitutions which are mappings from variables to terms, we also have type substitutions which are mappings from type variables to types. We assume a standard notion of strong typing as for example in Mercury [14]. This means that each predicate is associated with a single type declaration and that programs are well typed. Hence, one can associate a unique type with every term and variable in a program clause. A type containing variables is said to be *polymorphic*, otherwise it is *monomorphic*. The application of a type substitution to a polymorphic type gives a new type which is an *instance* of the original type. For each type symbol, a unique type rule associates that symbol with a finite set of function symbols. A finite set of type rules forms a type definition and associates terms with types.

Definition 2. *The rule for a type symbol $h/n \in \Sigma_{\mathcal{T}}$ is a definition of the form $h(\bar{V}) \longrightarrow f_1(\bar{\tau}_1)\,;\dots\,;\, f_k(\bar{\tau}_k)$. where: $\bar{V}$ is an n-tuple from $V_{\mathcal{T}}$; for $1 \leq i \leq k$, $f_i/m \in \Sigma$ with $\bar{\tau}_i$ an m-tuple from $\mathcal{T}$; and type variables occurring in the right hand side occur in $\bar{V}$. The function symbols $\{f_1, \dots, f_k\}$ are said to be associated with the type symbol h. A finite set of type rules is called a type definition.*

In addition to types defined by the user, we assume also predefined types which are associated with (possibly infinite) sets of terms. For example, the types `int` and `char` may be specified to consists of the integers, respectively characters.

Example 2. The keyword `type` introduces a type rule:

```
- type list(T) ---> [] ; [T | list(T)].
```

The function symbols $[\,]$ (nil) and $[|]$ (cons) are associated with the type symbol $list$. The type definition defines also the denotation of each type (the set of terms belonging to the type). For this example, terms of polymorphic type $list(T)$ are variables (typed $list(T)$), terms of the form $[\,]$, or terms of the form $[t_1|t_2]$ with t_1 of type T and t_2 of type $list(T)$. As T is a type variable we cannot determine or commit to its structure under instantiation. Hence only a variable (typed T) can be of type T. Applying the type substitution $\{T/int\}$ on $list(T)$ gives the type $list(int)$. Terms of the form $[t_1|t_2]$ are of type $list(int)$ if t_1 is of type int and t_2 is of type $list(int)$. Type instances can also be polymorphic, e.g. $list(list(T))$.

The next definition specifies the *constituents* of a type τ. These are the possible types for subterms of terms of type τ.

Definition 3. *Let $h(\bar{\tau}) \longrightarrow f_1(\bar{\tau}_1)\,;\dots\,;\, f_k(\bar{\tau}_k)$ be an instance of a rule in a type definition ρ and τ an argument of one of the $f_i(\bar{\tau}_i)$. We say that τ is a ρ-constituent of $h(\bar{\tau})$. The constituents relation is the minimal pre-order (reflexive and transitive) $\preceq_\rho$: $\mathcal{T} \times \mathcal{T}$ including all pairs $\tau' \preceq_\rho \tau$ such that τ'*

is a ρ-constituent of τ. The set of ρ-constituents of τ is $Constituents_\rho(\tau) = \left\{ \tau' \in \mathcal{T} \,\middle|\, \tau' \preceq_\rho \tau \right\}$. *When clear from the context we omit ρ and write $\preceq$ and* $Constituents$.

Example 3. With $list/1$ as in Example 2 and the atomic type int, we have:

- $T \preceq list(T)$, $list(T) \preceq list(T)$, $int \preceq list(int)$, $int \preceq int$, $list(T) \preceq list(list(T))$, $T \preceq list(list(T))$, $list(list(T)) \preceq list(list(T))$, $T \preceq T$, $\ldots$
- $Constituents(int) = \{int\}$, $Constituents(T) = \{T\}$, $Constituents(list(T)) = \{T, list(T)\}$, $Constituents(list(int)) = \{int, list(int)\}$.

To ensure termination of our analysis, we impose the restriction that each type must have a finite set of constituents. This is a fairly sensible restriction as a type with an infinite set of constituents is not well defined. As an example, consider the type definition `type t(T) ---> f(t(t(T))).` The type $t(T)$ is defined in terms of its constituent type $t(t(T))$, which in turn is defined in terms of its constituent type $t(t(t(T)))$, $\ldots$. Similar conditions are common in other analyses of typed programs, e.g. [13].

2.2 Size Information

In what follows, we measure the "size" of a term with respect to a particular constituent of its type. The τ-size of a term $t : \tau'$ is defined for types $\tau \preceq \tau'$ and is the number of subterms in t having τ as type.

Definition 4. *Let τ and τ' be defined by a type definition ρ. The* τ-size *of $t : \tau'$ with $\tau \in Constituents(\tau')$ is defined as*

$$\sigma_\tau(t) = \#\{s \mid s : \tau \textit{ is a subterm of } t\}.$$

As such, a term can be measured by a finite number of measures, one for each of the constituents of its type. The table denotes the sizes of several terms s of type $list(list(int))$ when measured by counting, respectively, the subterms of type int, $list(int)$ and $list(list(int))$. Note that a same functor is measured differently according to which type it belongs to. If the term [] is considered to be of type $list(list(int))$ (as in the first row of the table, its $list(int)$-size is 0. If, on the other hand, [] is considered to be of type $list(int)$ (as the innermost [] in the second row of the table, its $list(int)$-size is 1. Note also that the $list(list(int))$-size of the terms in the example is one more than their listlength norm.

s	σ_{int}	$\sigma_{list(int)}$	$\sigma_{list(list(int))}$
[]	0	0	1
[[]]	0	1	2
[[1]]	1	2	2
[[1,2],[3]]	3	5	3

During termination analysis, one is not only interested in the concrete sizes of terms, but also in the relations that exist between the sizes of a predicate's arguments upon success. These relations are hard to compute when they have to relate sizes measured by different norms as in e.g. [8] but are straightforward to compute when the applied measurement is the same, which is the case in our approach since we can compute a size relation with respect to a particular constituent. For such a constituent, say τ, a τ-size relation is an (in)equality relation between the τ-sizes of program variables. Symbolically, an occurrence of a program variable in a τ-size relation represents the τ-size of the variable's value.

To ease the discussion, let us assume that programs are in normal form, meaning that every atom is either of the form $p(X_1, \ldots, X_n)$, $X = Y$ or $X = f(X_1, \ldots, X_n)$. First, we define the *τ-size abstraction* of a unification as follows:

Definition 5. *(τ-size abstraction)*

- *Let τ be a type and X, Y variables of type τ'. If $\tau \in Constituents(\tau')$, then the τ-size abstraction of $X = Y$ is $X = Y$, otherwise it is true.*
- *Let τ be a type, $X, Y_1, \ldots, Y_n$ variables of types $\tau_0, \tau_1, \ldots, \tau_n$ respectively. If $\tau \in Constituents(\tau_0)$, then the τ-size abstraction of $X = f(Y_1, \ldots, Y_n)$ is*

$$X = c + W_1 + \ldots + W_k$$

where: if $\tau = \tau_0$ then $c = 1$ else $c = 0$ and $\{W_1, \ldots, W_k\} = \{Y_i | 1 \leq i \leq n,\ \tau \preceq \tau_i\}$, otherwise it is true.

Note that the τ-size abstraction of a unification $X = \ldots$ is true (the top element of the abstract domain) and carries no information when τ is not a constituent of X's type. Note also that the term $f(Y_1, \ldots, Y_n)$ is not counted (being of a different type) in the τ-abstraction of the unification if X is of a type other than τ; only its subterms are taken into account.

Example 4. Assume the variable X is of type $list(int)$. The following table denotes the int and $list(int)$ abstractions of some unifications:

	$int-abstraction$	$list(int)-abstraction$
$X = []$	$X = 0$	$X = 1$
$X = [E \mid Es]$	$X = E + Es$	$X = 1 + Es$

The int-abstraction of $X = []$ resembles the fact that upon success of the unification, X's value does not contain an integer value. Likewise, its $list(int)$ abstraction denotes that X's value contains a single subterm of type $list(int)$. Also, the $list(int)$-abstraction of $X = [E|Es]$ denotes that the X's value contains precisely one more subterm of type $list(int)$ than Es does and the unification's int-abstraction denotes that the number of integer values in X equals the sum of the number of integer values in E (which is 1) and those in Es.

Starting from the τ-size abstraction of the unifications, one can compute a valid τ-size abstraction of a predicate definition by abstract interpretation. In order to obtain a finite analysis, one has to further approximate the abstract domain using, for example, polyhedral approximations [2, 6] or disjunctions of monotonicity and equality constraints [10].

Example 5. Consider the definition of the classical `append/3` predicate:

```
pred append(list(int),list(int),list(int)).
append(X,Y,Z):- X = [], Y = Z.
append(X,Y,Z):- X = [E|Es], append(Es,Y,Zs), Z = [E|Zs].
```

Examples of valid int and $list(int)$ abstractions of `append/3` are respectively $Z = X + Y$ and $Z + 1 = X + Y$.

When considering a term of type $list(int)$, there is only a minor difference between counting the number of integer values versus the number of integer lists in such a term, since both actually end up denoting the length of the list (up to a constant) and thereby the traditional *listlength* norm. For more involved lists (or types in general), this is no longer the case. Consider for example the terms $[[1,2],[3]]$, $[[2],[3]]$ and $[[2],[3|X]]$ of type $list(list(int))$. While in this case, the $list(list(int))$-size again denotes the length of the outermost lists, the $list(int)$-size does not. We have that $\sigma_{list(list(int))}([[1,2],[3]]) = \sigma_{list(list(int))}([[2],[3]]) = \sigma_{list(list(int))}([[1,2],[3|X]]) = 3$ while $\sigma_{list(int)}([[1,2],[3]]) = 5, \sigma_{list(int)}([[2],[3]]) = 4$ and $\sigma_{list(int)}([[2],[3|X]]) = 3 + X$. For the `flat/2` predicate from Example 1, the latter difference implies the difference between the ability or disability of proving termination for `flat/2`.

3 Reformulating the Termination Condition

Recall that the basic motivation for computing size relations is to discard the necessity to reason over the computed answer substitutions in the definition of acceptability (Definition 1). In addition to the computed size relations, one needs instantiation information that can be combined with the size relations. Since in our approach we measure the size of a term by counting the number of subterms of a particular type it has, the instantiation information we require is whether further instantiating the term can introduce more subterms of the type of interest. In [4], we introduce precisely such an instantiation property:

Definition 6. *Let ρ define the types τ and τ'. We say that the term $t{:}\tau'$ is τ-instantiated (under ρ) if: (1) τ is a constituent of τ'; and (2) there does not exist a well-typed instance $t\sigma{:}\tau'$ containing a variable of type τ. The predicate $\mu^{\rho}_{\tau}(t{:}\tau')$ is true if and only if $t{:}\tau'$ is τ-instantiated.*

The table shows some terms of type $list(int)$. $\mu_{list(int)}([1,X]) = true$ because all *list(int)*-subterms of well-typed instances of $[1,X]$ are instantiated. On the other hand, $\mu_{list(int)}([1|X]) = false$ because the subterm X of type *list(int)* is a variable. Also $\mu_{int}([1|X]) = false$ as e.g. $[1,Y|Z]$ is an instance with the variable Y of type *int*.

$t{:}list(int)$	$\mu_{list(int)}(t)$	$\mu_{int}(t)$
$[1,2]$	$true$	$true$
$[1,X]$	$true$	$false$
$[1\|X]$	$false$	$false$
$[\,]$	$true$	$true$

In [4], we developed an analysis that computes the instantiation dependencies that exist between the arguments of a predicate upon success of the predicate. The dependencies are expressed by positive boolean (Pos) formulas at the level of a particular constituent of the types in the predicate's definition. For the `append/3` predicate from above in which each argument is declared having $list(int)$ as type, it derives for example the int-dependency $z \leftrightarrow x \wedge y$ – denoting that all subterms of Z of type int are instantiated if and only if those of X and Y are. The derived $list(int)$ dependency, on the other hand, is $x \wedge (y \leftrightarrow z)$ – denoting that all subterms of X of type $list(int)$ are instantiated (in other words the backbone of the list is instantiated when `append/3` succeeds) and that those of Y are instantiated if and only if those of Z are instantiated. Note that we write a lower case variable x to denote the propositional variable representing τ-instantiatedness of the program variable X.

Given the τ-instantiatedness of a query, one can use the τ-instantiatedness analysis of [4] to compute the τ-instantiatedness of all calls selected when the query is executed under a left-to-right computation rule. (Similar as one can compute the groundness of selected calls based on the results of a bottom-up POS-analysis of the program). Hence, we can reformulate the acceptability condition as follows:

Definition 7. *Let S be a set of atomic queries and P a definite directly recursive program. P is acceptable with respect to S if for every recursive predicate $p(\tau_1, \ldots, \tau_n)$ such that $call(P, S)$ contains calls to it, the following holds:*
There exists a type $\tau \in Constituents(\tau_1, \ldots, \tau_n)$ and a set of argument positions $\{j_1, \ldots, j_m\}$ such that

- *$\{j_1, \ldots, j_m\} \subseteq \{1, \ldots, n\}$ and every atom $p(t_1 : \tau_1, \ldots, t_n : \tau_n) \in call(P, S)$ is τ-instantiated in every argument position $j_1, \ldots, j_m$.*
- *for every clause $p(H_1 : \tau_1, \ldots, H_n : \tau_n) \leftarrow B_1, \ldots, B_{l-1}, p(X_1 : \tau_1, \ldots, X_n : \tau_n), B_{l+1}, \ldots, B_m \in P$:*

$$(\mu_1, \ldots, \mu_{l-1}) \models \sigma_\tau(H_{j_1}) + \ldots + \sigma_\tau(H_{j_m}) > \sigma_\tau(X_{j_1}) + \ldots + \sigma_\tau(X_{j_m})$$

where for all i, μ_i denotes the τ-size relation associated with B_i.

Let us, by means of example, reconsider the `flat/2` predicate from the introduction, now written in normal form:

```
flat(X,Y):- X = [], Y = [].
flat(X,Y):- X = [A|B], A = [], flat(B, Y).
flat(X,Y):- X = [A|B], A = [H|T], N = [T|B], flat(N,R), Y = [H|R].
```

Again, assume that `flat/2` is defined with type $list(list(int))$ for its first argument, and $list(int)$ for its second argument. Assume we want to prove termination for a query $flat([[1, 2], [3]], R)$; The analysis of [4]can show that the first argument in every call to `flat/2` is $list(list(int))$-instantiated, $list(int)$-instantiated and int-instantiated. Using the $list(list(int))$-size abstraction, one cannot prove termination because of the third clause: $X = 1 + B$ and $N = 1 + B$ hence $N = X$. However, using the $list(int)$-size abstraction, for the first recursive clause, one can derive $X = 1 + B$ and for the second clause $X = A + B$,

$A = 1 + T$ and $N = T + B$, i.e. $X = 1 + N$ and acceptability is proven. More generally, instead of considering a single size, one could consider a linear combinations of sizes. For examples with a size τ_1 such that the first argument remains the same in the first recursive clause and gets smaller in the second, and a size τ_2 where it is the other way around, one could prove acceptability by considering their sum.

Note that the above sketched technique can not only be used to prove termination of a given query, but also the other way round: to derive a sufficient condition for a query to terminate. Indeed, after computing the size and instantiation relations (a call-independent computation process), one could search for possibly instantiation states of a predicate's arguments such that the condition of definition 7 is satisfied. Since the domain of possible instantiations is finite for every argument, the search process must terminate.

4 Discussion

In this abstract, which is a report on work in progress, we have defined a sufficient condition for termination of statically typed logic programs. In our approach, no norm needs to be specified beforehand; a number of size-measures are automatically derived from the type information. This relates our work to [9], which considers the automatic derivation of suitable norms based on type information.

Our approach is quite different, however. [9] derives for each kind of term a norm that is tailored towards measuring the size of the particular term. A problem with this approach is the computation of size relations; the relations that hold between the sizes of a predicate's arguments. The fact that these sizes can be computed by different norms makes the derivation of these size relations far from trivial [8]. In our approach, although a number of different measures are constructed, terms are always compared according to the *same* measure (counting their subterms of a particular type). This reduces the derivation of size-relations to the classical case in which these relations are expressed using a single norm. The drawback is a possible loss of generality, since we consider only relations between arguments that have a common constituent. Whether or not this is acceptable in practice needs experimental work. It is not difficult to write a predicate $p(\tau_1, \tau_2)$ such that τ_1 and τ_2 have no common constituents while the sizes of $t1$ and $t2$ in atoms $p(t1, t2)$ of the success set of $p/2$ are related. However, we expect that such size relations are seldom needed in proving acceptability. Indeed, consider a recursive clause $q(X) \leftarrow p(X, Y), \ldots, q(Z)$ the types of X and Z must be identical hence another call in the body would have to convert Y back to a term Z of type τ_1, a rather unlikely program scheme.

References

1. K. R. Apt. Logic programming. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science, Volume B, Formal Models and Semantics*, pages 493–574. Elsevier Science Publishers B.V., 1990.

2. F. Benoy and A. King. Inferring argument size relationships with CLP(R). In *Proceedings of LOPSTR'96*, volume 1207 of *Lecture Notes in Computer Science*, pages 204–223, 1997.
3. Annalisa Bossi, Nicoletta Cocco, and Massimo Fabris. Typed norms. In Bernd Krieg-Brückner, editor, *ESOP'92, 4th European Symposium on Programming*, volume 582 of *Lecture Notes in Computer Science*, pages 73–92, Rennes, France, 26–28 February 1992. Springer.
4. Maurice Bruynooghe, Wim Vanhoof, and M Codish. Pos(T) : Analyzing dependencies in typed logic programs. In *Proceedings of Andrei Ershov Fourth International Conference "Perspectives of System Informatics"*, 2001. accepted.
5. Michael Codish and Cohavit Taboch. A semantic basis for termination analysis of logic programs and its realization using symbolic norm constraints. In *ALP/HOA*, pages 31–45, 1997.
6. Patrick Cousot and Nicholas Halbwachs. Automatic discovery of linear restraints among variables of a program. In *Conference Record of the Fifth annual ACM Symposium on Principles of Programming Languages*, pages 84–96. ACM, ACM, 1978.
7. Danny De Schreye, C Verschaetse, and Maurice Bruynooghe. A framework for analysing the termination of definite logic programs with respect to call patterns. In ICOT, editor, *Proc. of the Fifth Generation Computer Systems, FGCS92*, pages 481–488. ICOT, 1992.
8. S Decorte, Danny De Schreye, and M Fabris. Exploiting the power of typed norms in automatic inference of interargument relations. Report CW 246, Department of Computer Science, K.U.Leuven, Leuven, Belgium, January 1997.
9. Stefaan Decorte, Danny de Schreye, and Massimo Fabris. Automatic inference of norms: A missing link in automatic termination analysis. In *Logic Programming - Proceedings of the 1993 International Symposium*, pages 420–436, 1993.
10. Naomi Lindenstrauss and Yehoshua Sagiv. Automatic termination analysis of logic programs. In Lee Naish, editor, *Proceedings of the 14th International Conference on Logic Programming*, pages 63–77, Cambridge, July 8–11 1997. MIT Press.
11. J. W. Lloyd. *Foundations of Logic Programming*. Springer-Verlag, 1987.
12. Danny De Schreye and Stefaan Decorte. Termination of logic programs: the never-ending story. *The Journal of Logic Programming*, 19 & 20:199–260, May 1994.
13. J.-G. Smaus, P.M. Hill, and A.M. King. Mode analysis domains for typed logic programs. In A. Bossi, editor, *Proceedings of the 9th International Workshop on Logic-based Program Synthesis and Transformation*, pages 82 – 101. Springer-Verlag, 2000.
14. Zoltan Somogyi, Fergus Henderson, and Thomas Conway. The execution algorithm of Mercury, an efficient purely declarative logic programming language. *Journal of Logic Programming*, 29(1–3):17–64, 1996.
15. Jeffrey D. Ullman and Allen Van Gelder. Efficient tests for top-down termination of logical rules. *JACM*, 35(2):345–373, 1988.
16. W. Vanhoof and M. Bruynooghe. Binding-time annotations without binding-time analysis (extended abstract). In N. Dershowitz, editor, *Extended Abstracts of the Fifth International Workshop on Termination (WST'01)*, pages 52 – 54, Utrecht, The Netherlands, 2001. Full paper submitted for publication.