Combining norms to prove termination

Genaim, S.; Codish, M.; Gallagher, John Patrick; Lagoon, V.

Published in:
Verification, model checking, and abstract interpretation, Third International Workshop, VMCAI 2002

Publication date:
2002

Document Version
Publisher's PDF, also known as Version of record

Citation for published version (APA):
Genaim, S., Codish, M., Gallagher, J. P., & Lagoon, V. (2002). Combining norms to prove termination. In A. Cortesi (Ed.), *Verification, model checking, and abstract interpretation, Third International Workshop, VMCAI 2002: Venice, Italy, January 21-22, 2002* (pp. 126-138). Springer. Lecture Notes in Computer Science Vol. 2294

Combining Norms to Prove Termination

Samir Genaim[1], Michael Codish[1], John Gallagher[2], and Vitaly Lagoon[3]

[1] Dept. of Computer Science, Ben-Gurion University of the Negev, Israel
[2] Dept. of Computer Science, University of Bristol, United Kingdom
[3] Dept. of Computer Science and Software Eng., University of Melbourne, Australia

Abstract. Automatic termination analysers typically measure the size of terms applying norms which are mappings from terms to the natural numbers. This paper illustrates how to enable the use of size functions defined as tuples of these simpler norm functions. This approach enables us to simplify the problem of deriving automatically a candidate norm with which to prove termination. Instead of deriving a single, complex norm function, it is sufficient to determine a collection of simpler norms, some combination of which, leads to a proof of termination. We propose that a collection of simple norms, one for each of the recursive data-types in the program, is often a suitable choice. We first demonstrate the power of combining norm functions and then the adequacy of combining norms based on regular types.

1 Introduction

Termination analysis aims to determine that a given program definitely terminates on a given input. An analyser must guarantee a (correct) verdict within a finite amount of time. Such a tool typically reports either "yes" - it succeeded to prove termination, and in this case the program is guaranteed to terminate; or "no" - it did not succeed to prove termination. The quality of the tool is a function of its usability. A strong tool will succeed to prove termination for a wide range of terminating programs, preferably with less intervention from the user.

Proofs of termination are often based on size functions which map program states to the elements of a well founded domain. A proof follows by showing that the states encountered through computation decrease in size and in particular as the program goes through its loops. As the domain is well-founded and the size of the input is bounded, the size of the initial state can decrease only a finite number of times and hence the computation must terminate.

For logic programs, loops occur through recursion and it is the size of the predicate calls that is required to decrease between recursive calls. Termination analysers such as those described in [6, 19, 22] choose the natural numbers as the well-founded domain. Size is measured using so-called semi-linear norms [2] which map to the natural numbers and define the size of a term as the sum of the sizes of some of its arguments.

In this setting, a term is said to be *rigid* with respect to a given norm if its size does not change under instantiation. For example, assuming a list-length

A. Cortesi (Ed.): VMCAI 2002, LNCS 2294, pp. 126–138, 2002.

norm (which indicates the number of elements in a list), both $[X, Y, Z]$ and $[X, Y, Z|Xs]$ contain 3 elements but only the first term is rigid as the length of the second term can change under instantiation. To illustrate the importance of this notion for termination analysis, consider the recursive clause of the *append*/3 relation: $append([X|Xs], Ys, [X|Zs]) \leftarrow append(Xs, Ys, Zs)$. To prove termination it does not suffice to observe that the length of the list in the first (and third) argument decreases in the recursive call (by one). One must also ensure that the argument is rigid when this clause is used. Otherwise the decrease in size could occur infinitely many times. Analysers hence maintain two types of information: about size — to detect a decrease; and about instantiation — to detect rigidity.

Instantiation information with respect to the given norm is obtained through abstract interpretation over the domain Pos of positive Boolean functions. The domain elements are interpreted as instantiation dependencies with respect to the given norm. For example, a formula of the form $x \wedge (y \rightarrow z)$ describes a program state in which x is definitely bound to a rigid term and there exists an instantiation dependency such that whenever y becomes bound to a rigid term then so does z. For details on Pos see [20].

Size relations express linear information about the sizes of terms with respect to a given norm function [1, 4, 7, 16]. For example, the relation $x \leq z \ \wedge \ y \leq z$ describes a program state in which the sizes of the terms associated with x and y are less or equal to the size of the term associated with z. Similarly, a relation of the form $z = x + y$ describes a state in which the sum of the sizes of the terms associated with x and y is equal to the size of the term associated with z. Several methods for inferring size relations are described in the literature [1, 4, 7, 8]. They differ primarily in their approach to obtaining a finite analysis as the abstract domain of size relations contains infinite chains.

This paper makes two contributions. First we address the situation where termination analysis should consider a combination of several norms. Namely, the size function used to prove termination combines several different measures on terms, perhaps because at least one of these measures decreases, or because a linear combination of the measures decreases. In many cases termination proofs follow due to the extra precision gained from dependencies between the size (and instantiation) information with respect to the different norms. In [17] the idea of using tuple of norm was used to increase the precision of *lower-bound time-complexity analysis*.

Second, we consider an alternative approach to guessing a suitable norm for termination analysis. Instead of trying to derive a single complex norm function (perhaps defined as a set of interdependent norms), we derive a collection of simpler norms, some combination of which, hopefully leads to a proof of termination. We do not specify how these norms should be combined. Instead, the system tries to find an appropriate combination. Of course, a general solution is impossible because if the program is terminating then there always exists a well founded domain and a size function which satisfy the requirements for the proof of termination [13].

Guessing a suitable norm reduces the level of intervention by the user and is often considered the main missing link in automatic termination analysis [10]. It has been recognised that type information provides a useful insight to this problem [3, 10, 21, 11, 12, 24] as recursive types represent recursive data-structures and thus identify potential sources of infinite recursion. We infer one norm per recursive data type in the program. Intuitively, for each type σ a corresponding norm $\|\cdot\|_\sigma$ counts the number of subterms of type σ in (typed) terms. This idea has been applied recently also in [24]. We take the extra step and propose that combining this collection of norms results in a very powerful technique not only for the inter-arguments size relations analysis but also for the instantiation dependency analysis.

Our presentation is based on regular types, expressed as deterministic "regular unary logic" (RUL) programs [25]. The types could either be declared or inferred, and we do not even require that the types are correct, although we are more likely to derive useful norms for proving termination if the types are correct and accurate.

Our aim is to generate norms from the types inferred by a recent type inference system [14]. This system does not use a "widening" to introduce recursive types. This means that recursion in the inferred types always reflects some recursive dependency in the program itself. For this reason it seems a promising starting point for deriving norms for termination analysis.

2 Preliminaries

Termination analysis for logic programs can be implemented (as for example in [6, 22]) using a technique termed abstract compilation. The program to be analysed is first abstracted, using the chosen norm, to corresponding constraint logic programs over CLP(R) and CLP(B) programs. These describe size and instantiation dependencies specified by the original program. The analyser characterises also size and instantiation for data occurring in loops. We do not detail the techniques in which this information is derived. Details can be found in the literature on termination analysis. See for example [9] for a survey and [6, 22] for specific analysers. Instead we limit our presentation on termination to the abstraction process to CLP(R) and explain intuitively the results obtained.

At the heart of the process is the choice of a norm. A semi-linear norm $|\cdot|$ is a mapping from terms to the natural numbers defined recursively such that $|X| = 0$ for a variable X and for each function symbol f/n in the underlying signature there is a statement of the form

$$|f(t_1, \ldots, t_n)| = c_f + \Sigma_{i \in I_f} |t_i|$$

where constant c_f and indecies $I_f \subseteq \{1, \ldots, n\}$ are determined by f/n.

In the examples we mention two norms: list-length (ll) which measures the number of elements of a list, and term-size (ts) which measures the number of nodes in the tree representation of a term. These are defined as:

$$\left|T\right|_{ll} = \begin{cases} 1 + \left|Xs\right|_{ll} & \text{if } T = [X|Xs] \\ 0 & \text{otherwise} \end{cases} \qquad \left|T\right|_{ts} = \begin{cases} 1 + \sum_{i=1}^{n} \left|t_i\right|_{ts} & \text{if } f/n \in \Sigma \text{ and } T = f(t_1, \ldots, t_n) \\ 0 & \text{otherwise} \end{cases}$$

The abstraction of a program with respect to a given norm is obtained by systematically replacing the predicate arguments in the program by corresponding abstract arguments. These are obtained by applying the norm to the argument, except that, whenever the norm is applied to a variable it is mapped to a fresh variable representing its size, instead of being mapped to 0. A given variable is mapped to the same size variable, wherever it occurs in a clause. For example, consider the *append*/3 relation depicted below (on the left), and its abstraction using the list-length norm (on the right). The concrete term [ ] is abstracted to 0 because $\left|[\,]\right|_{ll} = 0$ and the concrete term $[A|B]$ is abstracted to $1 + B1$ because $\left|[A|B]\right|_{ll} = 1 + \left|B\right|_{ll}$ which we denote as $1 + B1$. The CLP(R) program on the right is an abstraction of the concrete logic program on the left, in the sense that whenever $append(t_1, t_2, t_3)$ is a consequence of the concrete program, then $append(\left|t_1\right|_{ll}, \left|t_2\right|_{ll}, \left|t_3\right|_{ll})$ is a consequence of the abstract program.

```
append([],A,A).                    append(0,A1,A1).
append([A|B],C,[A|D]) :-           append(1+B1,C,1+D1) :-
    append(B,C,D).                     append(B1,C1,D1).
```

The *append* program specifies the relation $\{\ (x, y, z) \mid z = x.y\ \}$ (z equals the concatenation of x and y). The abstract program specifies the relation $\{\ (x, y, z) \mid z = x + y\ \}$ (the length of z is equal to the sum of the lengths of x and y). The instantiation analysis which can be obtained by applying a Pos analysis to the program on the right specifies the relation $\{\ (x, y, z) \mid x \wedge (y \leftrightarrow z)\ \}$ (x is rigid with respect to the norm and that y is rigid if and only if z is). A termination analysis based on the *list-length* norm infers also that all the loops in the program are of the form $append(x, y, z) \leftarrow append(u, v, w)$ with size information: $(u{<}x) \wedge (y{=}v) \wedge (w{<}z)$. From all of this information together it infers that the program terminates for queries in which the first or third arguments are instantiated to rigid terms. For details concerning the specific termination analyser we have extended in this work, see [6, 15].

3 Combining Norms

When basing termination analysis on program abstraction it is important to remember that variables occurring in the abstract program range over information about size and rigidity *with respect to* a given norm. This makes it difficult to apply one norm on one part of the program and another on a different part. First, one must know how to interpret values for each abstracted variable (with respect to which norm); and second, one must take care that variables abstracted by different norms do not interact (if different occurrences of variable X are abstracted by different norms then this can lead to problems). Both of these problems are

solved if care is taken so that the two abstractions introduce distinct sets of abstracted variables. Namely, the abstraction of a variable X by the i^{th} norm is X_i (interpreted as "the size of X by $norm_i$").

The key idea in this paper is to combine two or more norms by applying them simultaneously. This means that each argument in a predicate of the original program is replaced by two or more (renamed apart) abstract arguments each one specifying size and rigidity information with respect to the corresponding norm. The advantage of this approach is that inter-arguments relations can provide information about the dependencies with respect to each norm and between different norms. A similar phenomenon is observed in [12] where the authors abstract each argument by a single but different norm (depending on its type). However, we do not encounter the technical difficulties described in [12] by considering a semantic approach based on binary clauses as described in [6].

Example 1. Consider the program below (on the left) where some of the program points have been annotated (e.g., ⓐ). Termination analysis (of ground queries) using a list-length norm or a term-size norm does not succeed. The program contains three types of loops: (1) those where list-length is invariant but term-size decreases — recursive calls to point ⓐ, (2) those where list-length decreases but term-size increases — recursive calls to point ⓑ, and (3) those where both measures decrease — recursive calls to point ⓒ (keep in mind that $\left|0\right|_{ts} = 1$).

To perform a termination analysis combining the two norms each argument in the original program is abstracted first with respect to term-size and then with respect to list-length. The resulting abstract program is given below (on the right). Note that the two abstractions introduce disjoint sets of variables. Analysing this program indicates that all loops decrease in one of the two arguments which correspond respectively to the term-size and list-length of the original program. For ground queries both arguments are rigid (each with respect to the corresponding norm).

```
p([_]).
p([s(s(X)),Y|Xs]) :-
    ⓐ p([X,Y|Xs]),
    ⓑ p([s(s(s(Y)))|Xs]).
p([0|Xs]) :-
    ⓒ p(Xs).
```

```
p(2+X1,1).
p(4+X1+Y1+Xs1, 2+Xs2) :-
     p(2+X1+Y1+Xs1,2+Xs2),
     p(5+Y1+Xs1,1+Xs2).
p(2+Xs1,1+Xs2) :-
     p(Xs1,Xs2).
```

□

The previous example illustrates an argument for a program which is directly recursive. Our approach is not restricted to direct recursion as the termination analyzer we use makes all (indirect) loops explicit in terms of a direct recursion.

Example 2. Consider the program below (left) which multiplies the elements (natural numbers) in a (non-empty) list by iteratively replacing the first two elements by their multiplication. The program terminates if the list is ground. Attempting to prove this automatically using the term-size norm will indicate that the calls to *times* and *plus* are rigid (in their first and second arguments) and decrease in size (in their first arguments). So, corresponding calls to these

predicates surely terminate. However the loop on *factor* does not decrease in term-size (we replace the first two elements by their multiplication). Termination for these loops can be shown using the list-length norm. This, on the other hand gives no useful information for *times* and *plus*. So, neither of the two single analyses provide a proof or termination.

The abstract program for the combined analysis is given below (right). Each argument in the original program is abstracted first with respect to term-size and then with respect to list-length. Analysing this program does give a proof of termination because in all loops one of the two abstract arguments corresponding to the first argument in the original program decreases in size and is rigid (with respect to the appropriate norm).

```
factor([X],X).
factor([X,Y|Xs],T) :-
  times(X,Y,Z),
  factor([Z|Xs],T).
times(0,X,0).
times(s(X),Y,Z) :-
  times(X,Y,XY),
  plus(XY,Y,Z).
plus(0,X,X).
plus(s(X),Y,s(Z)):-
  plus(X,Y,Z).
```

```
factor(2+X1,1, X1,X2).
factor(2+X1+Y1+Xs1,2+Xs2, T1,T2) :-
     times(X1,X2, Y1,Y2, Z1,Z2),
     factor(1+Z1+Xs1,1+Xs2, T1,T2).
times(1,0, X1,X2, 1,0).
times(1+X1,0, Y1,Y2, Z1,Z2) :-
     times(X1,X2, Y1,Y2, XY1,XY2),
     plus(XY1,XY2, Y1,Y2, Z1,Z2).
plus(1,0,X1,X2,X1,X2).
plus(1+X1,0, Y1,Y2, 1+Z1,0) :-
     plus(X1,X2, Y1,Y2, Z1,Z2).
```

□

In the previous two examples we observe that when performing two separate analyses, each loop in the program decreases in size for at least one of two measures considered. The question is: Does this constitute a proof of termination? The answer depends on rigidity information. The point is that when observing a decreasing size with respect to one of the norms, we must observe also rigidity with respect to the same norm. For the above two examples, assuming that the initial query is ground (in the respective first arguments of p or $factor$), we can guarantee that each loop is both decreasing and rigid with respect to the appropriate norm (because rigidity with respect to term-size implies rigidity with respect to list-length). This is not always the case as demonstrated by the following example.

Example 3. For the following program neither of the two separate termination analyses, using list-length or term-size, detect a decrease in size for the loop on `t`. The combined analysis does give a proof of termination for queries in which the first argument of t is bound to a ground term. With the combined analysis we maintain a dependency between the term-size of N and the list-length of Xs in $ll(N, Xs)$ (they are equal).

```
t(N) :-
  ll(N,Xs),
  select(_,Xs,Xs1),
  ll(M,Xs1), t(M).
t(0).
```

```
ll(s(N),[X|Xs]) :- ll(N,Xs).
ll(0,[]).
select(X,[Y|Xs],[Y|Ys]) :-
   select(X,Xs,Ys).
select(X,[X|Xs],Xs).
```

□

Correctness: The correctness of our approach is straightforward. All we have done is to implicitly duplicate the original arguments of each concrete predicate. E.g. the *plus* definition becomes:

```
plus(0,0, X,X, X,X).
plus(s(X),s(X), Y,Y, s(Z),s(Z)):- plus(X,X, Y,Y, Z,Z).
```

and clearly the success set of this program is isomorphic to the original. Then, each copy of an argument is abstracted with respect to a corresponding norm. Correctness follows because the different norms rename the copies apart and because the analyses (for size and instantiation dependencies) applied to each copy are known to be correct. At first it might seem that we could as well have done the analyses separately. But (1) this would complicate the specification of the termination check; and (2) by doing the analyses together we often derive (size and rigidity) dependencies between the different abstractions of various arguments. This sometimes helps provide termination proofs. These are the main differences between our approach and the one described in [24].

4 Norms from Types

In this section we reconsider how norms can be defined based on type information which may be inferred or provided. We refer (as do others) to such norms as *typed-norms*. This has been considered previously in [3, 10, 21, 11, 12, 24]. Inferring norms from type information makes sense as recursive types represent recursive data-structures and thus identify some potential sources of infinite recursion. Moreover, typed-norms are more refined than semi-linear norms because whereas semi-linear norms measure the size of a term T according to its prime functor (recursively, as a function of the size of its arguments), typed-norms define the size of T based on its type. This means that the same term can be measured differently depending on its type. This is particularly useful when the same function symbol may occur in different type contexts. Our construction is based on the notion of *regular types* [25]. The main intuition is that for each type σ defined in the program, a typed-norm $\|\cdot\|_\sigma$ counts the number of sub-terms of type σ in the (typed) term it is applied to. The novelty in our approach is to then compose the norms corresponding to the (recursive) types defined in the program. This leads to a powerful technique which avoids many of the problems encountered in previous works.

Regular Types: A *regular type* is a set of terms defined by a regular tree grammar. For our purposes, the formulation of regular tree grammars using regular unary logic (RUL) programs is convenient. An RUL program is a logic program consisting of clauses of the form: $\tau(f(X_1,\ldots,X_n)) \leftarrow \tau_1(X_1),\ldots,\tau_n(X_n)$, where $X_1,\ldots,X_n$ are distinct variables. If f has arity zero, then the body of the clause is *true*. An RUL program is deterministic if no two clause heads have a common instance. In this paper we assume deterministic RUL programs whenever we mention RUL programs. Let R be an RUL program, and τ be a predicate in

R. Then the *regular type* τ is the success set of τ in R. There is a distinguished regular type *any*, which represents the set of all terms over the signature.

Example 4. The following RUL program defines *list_of_nat*, the regular type consisting of the set of lists of natural numbers in successor notation.

```
list_of_nat([]).                    nat(0).
list_of_nat([X|Xs]) :-              nat(s(X)) :- nat(X).
    nat(X), list_of_nat(Xs).
```

□

We assume that each predicate p/n in a given program comes with a type declaration of the form ":–type($\tau_1, \ldots, \tau_n$)" where $\tau_1, \ldots, \tau_n$ are types defined in an accompanying RUL. In our examples we use declared types, but inferred types, or a combination of declared and inferred types can be used as well. The more accurate the types are, the more likely there are to derive useful norms for proving termination.

Example 5. The programs given in Examples 1 and 2 can be typed by adding the following declarations which assume the type definitions given in Example 4.

```
:- type p(list_of_nat).          :- type factor(list_of_nat,nat).
                                 :- type times(nat,nat,nat).
                                 :- type plus(nat,nat,nat).
```

These declarations be can inferred automatically using the goal directed analysis described in [14].

□

Defining Norms from Regular Types: The idea of type-based norms consists in associating each type σ used in the program with a corresponding norm function $\|\cdot\|_\sigma$. When applied to a term t of type τ (denoted $t{:}\tau$) $\|\cdot\|_\sigma$ counts the number of subterms of type σ within t. Typed norms can be computed directly from an RUL program simply by running the RUL on the term to be measured. For example, let σ and τ be RUL predicates and let t be a term of type τ. Then $\|t{:}\tau\|_\sigma$ is the number of calls to σ encountered when executing the query $\tau(t)$ (excluding calls in which the argument is a variable). In fact it is implemented as a meta-interpreter for RUL's which counts calls.

Typed-norm definitions are derived from an RUL program as follows: for each type σ defined in the program (excluding `any`) and clause $\tau(f(X_1, \ldots, X_n)) \leftarrow \tau_1(X_1), \ldots, \tau_n(X_n)$ we introduce an equation of the form

$$\|f(X_1, ..., X_n) : \tau\|_\sigma = c(\tau, \sigma) + \|X_1{:}\tau_1\|_\sigma + \ldots + \|X_n{:}\tau_n\|_\sigma$$

where $c(\tau, \sigma) = 1$ if $\tau = \sigma$, and 0 otherwise. In addition we assume that $\|t{:}\tau\|_{any} = 0$, for all $t{:}\tau$ and that $\|X{:}\tau\|_\sigma = 0$ where X is a variable, for all τ and σ.

Example 6. Consider the type definition from Example 4. There are two types and so we define two norm functions, one to count the number of subterms of type `list_of_nat` (denoted by l) and one to count the number of subterms of

type `nat` (denoted by n) within a term (of type `list_of_nat` or of type `nat`). For instance, to evaluate $\| [s(0), s(s(0))] \|_l$ and $\| [s(0), s(s(0))] \|_n$ we count respectively the number of calls to `list_of_nat` and to `nat` in the derivation of the query "`?- list_of_nat([s(0),s(s(0))])`" (and this works out to 3 and 5). The typed-norms are defined as:

$$\|T\|_l = \begin{cases} 1 + \|Xs\|_l & \text{if } T = [X|Xs] \\ 1 & \text{if } T = [\,] \\ 0 & \text{if } T = s(X) \\ 0 & \text{if } T = 0 \\ 0 & \text{otherwise} \end{cases} \qquad \|T\|_n = \begin{cases} 1 + \|X\|_n & \text{if } T = s(X) \\ 1 & \text{if } T = 0 \\ \|X\|_n + \|Xs\|_n & \text{if } T = [X|Xs] \\ 0 & \text{if } T = [\,] \\ 0 & \text{otherwise} \end{cases}$$

□

Program abstraction with respect to typed norms is similar to the usual abstraction with respect to semi-linear norms. The abstraction for type σ replaces an argument of type τ by $\|t{:}\tau\|_\sigma$ except that typed variable $X{:}\tau$ is mapped to a corresponding size variable X_σ if the predicate defining σ is reachable from the predicate definition τ and otherwise to 0. Abstracting the programs shown in Examples 1 and 2 according to the *typed-norms* of Example 6 give similar results (except for constants) to the abstraction made in the Examples.

Example 7. The following program (14.4 in [23]) colors a map so that no two adjacent regions have the same color. The predicates *member*/2 and *select*/3 (omitted) are the standard predicates (see [23]). A map is represented as a list of regions where each region has a color (represented by a variable) and a list of colors (represented by variables) for the adjoining regions. An example initial query is given in the box (note the use of shared variables).

```
color_map([Region|Regions],Colors) :-
   color_region(Region,Colors),
   color_map(Regions,Colors).
color_map([],Colors).

color_region(region(Color,NBRs),Cs) :-
   select(Color,Cs,Cs1),
   members(NBRss,Cs1).

members([X|Xs],Ys) :-
   member(X,Ys),
   members(Xs,Ys).
members([],Ys).
```

```
?- color_map(
     [region(P,[E]),          % portugal
      region(E,[F,P]),        % spain
      region(F,[E,I,S,B,G,L]),% france
      region(B,[F,H,L,G]),    % belgium
      region(H,[B,G]),        % holland
      region(G,[F,A,S,H,B,L]),% germany
      region(L,[F,B,G]),      % luxembourg
      region(I,[F,A,S]),      % italy
      region(S,[F,I,A,G]),    % switzerland
      region(A,[I,S,G])],     % austria
     [red,yellow,blue,white]).
```

Proving termination for this program is not straightforward and cannot be performed using the available termination analysers. The term-size norm is not suitable because the list of regions in the initial query contains variables (and hence is not rigid with respect to term-size). The list-length norm is not suitable because the first clause invokes a call to `color_region` which traverses the list of neighbours in that region (so even if the first argument in `color_map` is rigid, still the lists of neighbours inside the elements of that list are not rigid). A specialised semi-linear norm cannot be defined because the list functor occurs in two different type contexts (for regions and for neighbours) and should be

treated differently in each. Using norms derived from types we obtain a suitable norm and a proof of termination. The types for this program are given as:

```
:- type color_map(list_of_region,list_of_color).
:- type color_region(region,list_of_color).
:- type select(color,list_of_color,list_of_color).
:- type members(list_of_color,list_of_color).
:- type member(color,list_of_color).

list_of_region([]).                    list_of_color([]).
list_of_region([X|Xs]) :-              list_of_color([X|Xs]) :-
    region(X), list_of_region(Xs).         color(X), list_of_color(Xs).
region(region(A,B)) :-                 color(red).    color(blue).
    color(A), list_of_color(B).        color(white).  color(yellow).
```

□

5 Implementation

The implementation of an analyser which supports the combination of norms and typed-norms is derived from the termination analyser described in [6] simply by changing the abstraction module. No other changes are necessary. The user provides a program and selects a set of norms and then the program is abstracted with respect to this selection. Norms can be selected from a predefined collection (like *listlength, termsize, etc,*) or defined by the user. Alternatively the user can supply types (inferred or declared) and specify that the analyser should use a combination of the corresponding typed-norms. For regular types we use the analyser described in [14]. The termination analyser can be accessed at `http://wwww.cs.bgu.ac.il/~mcodish/TeminWeb`.

6 Related Work

The idea of using type information to define norms has previously been studied by Bossi *et al.* [3], Martin *et al.* [21], Decorte *et al.* [10, 11, 12] and more recently by Vanhoof and Bruynooghe [24]. Our approach builds primarily on the techniques of Decorte *et al.* and of Vanhoof and Bruynooghe.

Decorte *et al.* observe in [10] and [11] that different predicate arguments can be measured by different (typed-) norms. The authors also observe that inter-arguments relations between different norms can provide useful information. Their work suffers from two restrictions. First, at most one typed-norm can be applied to a term of a given type. Second, as reported in [12], the use of different norms for different types renders the computation of inter-arguments relations far from trivial (they propose a solution based on a notion re-execution).

Vanhoof and Bruynooghe make the observation in [24] that interesting typed-norms can be defined by counting the number of subterms of a given type occurring in (typed-) terms. They address the "single norm per type" restriction of the Decorte *et al.*, observing that sometimes an argument should be measured by several different norms and they propose to consider a collection of

norms, one per type defined in the program. However, to avoid the problems with inter-arguments relations described in [12] they perform a collection of separate analyses one for each norm. As a consequence there are no inter-arguments relations between arguments measured by different norms. However, the nature of their basic norms (counting in a term the number of subterms of a given type) is quite powerful and for many examples this leads to a proof of termination.

Our approach to combining norms simply by duplicating predicate arguments and applying the analyses simultaneously solves the problems encountered in both of the lines of work described above: we take the same collection of norms as proposed in [24]; we allow mixed norms (determined by different types); we allow several norms for arguments of the same type; we maintain inter-arguments size dependencies between different measures; and finally, if there are several candidate norms and it is not clear which (linear combination of different norms) is suitable then our system can find it automatically. This approach was also applied in [17] in the context of *lower-bound time-complexity analysis* to increase the precision of the inter-arguments size relations analysis.

The use of types to refine other program analyses has recently been considered in [5] and [18]. In those papers the authors observe that if a program is well-typed then only subterms of the same type can be unified. Hence, type information is used to refine the analysis of unifications in a program by considering for each type which of subterms can be matched. The idea of multiplying the arguments of predicates — one copy per type containing size information with respect to that type — closely resembles the approach used in [18] where for each type the corresponding copy of an argument contains its subterms of that type.

7 Conclusion

This paper describes a technique which enables a termination analyser to consider a combination of several (typed-) norms. The simple idea to replicate the arguments of a predicate for each norm goes a long way and solves many of the problems encountered in previous works. First because the analysis benefits from dependencies amongst different measures of the terms. Second because the analyser can discover which combination of the measures derived from the program's types is suitable to prove termination; and finally, because the implementation becomes straightforward.

References

[1] F. Benoy and A. King. Inferring argument size relationships with CLP(R). In *Sixth International Workshop on Logic Program Synthesis and Transformation (LOPSTR'96)*, pages 204–223, 1996.

[2] A. Bossi, N. Cocco, and M. Fabris. Proving termination of logic programs by exploiting term properties. In S. Abramsky and T.S.E. Maibaum, editors, *Proceedings of Tapsoft 1991*, volume 494 of *Lecture Notes in Computer Science*, pages 153–180. Springer-Verlag, Berlin, 1991.

[3] Annalisa Bossi, Nicoletta Cocco, and Massimo Fabris. Typed norms. In B. Krieg-Brückner, editor, *Proceeedings ESOP '92*, volume 582 of *Lecture Notes in Computer Science*, pages 73–92. Springer-Verlag, Berlin, 1992.

[4] A. Brodsky and Y. Sagiv. Inference of monotonicity constraints in Datalog programs. In *Proceedings of the Eighth ACM SIGACT-SIGART-SIGMOD Symposium on Principles of Database Systems*, pages 190–199, 1989.

[5] Maurice Bruynooghe, Wim Vanhoof, and Michael Codish. Pos(t): Analyzing dependencies in typed logic programs. Technical report, Presented at the Andrei Ershov Fourth International Conference on Perspectives of System Informatics, July 2001.

[6] M. Codish and C. Taboch. A semantic basis for the termination analysis of logic programs. *The Journal of Logic Programming*, 41(1):103–123, 1999.

[7] Patrick Cousot and Nicholas Halbwachs. Automatic discovery of linear restraints among variables of a program. In *Proceedings of the Fifth Annual ACM Symposium on Principles of Programming Languages*, pages 84–96, January 1978.

[8] D. De Schreye and K. Verschaetse. Deriving linear size relations for logic programs by abstract interpretation. *New Generation Computing*, 13(02):117–154, 1995.

[9] 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.

[10] Stefaan Decorte, Danny de Schreye, and Massimo Fabris. Automatic inference of norms: A missing link in automatic termination analysis. In Dale Miller, editor, *Logic Programming - Proceedings of the 1993 International Symposium*, pages 420–436, Massachusetts Institute of Technology, Cambridge, Massachusetts 021-42, 1993. The MIT Press.

[11] Stefaan Decorte, Danny De Schreye, and Massimo Fabris. Integrating types in termination analysis. Technical Report CW 222, K.U.Leuven, Department of Computer Science, January 1996.

[12] Stefaan Decorte, Danny De Schreye, and Massimo Fabris. Exploiting the power of typed norms in automatic inference of interargument relations. Technical Report CW 246, K.U.Leuven, Department of Computer Science, January 1997.

[13] R. W. Floyd. Assigning meanings to programs. In J.T Schwartz, editor, *Proceedings of Symposium in Applied Mathematics*, volume 19, Mathematical Aspects of Computer Science, pages 19–32, New York, 1967. American Mathematical Society, Providence, RI.

[14] J. Gallagher and G. Puebla. Abstract Interpretation over Non-Deterministic Finite Tree Automata for Set-Based Analysis of Logic Programs. In Shriram Krishnamurthi and C. R. Ramakrishnan, editors, *Practical Aspects of Declarative Languages, 4th International Symposium, PADL 2002, Portland, OR, USA, January 19-20, 2002*, volume 2257 of *Lecture Notes in Computer Science*, pages 243–261. Springer, 2002.

[15] Samir Genaim and Michael Codish. Inferring termination conditions for logic programs using backwards analysis. In R. Nieuwenhuis and A. Voronkov, editors, *Proceedings of the Eighth International Conference on Logic for Programming, Artificial Intelligence and Reasoning*, volume 2250 of *Lecture Notes in Artificial Intelligence*, pages 681–690. Springer-Verlag, December 2001.

[16] M. Karr. Affine relationships among variables of a program. *Acta Informatica*, 6:133–151, 1976.

[17] Andy King, Kish Shen, and Florence Benoy. Lower-bound time-complexity analysis of logic programs. In Jan Maluszynski, editor, *International Symposium on Logic Programming*, pages 261 – 276. MIT Press, November 1997.

[18] V. Lagoon and P. Stuckey. A framework for analysis of typed logic programs. In *FLOPS*, volume 2024 of *Lecture Notes in Computer Science*, pages 296–310. Springer-Verlag, Berlin, 2001.
[19] N. Lindenstrauss and Y. Sagiv. Automatic termination analysis of logic programs. In Lee Naish, editor, *Proceedings of the Fourteenth International Conference on Logic Programming*, pages 63–77, Leuven, Belgium, 1997. The MIT Press.
[20] K. Marriott and H. Søndergaard. Precise and efficient groundness analysis for logic programs. *ACM Letters on Programming Languages and Systems*, 2(1–4):181–196, 1993.
[21] Jon Martin and Andy King. Typed norms for typed logic programs. In *Logic Program Synthesis and Transformation*. Springer-Verlag, August 1996. Available at http://www.cs.ukc.ac.uk/pubs/1996/511.
[22] F. Mesnard and U. Neumerkel. Applying static analysis techniques for inferring termination conditions of logi programs. In *Static Analysis Symposium*, 2001.
[23] L. Sterling and E. Shapiro. *The Art of Prolog*. MIT Press, second edition, 1994.
[24] W. Vanhoof and M. Bruynooghe. When size does matter. Preproceedings of the Eleventh International Workshop on Logic-based Program Synthesis and Transformation (LOPSTR), November 2001.
[25] E. Yardeni and E.Y. Shapiro. A type system for logic programs. *Journal of Logic Programming*, 10(2):125–154, 1990.