Reuse of Results in Termination Analysis of Typed Logic Programs

Maurice Bruynooghe[1], Michael Codish[2], Samir Genaim[2⋆], and Wim Vanhoof[1]

[1] Katholieke Universiteit Leuven, Department of Computer Science
Celestijnenlaan 200A, B-3001 Heverlee, Belgium
{maurice,wimvh}@cs.kuleuven.ac.be
[2] Ben-Gurion University of the Negev, Department of Computer Science,
P.O.B. 653, 84105 Beer-Sheva, Israel
{mcodish,genaim}@cs.bgu.ac.il

Abstract. Recent works by the authors address the problem of automating the selection of a candidate norm for the purpose of termination analysis. These works illustrate a powerful technique in which a collection of simple type-based norms, one for each data type in the program, are combined together to provide the candidate norm. This paper extends these results by investigating type polymorphism. We show that by considering polymorphic types we reduce, without sacrificing precision, the number of type-based norms which should be combined to provide the candidate norm. Moreover, we show that when a generic polymorphic typed program component occurs in one or more specific type contexts, we need not reanalyze it. All of the information concerning its termination and its effect on the termination of other predicates in that context can be derived directly from the context independent analysis of that component based on norms derived from the polymorphic types.

1 Introduction

Termination analysis aims to determine that a given program definitely terminates for a specified, usually infinite, class of inputs. Proofs of termination are typically based on size functions (or *norms)* which map program states to the elements of a well founded domain. Termination is guaranteed if the states encountered through computation decrease in size.

For logic programs, loops occur through recursion and the size of a term is bounded if it is *rigid*. Namely, sufficiently instantiated so that its size does not change under further instantiation. Analyzers hence maintain two types of information: about the sizes of terms — to detect a decrease; and about their degree of instantiation — to detect rigidity. The ability to prove termination depends on finding a suitable norm and inferring sufficient size and instantiation information to detect with respect to that norm that all loops are decreasing and bound. In practice, most termination analyzers choose the natural numbers as the well-founded domain and measure size using semi-linear norms. Guessing a

⋆ Supported by a PhD fellowship from the Kreitman Foundation.

M. Hermenegildo and G. Puebla (Eds.): SAS 2002, LNCS 2477, pp. 477–492, 2002.

suitable norm reduces the level of intervention by the user and is often considered the main missing link in automatic termination analysis [11].

Termination analyzers typically rely on the user to select a candidate norm. Recently, some new ideas which are easily integrated into existing analyzers and facilitate automated norm selection emerged. In [26], a finite collection of norms is derived from the data types occurring in the program. The authors of [14] combine this collection in a single analysis and obtain a powerful system able to prove termination of programs not handled by previous implementations.

The current paper extends these ideas and shows that by considering polymorphic types we can reduce, without sacrificing precision, the number of norms which should be combined to provide the candidate norm. The intuition is simple: types which are instances of polymorphic typed variables need not be considered. All of the information required for termination analysis relevant to such types can be reconstructed from the polymorphic types. A preliminary investigation of the reuse of rigidity and size information based on polymorphic types are described in [6] and [26] respectively. However, these works focus mainly on monomorphic types and contain preliminary ideas for polymorphic types applicable only for simple instances of the polymorphic types.

The reuse of analysis results about polymorphic predicates is an important step in supporting modular termination analysis and in reducing the overall cost of termination analysis of large programs. It allows the analyzer to perform a polymorphic analysis of a module and to store results about the exported predicates in the module interface. The analysis of a module importing predicates in a specific type context can be completed by processing the results of the polymorphic analysis in the interface of the module defining the imported predicates (at least when there are no circular dependencies).

It has been recognized before that types provides a useful insight to the problem of guessing a norm [4,11,21,12,10] as recursive types represent recursive data-structures and thus identify potential sources of infinite recursion. The idea of combining norms has been suggested before by King *et al.* [16] in the context of lower-bound time-complexity analysis. For what concerns the problem of norm selection, [9] starts the analysis with a parameterized semi-linear norm and uses constraint solving to try to select one able to prove termination. Here we focus on (the combination of) type-based norms which are more refined than semi-linear norms and in particular on norms derived from polymorphic types.

The next section recalls some preliminaries about types. Section 3 reviews the previous work of the authors presented in [26] and [14] on defining type based norms and combining several norms in a single analysis. Section 4 presents the contribution of this paper extending the previous results for polymorphic types and proving that it is sufficient to consider norms corresponding to the polymorphic types without those for the types in the specific contexts in which they appear. We discuss related work in Section 5 and conclude in Section 6. A full version of this paper including detailed proofs and additional examples appears as [5].

2 Type Information

We assume familiarity with logic programming concepts [19,1]. We consider the set of terms constructed from a given set of function symbols Σ and variables V and use the notations f/n and p/n for an n-ary functor and predicate respectively. We adopt a standard notion of types for logic programs. 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 $\Sigma_{\mathcal{T}}$. The sets of type variables, type symbols, variables and function symbols are assumed to be disjoint. Besides the usual substitutions which are mappings from variables to terms, we also have type substitutions which are mappings from type variables to types. For each type symbol, a unique type rule associates that symbol with a finite set of function symbols. We allow function symbols to be overloaded, i.e. they can be used in different type rules. A finite set of type rules forms a type definition and associates terms with types.

Definition 1 (type). *A* type rule *for a type symbol $h/n \in \Sigma_{\mathcal{T}}$ is a definition of the form $h(\bar{v}) \longrightarrow f_1(\bar{\tau}_1) ; \ldots ; f_k(\bar{\tau}_k)$ where: $\bar{v}$ is an n-tuple of distinct type variables from $V_{\mathcal{T}}$, $f_1, \ldots, f_k$ are distinct function symbols from Σ associated with the type symbol h, $\bar{\tau}_i$ $(1 \leq i \leq k)$ are corresponding tuples from $\mathcal{T}$, and type variables in $\bar{\tau}_i$, if any, are from $\bar{v}$. A* type definition *is a finite set of type rules for distinct type symbols. A type definition containing variables is said to be* polymorphic*, otherwise it is* monomorphic*.*

The application of a type substitution to a type rule gives an *instance* of the original rule. In what follows, we treat type rules as well as their instances as type rules.[1] In addition to types defined by the user, we allow also predefined types. An example is `int`.

Example 1. Below are several instances of the polymorphic type rule

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

They define the monomorphic types $list(list(int))$ (for $T = list(int)$), $list(int)$ (for $T = int$) and the polymorphic type $list(list(V))$ (for $T = list(V)$).

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

□

A monomorphic type definition determines the denotation of each type it defines and predefined types have a predefined denotation. For example, the terms 1 and 2 are of type int and $[1, 2]$ is of type $list(int)$. A polymorphic type definition is a schema that defines denotations for all its monomorphic instances. A common restriction in type based program analysis is that any monomorphic type implicitly defined by a polymorphic type definition can be defined by a finite set of monomorphic type rules that are instances of the polymorphic schema.

[1] Although in this case several rules for the same type symbol can occur.

This excludes type rules such as $t(T) \longrightarrow f(t(t(T)))$ and avoids termination problems of analysis.

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

Definition 2 (type constituent). *Let τ be a type. We say that a type σ is a constituent of τ, denoted $\sigma \preceq \tau$, if there exists a term of type τ which has a subterm of type σ. The set of constituents of τ is denoted Constituents(τ). We denote by Constituents(ρ) the union of the constituent sets for all types defined in the type definition ρ.*

Example 2. In the context of Example 1, observe that:

- $Constituents(T) = \{T\}$
- $Constituents(list(T)) = \{T, list(T)\}$
- $Constituents(int) = \{int\}$
- $Constituents(list(int)) = \{int, list(int)\}$
- $Constituents(list(list(V))) = \{V, list(V), list(list(V))\}$ □

In what follows, it is often convenient to consider the normal form of programs. In normal form, each atom is of the form $p(X_1, \ldots, X_n)$, $X = Y$ or $X = f(X_1, \ldots, X_n)$ (with $X_1, \ldots, X_n$) different variables). To develop our approach, we adopt the concept of *well-typing* of [23]. It relies on the notion of *variable typing*, which is a function from variables to types. First, it is assumed that each predicate p/n has a unique declaration $p(\tau_1, \ldots, \tau_n)$ stating the (possibly polymorphic) types of its arguments. Next, it is assumed that the program has a *well-typing*. This means that each clause $p(X_1, \ldots, X_n) \leftarrow B_1, \ldots, B_m \in P$ has a variable typing μ such that (1) $p(\mu(X_1), \ldots, \mu(X_n))$ is the declared type of p/n, (2) for each B_i: (a) If the atom is of the form $X = Y$, then $\mu(X) = \mu(Y)$. (b) If the atom is of the form $X = f(Y_1, \ldots, Y_l)$ then $\mu(X) = \tau$ such that τ is defined by a type rule (instance) including an alternative $f(\tau_1, \ldots, \tau_m)$ and $\mu(Y_i) = \tau_i$ for each i. (c) If the atom is of the form $q(Y_1, \ldots, Y_l)$ then $q(\mu(Y_1), \ldots, \mu(Y_l))$ is an instance of the type declared for q/l. The well-typing associates a type with each variable (and term) in the program. The notation $t : \tau$ is used to indicate that τ is the type of t.

Types may be declared by the user, as for example in Mercury [25], or inferred, as for example by the type inference system described in [13]. Requiring the existence of a well-typing is no limitation of our method. There is always a variable typing such that condition (1) is satisfied. Also, introducing fresh variables and extra unifications, one can always meet condition (2.c). Hence violations are limited to the unifications. Ignoring unifications that are not well-typed ensure that the analysis results are correct (although less precise than with a well-typing).

3 Termination Using Type Based Norms

We consider universal termination for well-typed logic programs using Prolog's leftmost selection rule and assume that unifications do not violate the occurs

check. Given a program P and a set of initial atomic queries S, we denote by $calls(P,S)$ 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 a query $Q \in S$.

To measure the size of terms and atoms, one usually employs the notions of a *norm* and a *level mapping* which respectively map terms and atoms to natural numbers. Semi-linear norms [3] are well known. They map to the natural numbers and measure a term as the sum of the sizes of some of its arguments. For a term $f(t_1,\ldots,t_n)$, $\left|f(t_1,\ldots,t_n)\right| = c_f + \Sigma_{i\in I_f}\left|t_i\right|$ where constant c_f and indices $I_f \subseteq \{1,\ldots,n\}$ are determined by f/n; and for a variable X, $\left|X\right| = 0$.

A limitation of semi-linear norms is that the same function symbol f/n always makes the same contribution, c_f, independent of its context. Type based norms overcome this limitation. In our approach, a norm based on a type σ is defined to count the number of non-variable subterms of type σ in a term. It means that the contribution of a functor depends on the type context in which it appears. This simple idea turns out to be very powerful in practice and is captured in the following definition.

Definition 3 (type based norm). *Let ρ be a type definition and let σ be a type, predefined or defined in ρ. The σ-size of a term t of type τ, denoted $\left|t:\tau\right|_\sigma$, is computed by the following rules:*

1. *If $t = f(t_1,\ldots,t_n)$ is of type τ defined in ρ by the (unique) rule instance $\tau \longrightarrow f_1(\bar{\tau}_1)\,;\ldots;\,f_k(\bar{\tau}_k)$ and one of the $f_i(\bar{\tau}_i)$ on the right side of this rule is of the form $f(\tau_1,\ldots,\tau_n)$. Then, $\left|t:\tau\right|_\sigma = c(\sigma,\tau)+\left|t_1:\tau_1\right|_\sigma+\cdots+\left|t_n:\tau_n\right|_\sigma$ where $c(\sigma,\tau) =$ if $(\sigma=\tau)$ then 1 else 0.*
2. *If t is a non-variable term of predefined type τ and $\tau = \sigma$ then $\left|t:\tau\right|_\sigma = 1$.*
3. *Otherwise $\left|t:\tau\right|_\sigma = 0$*

Note the role of the expression $c(\sigma,\tau)$ in the first rule of the definition. Recall that we are counting the number of subterms of type σ in a term $t = f(t_1,\ldots,t_n)$ of type τ. Hence, in addition to the contribution from $t_1,\ldots,t_n$, we should count t as such a subterm ($c(\sigma,\tau)=1$) if $\sigma=\tau$, and otherwise not ($c(\sigma,\tau)=0$). Observe also that the above definition is meaningful for polymorphic types, including a type variable T. The only terms of type T in a well-typed program are variables. Their T-size, as well as the T-size of any other term is 0. Similar definitions for monomorphic types can be found in [14,26].

Example 3. The following is the norm based on the type $list(int)$ from Example 1 as derived from Definition 3.

$$\left|t\right|_{list(int)} = \begin{cases} 1+\left|t_2\right|_{list(int)} & \text{if } t=[t_1|t_2] : list(int) \\ 1 & \text{if } t=[\,] : list(int) \\ \left|t_1\right|_{list(int)}+\ldots+\left|t_n\right|_{list(int)} & \text{if } t=f(t_1,\ldots,t_n):\tau \text{ and } \tau \neq list(int) \\ 0 & \text{otherwise} \end{cases}$$

Note that $\left|t\right|_{list(int)}$ is 0 when $list(int)$ is not a constituent of the type of t. Similar norms can be derived for int and $list(list(int))$ as well as for T and $list(T)$. As the only well-typed terms of type T are variables, the T-size of all well-typed terms is 0. The norm based on $list(T)$ is identical

to the one presented above after replacing the occurrences of "$list(int)$" by "$list(T)$". The following illustrates the σ-sizes for several terms t and types σ.

$t : list(list(int))$	$\|t\|_{int}$	$\|t\|_{list(int)}$	$\|t\|_{list(list(int))}$
[ ]	0	0	1
[[ ]]	0	1	2
[[1]]	1	2	2
[[1, 2], [3]]	3	5	3

$t : list(T)$	$\|t\|_T$	$\|t\|_{list(T)}$
[ ]	0	1
$[X, Y]$	0	3

Note that equal subterms are measured differently according to the type context in which they appear. For instance, the typed term $[\,] : list(list(int))$ (first row on the left) measures 0 with $list(int)$-size, while $[\,] : list(int)$ (the inner [ ] in the second row on the left), measures 1 with $list(int)$-size. □

In practice, a term can be measured only by a finite number of measures, one for each of the constituents of its type. That is because when σ is not a constituent of τ the σ-size of a term t of type τ is always zero. More generally, one obtains a finite set of candidate norms to be used in the analysis of a predicate, namely those determined by the constituents of the types in its definition.

For proving termination, it often suffices to consider constituents that define recursive data types, as termination typically depends on a decrease for some recursively defined data structure. However, for what concerns size and rigidity information of a polymorphic predicate, we should consider all polymorphic constituents, as these may occur in a context binding type variables to a recursive data type.

Rigidity generalizes groundness. A definition that applies on the constituents of polymorphically as well as monomorphically typed terms is as follows:

Definition 4 (type based rigidity). *With σ a constituent of τ, a term $t : \tau$ is σ-rigid iff for all variables $X : \tau'$ occurring in t, σ is not a constituent of τ'.*

Note that the σ-size of a σ-rigid term $t : \tau$ is constant for all instances $t\theta$ of type τ.

Our technique is built upon the semantic basis for termination analysis described in [7]. In this approach a program is mapped to a (in general, infinite) set of binary clauses which make answers, calls and loops observable in a goal independent way. A binary clause of the form $p \leftarrow q$ in the semantics indicates that a call to p leads to a subsequent call to q. A termination analyzer is obtained by approximating this semantics with respect to size and rigidity information. This gives a finite approximation of the calls and loops from which the analyzer can try to determine that the program terminates.

Rigidity can be represented in the domain Pos of positive Boolean functions [20] as shown in [15]. A type based rigidity analysis is developed in [6]. Size relations express linear information about the sizes of terms with respect to a given norm function.

Termination analysis for logic programs can be based (examples are in [7, 18,22]) on a technique termed abstract compilation. The program to be analyzed is first abstracted, using the chosen norm, to a corresponding constraint logic program which describes the size and rigidity dependencies specified by

the original program. As explained in [6], the rigidity abstraction is applied on the normal form. Unifications are abstracted by Pos-formulas modeling them. Program variables (in arguments of predicates and in unifications) are replaced by corresponding rigidity variables representing the rigidity of the term bound to the variable. The size abstraction can be obtained by systematically replacing terms by corresponding abstract terms. The abstraction of a term t with respect to the norm based on a type σ its called the *σ-abstraction* and is denoted by $|t|^{\alpha}_{\sigma}$. It is obtained by applying the norm to the argument, except that, whenever the norm is applied to a program variable it is mapped to a corresponding size variable representing its size. In addition, for each size variable X we add the constraint $X \geq 0$ (as size must always be non-negative). However, some refinements are possible. When abstracting X of type τ with respect to τ we add $X \geq 1$ as any τ-rigid instance must contain at least one term of type τ. Similarly, when abstracting X of type τ with respect to σ and τ does not have σ as constituent, then the constraint $X = 0$ can be added as well-typed instances of $X : \tau$ will never contain subterms of type σ. These refinements can be crucial for proving termination.

Note that the σ-abstraction of $t : \tau$ gives a size expression which generalizes the sizes of all instances of t. Namely, if t' is a term instance of t, then $|t'|_{\sigma}$ is a "size instance" of the abstraction of t. In particular, if t is a variable X then X_{σ} denotes the number of subterms of type σ in the particular instance of X.

The intuition is that where program variables range over terms, size variables range over the sizes of terms with respect to $|\cdot|$ and where a program predicate p/n represents an n-ary relation on terms, the corresponding abstract predicate represents an n-ary relation on the sizes of terms. The abstract program, P^a, abstracts the concrete program, P, in the sense that whenever $p(t_1, \ldots, t_n)$ is a consequence of P, then $p(|t_1|, \ldots, |t_n|)$ is a consequence of P^a.

When abstracting the binary clause semantics defined in [7] we derive finite sets of abstract binary clauses and abstract call patterns. The recursive abstract binary clauses describe the loops in the program and the call patterns describe the calls that arise in its computations. The key idea in the analysis is then that non-termination of the original program P with an initial query G implies "non-termination" in the abstract binary program for some call pattern with a single recursive abstract binary clause. In contrast to techniques based on acceptability [8], the binary clauses approach can apply different level mappings for proving acceptability of different binary clauses from the same predicate definition. This turns out to be an important factor when considering termination analysis based on a combination of norms as required in our approach.

Example 4. Consider the classic `append/3` program

```
append([],Ys,Ys).
append([X|Xs],Ys,[X|Zs]):- append(Xs,Ys,Zs).
```

occurring in two different type contexts as specified respectively by:

```
:- pred append(list(list(int)),list(list(int)),list(list(int))).
:- pred append(list(nest(int)),list(nest(int)),list(nest(int))).
```

with the type definitions from Example 1 and:

```
:- type list(nest(int)) ---> []; [nest(int)|list(nest(int))]
:- type nest ---> e(int) ; l(list(nest)).
```

The types declared in the second context allow to represent lists of lists (to any depth) of integers capturing all sublists by the same constituent, unlike definitions such as $list(\ldots(list(int)\ldots)$ where lists at different depths must be specified explicitly and are captured by different constituents.

For the purpose of termination analysis, we distinguish five type constituents for append in the given contexts: int, $list(int)$, $list(list(int))$, $nest(int)$ and $list(nest(int))$. The corresponding abstractions are given below.

The concrete term [ ] is abstracted to 1 in the $list(int)$-, $list(list(int))$- and $list(nest(int))$-abstractions and to 0 in the others. The concrete term $[X|Xs]$ is abstracted respectively to $1+Xs$ in the $list(int)$- and $list(list(int))$-abstractions, to $X + Xs$ in the int- and $nest(int)$-abstractions, and to $1 + X + Xs$ in the $list(nest(int))$-abstraction.

list(list(int))- and list(int)-abstraction	int- and nest(int)-abstractions	list(nest(int))-abstraction
`append(1,Ys,Ys).`	`append(0,Ys,Ys).`	`append(1,Ys,Ys).`
`append(1+Xs,Ys,1+Zs):-`	`append(X+Xs,Ys,X+Zs):-`	`append(1+X+Xs,Ys,1+X+Zs):-`
`  append(Xs,Ys,Zs).`	`  append(Xs,Ys,Zs).`	`  append(Xs,Ys,Zs).`

In either of these contexts, termination analysis using any one of these norms infers that all of the loops in this program component are of the form $append(x, y, z) \leftarrow append(u, v, w)$ with size information: $(u{<}x) \wedge (y{=}v) \wedge (w{<}z)$. Together with the respective instantiation information it infers that the component terminates for queries in which the first or third arguments are instantiated to rigid terms. There is a subtlety here: for the int-norm the proof of termination relies on the fact that the int-size of variable X is at least 1.

As all three abstractions lead to a proof of termination, one might conclude that it does not matter which single norm we consider. However, depending on the context in which append is called, we might need inter-argument size and instantiation information with respect to any or several of the different norms in order to prove the termination of other parts of the program. □

As an alternative to the approach described in Example 4, one can consider a polymorphic type declaration: `:- pred append(list(T),list(T),list(T))` which has only two constituents: T and $list(T)$. One can observe that the only terms in the program of type T are variables (namely X). The T-size of X is 0, however, its T-abstraction is X_T. The $list(T)$- and T- abstractions work out to be the same as the $list(int)$- (left) and int- (middle) abstractions in Example 4.

In the next section we show that these two abstractions contain all the information that we need to analyze the termination of append and other parts of the programs it might occur in. In our approach, we combine the abstractions corresponding to the polymorphic constituents (T and $list(T)$ in the example). Basically, we duplicate the arguments of each predicate, making fresh copies,

and apply one abstraction to each set of arguments. For example, in the program below copies of the variables are indexed by $list(T)$ and by T indicating the abstraction they represent.

combined abstraction
append(1,0,Ys$_{list(T)}$,Ys$_T$,Ys$_{list(T)}$,Ys$_T$).
append(1+Xs$_{list(T)}$,X$_T$+Xs$_T$,Ys$_{list(T)}$,Ys$_T$,1+Z$_{list(T)}$,X$_T$+Zs$_T$):-
append(Xs$_{list(T)}$,Xs$_T$,Ys$_{list(T)}$,Ys$_T$,Zs$_{list(T)}$,Zs$_T$).

4 Polymorphic Reuse

Our aim is the following: We have a polymorphic typed program component P, for instance the append/3 relation typed with $list(T)$, and we have performed a type based termination analysis for P. Now we find predicates from P imported in a program Q imposing a specific type context. For instance, calls to append/3 in the contexts which manipulate lists of lists of integers and nested lists of integers as defined in Example 4). To analyze Q we want to reuse the analysis of P in its polymorphic context to: (1) consider the termination behavior of predicates defined in P for the more specific type context of Q, and (2) to reuse the size and rigidity information obtained for P based on polymorphic types to infer information based on the more specific type context in which it appears. This may be required for the termination analysis of the predicates defined in Q.

Type polymorphism is an important abstraction tool: a predicate defined with arguments of a polymorphic type can be called with actual arguments of any type that are instances of the defined types. We will argue two claims. First, that the polymorphic declarations are at least as precise as their instances as far as proving termination of a program. Namely, if we did not succeed to prove termination of a program using norms based on general polymorphic type definitions, then considering norms derived from more specific type information will not make the difference. Second, and more interesting, the size and rigidity information obtained during a termination analysis based on polymorphic type declarations for a program component can be reused to reason about termination of that component occurring in arbitrary more specific type contexts. Basing termination analysis on polymorphic types has another advantage. Recall that our proposal is to apply combined analysis considering each of the type constituents relevant to a predicate. Polymorphic type definitions are typically more conservative in the number of constituents involved and this reduces the number of norms combined and hence the cost of the analysis.

The key point to observe concerning the analysis of a program with polymorphic type declarations is that from the very essence of what a polymorphic type variable represents it follows that the program does not manipulate its "polymorphic parts". These are either completely ignored or simply passed from one data structure to another. Otherwise, if the program did manipulate a variable with a polymorphic type then that variable would have a more specific type. For

example, if a term `[X|Xs]` is declared to be of type `list(T)` then the type of X cannot be determined by the program. It is said to be "polymorphic".

The meaning of this observation as far as basing termination analysis on norms derived from polymorphic type information is that the polymorphic types contain already all the information relevant to termination also for any possible type instances. In particular, this means that the analysis based on norms for different instances of the polymorphic definition will always give the same result.

The basic question to answer is how do the size and rigidity abstractions with respect to a constituent of a type instance relate to those of the constituents of a polymorphic type. One could expect that there is a corresponding constituent in the polymorphic typing. However this is not always the case.

Let us assume a program component P typed by a polymorphic type definition ρ. The program may occur in several different type contexts each with more specific type information. For instance, append typed with $list(T)$ may occur in the contexts $list(int)$ or $list(color)$. Our goal is to reuse without loss of precision the termination analysis of P based on ρ when P occurs in different contexts. For the rest of this section let ρ' be a monomorphic type definition obtained by instantiating the type rules in ρ by a type substitution $\vartheta = \{T \mapsto \pi\}$ and adding rules to define any new types not already defined in ρ. The program (component) P is typed both by ρ as well as by ρ'.

The constituents of ρ' can be classified into two classes: $\mathcal{A}_{\rho'}$, those originating as instances of types defined in ρ; and $\mathcal{B}_{\rho'}$, those originating from the instantiation of ρ (constituents of π).

Example 5. Consider the following polymorphic type definition ρ and its instance ρ' obtained by the type substitution $\{T \mapsto color\}$ and the added rule to define *color*.

ρ (polymorphic)	ρ' (monomorphic)

`list(T) ---> [ ]; [T|list(T)].`	`list(color) --->`
	`  [ ]; [color|list(color)].`
	`color --->`
	`  red; blue; white; yellow.`

The monomorphic instance has two constituents giving $\mathcal{A}_{\rho'} = \{list(color)\}$ (because $list(color)$ corresponds to $list(T)$ defined in ρ), and $\mathcal{B}_{\rho'} = \{color\}$ (because the parameter T is mapped to *color*). In this case $\mathcal{A}_{\rho'}$ and $\mathcal{B}_{\rho'}$ are disjoint, however this is not always the case. Consider for example a definition ρ'' for type instance $list(nest(int))$ as defined in Example 4. One can observe that $list(nest(int))$ is a constituent of $nest(int)$ as well as an instance of $list(T)$. This implies that $list(nest(int)) \in \mathcal{A}_{\rho''} \cap \mathcal{B}_{\rho''}$. □

Let τ be a type defined in ρ and $\tau' = \tau[T \mapsto \pi]$ be its monomorphic instance defined in ρ'. Let t be a term of type τ as well as of type τ'. This means that the only subterms of type π in t are variables. We will show that the size (abstraction) of t with respect to any type σ' defined in ρ' can be described in terms of the

sizes (abstractions) of t with respect to some collection of types defined in ρ. The intuition is that counting subterms of type σ' in a term t is precisely like counting subterms of the types σ from which σ' originates. A similar intuition says that the σ'-rigidity of a term can be verified by checking the σ-rigidity of the term for the constituents σ corresponding to σ'. The following definition identifies the constituents in ρ corresponding to a constituent σ'.

Definition 5. *Let ρ be a polymorphic type definition and $\rho' = \rho[T \mapsto \pi]$ a monomorphic instance of ρ. We denote by $\mathcal{I}(\sigma')$ the type constituents in ρ that correspond to the constituent σ' of ρ'. It is defined as:*

$$\mathcal{I}(\sigma') = \begin{cases} \{\sigma \in \rho \mid \sigma' = \sigma[T \mapsto \pi]\} \cup \{T\} & \text{if } \sigma' \preceq \pi \\ \{\sigma \in \rho \mid \sigma' = \sigma[T \mapsto \pi]\} & \text{otherwise} \end{cases}$$

□

Example 6. Consider the type instances: $list(color)$, $list(int)$ and $list(nest(int))$ of $list(T)$. We have from Definition 5: $\mathcal{I}(color) = \mathcal{I}(int) = \mathcal{I}(nest(int)) = \{T\} \cup \{T\} = \{T\}$ from the first case in the definition ("$\sigma' \preceq \pi$"), $\mathcal{I}(list(color)) = \mathcal{I}(list(int)) = \{list(T)\}$ from the second case in the definition ("otherwise"), and $\mathcal{I}(list(nest(int)) = \{list(T)\} \cup \{T\} = \{T, list(T)\}$ also from the first case in the definition. □

We now proceed to state the main results of this paper. First we consider the termination of a program Q that calls predicates from a polymorphic typed, by ρ, program P, in a more specific type instance, ρ'. In order to prove the termination of Q we need to derive size and rigidity information of P with respect to the constituents of ρ'. The straightforward approach is to reanalyze P. The alternative contributed by this paper is to reconstruct this information from the analysis results obtained for P with the polymorphic constituents of ρ. The following theorem shows how to reconstruct this information.

Theorem 1. *Let p/n be a predicate typed under the polymorphic type definition ρ with k constituents: $\sigma_1, \ldots, \sigma_k$ and let:*

$$p(x_1^{\sigma_1}, \ldots x_1^{\sigma_k}, \ldots, x_n^{\sigma_1}, \ldots x_n^{\sigma_k}) \leftarrow \pi \in [\![P^{\rho}]\!]_{CLP(N)}$$
$$p(x_1^{\sigma_1}, \ldots x_1^{\sigma_k}, \ldots, x_n^{\sigma_1}, \ldots x_n^{\sigma_k}) \leftarrow \varphi \in [\![P^{\rho}]\!]_{Pos}$$

denote the size relation and rigidity information derived in the combined analysis using the type-based norms corresponding to these constituents (π is a linear constraint and φ is a Boolean formula). And Let:

$$p(x_1^{\sigma'_1}, \ldots x_1^{\sigma'_l}, \ldots, x_n^{\sigma'_1}, \ldots x_n^{\sigma'_l}) \leftarrow \pi' \in [\![P^{\rho'}]\!]_{CLP(N)}$$
$$p(x_1^{\sigma'_1}, \ldots x_1^{\sigma'_l}, \ldots, x_n^{\sigma'_1}, \ldots x_n^{\sigma'_l}) \leftarrow \varphi' \in [\![P^{\rho'}]\!]_{Pos}$$

denote the size relation and rigidity information with respect to a type definition ρ' which is an instance of ρ. Then:

$$\pi' \stackrel{def}{\equiv} \exists X : \pi \wedge \bigwedge_{j=1}^{l} \bigwedge_{i=1}^{n} (x_i^{\sigma'_j} = \sum_{\sigma \in \mathcal{I}(\sigma'_j)} x_i^{\sigma})$$
$$\varphi' \stackrel{def}{\equiv} \exists X : \varphi \wedge \bigwedge_{j=1}^{l} \bigwedge_{i=1}^{n} (x_i^{\sigma'_j} \leftrightarrow \bigwedge_{\sigma \in \mathcal{I}(\sigma'_j)} x_i^{\sigma})$$

Proof:. *A proof sketch is in [5].* □

In practice the analyzer uses widening while computing the size-relations; this can lead to differences in precision between the results obtained via reuse and the results obtained via direct computation. Only experience can show whether there is a relevant difference.

Example 7. Consider again the append/3 relation with the polymorphic type declaration ρ as given in Example 5. The size relation and rigidity information derived in the combined analysis using the type-based norms corresponding to the constituents T and $list(T)$ are:

$$\begin{array}{l}\texttt{append(X}_T\texttt{,X}_{list(T)}\texttt{,Y}_T\texttt{,Y}_{list(T)}\texttt{,Z}_T\texttt{,Z}_{list(T)}\texttt{) :-}\\ \quad \texttt{X}_{list(T)}\texttt{+Y}_{list(T)}\texttt{=1+Z}_{list(T)} \ \wedge\ \texttt{X}_T\texttt{=Y}_T\texttt{+Z}_T\\ \texttt{append(X}_T\texttt{,X}_{list(T)}\texttt{,Y}_T\texttt{,Y}_{list(T)}\texttt{,Z}_T\texttt{,Z}_{list(T)}\texttt{) :-}\\ \quad \texttt{((X}_{list(T)} \wedge \texttt{Y}_{list(T)}\texttt{)} \leftrightarrow \texttt{Z}_{list(T)}\texttt{)} \wedge \texttt{(}\ \texttt{X}_T \leftrightarrow \texttt{(Y}_T \wedge \texttt{Z}_T\texttt{))}\end{array}$$

we can derive the size relation and rigidity information for the combined analysis of $\sigma_1 = nest(int)$ and $\sigma_2 = list(nest(int))$ as follows:

$$\begin{array}{l}\texttt{append(A}_{\sigma_1}\texttt{,A}_{\sigma_2}\texttt{, B}_{\sigma_1}\texttt{,B}_{\sigma_2}\texttt{, C}_{\sigma_1}\texttt{,C}_{\sigma_2}\texttt{) :-}\\ \qquad \texttt{A}_{\sigma_1}\ \texttt{=}\ \texttt{X}_T \wedge \texttt{A}_{\sigma_2}\ \texttt{=}\ \texttt{1+X}_T\texttt{+X}_{list(T)}\wedge\\ \qquad \texttt{B}_{\sigma_1}\ \texttt{=}\ \texttt{Y}_T \wedge \texttt{B}_{\sigma_2}\ \texttt{=}\ \texttt{Y}_{list(T)}\texttt{+Y}_T \wedge\\ \qquad \texttt{C}_{\sigma_1}\ \texttt{=}\ \texttt{Z}_T \wedge \texttt{C}_{\sigma_2}\ \texttt{=}\ \texttt{1+Z}_T\texttt{+Z}_{list(T)} \wedge\\ \qquad \texttt{X}_{list(T)}\texttt{+Y}_{list(T)}\texttt{=1+Z}_{list(T)} \ \wedge\ \texttt{X}_T\texttt{=Y}_T\texttt{+Z}_T\\ \texttt{append(A}_{\sigma_1}\texttt{,A}_{\sigma_2}\texttt{, B}_{\sigma_1}\texttt{,B}_{\sigma_2}\texttt{, C}_{\sigma_1}\texttt{,C}_{\sigma_2}\texttt{) :-}\\ \qquad \texttt{A}_{\sigma_1} \leftrightarrow \texttt{X}_T \wedge \texttt{A}_{\sigma_2} \leftrightarrow \texttt{(X}_{list(T)} \wedge \texttt{X}_T\texttt{)} \wedge\\ \qquad \texttt{B}_{\sigma_1} \leftrightarrow \texttt{Y}_T \wedge \texttt{B}_{\sigma_2} \leftrightarrow \texttt{(Y}_{list(T)}\wedge \texttt{Y}_T\texttt{)} \wedge\\ \qquad \texttt{C}_{\sigma_1} \leftrightarrow \texttt{Z}_T \wedge \texttt{C}_{\sigma_2} \leftrightarrow \texttt{(Z}_{list(T)}\wedge \texttt{Z}_T\texttt{)} \wedge\\ \qquad \texttt{((X}_{list(T)} \wedge \texttt{Y}_{list(T)}\texttt{)} \leftrightarrow \texttt{Z}_{list(T)}\texttt{)} \wedge \texttt{(X}_T \leftrightarrow \texttt{(Y}_T \wedge \texttt{Z}_T\texttt{))}\end{array}$$

□

Another issue concerns the power of the analysis. Perhaps using type based constituents of a monomorphic instance, one can get an automated proof for a predicate that has no counterpart in the type based constituents of the polymorphic type. The next theorem shows this is impossible.

Theorem 2. *If termination of a predicate can be shown with type based norms derived from its monomorphic typing then termination can also be shown for type based norms derived from its polymorphic typing.*

Proof (intuition). If there is a proof of termination then there is a proof for every binary clause. Consider a binary clause (without loosing generality, we assume only one argument): `p(x) :-` π`, p(y).` Firstly, if an argument position is rigid with respect to σ' of the monomorphic typing ρ' then, it is also rigid with respect to all $\sigma \in \mathcal{I}(\sigma')$ of the polymorphic typing ρ. A termination proof of the binary clause for the monomorphic typing implies that there exists a constituent σ' such that the σ'-size relation of π implies that $y^{\sigma'}$ is strictly less than $x^{\sigma'}$. But according to Theorem 1, the σ'-size relation of π is a linear combination of the σ-size relations of the constituents $\sigma \in \mathcal{I}(\sigma')$. Hence there must be at least

one constituent σ in the polymorphic typing such that the σ-size relation of π implies that y^σ is strictly less than x^σ. □

In fact, it appears that a polymorphic analysis could even be more powerful than a monomorphic one. If $\mathcal{I}(\sigma')$ is not a singleton, then it could be that the monomorphic σ' based norm shows no reduction for some binary clause while a polymorphic σ based norm for a constituent of $\mathcal{I}(\sigma')$ does. The reason is that what is counted by the σ' based norm can be split over several polymorphic norms and that one of these norms shows a decrease while their sum does not.

Example 8. Consider the following typed logic program:

```
:- type pair(T1,T2) ---> p(T1,T2).

:- pred q(pair(list(int),list(T)).
q(p([],_)).
q(p([X|Xs],Ys)) :-
     dupl(Ys,Ys,[],Zs),
     q(p(Xs,Zs)).

:- pred dupl(list(T),list(T),list(T),list(T)).
dupl([],Ys,Accm,Accm).
dupl([_|Xs],Ys,Accm,Zs) :-
     append(Ys,Accm,NewAccum),
     dupl(Xs,Ys,NewAccm,Zs).
```

Termination analysis based on the typed-norm $|.|_{int}$ obtain the binary clause $\mathtt{q(A)} \leftarrow [\mathtt{B{=}1{+}A}], \mathtt{q(B)}$ for $q/1$ which is sufficient for proving termination. But using the monomorphic instance which obtained by substituting T to int the analyzer fails to detect decreasing in the first argument of $q/1$ since the number subterm of type int is increasing. The same claim holds for $list(int)$.

□

All this shows that —ignoring the effects on precision caused by the widening of the size relations— it suffices for an automated type based termination proof to analyze a program component for the constituents of its polymorphic typing. Re-analyzing the program for the type based constituents of the context in which it appears is pointless.

5 Related Work

The idea of using type information to define norms for termination analysis has previously been studied by Bossi *et al.* [4], Martin *et al.* [21], and by Decorte *et al.* [11,12,10]. In this approach attention is focused on the class of programs, termination for which depends on disciplined manipulation of recursive data structures. Our approach to basing termination analysis on type information builds primarily on the technique of Decorte *et al.* and is described in [26] and [14]. These two works when combined together lead to an implemented system which avoids many of the difficulties and problems identified in the earlier works

in this area. The current paper extends this approach to deal with polymorphism and analysis reuse.

The use of types to refine other program analyses has recently been considered also in [6] and [17]. 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 subterms can be matched.

A preliminary exploration of polymorphism is in [26]. There, it observed that programs can be abstracted for polymorphic constituents. The authors notice that in the simple case where the instantiation $T \mapsto \pi$ of a polymorphic definition ρ does not interact with ρ itself then program abstraction for π information is identical to the polymorphic T-abstraction. The current paper handles the general case and shows how to reuse the polymorphic analyses in the context of other program components imposing more specific type contexts.

Modularity in termination analysis of logic programs has been considered before. See for example the recent paper [2] and references within. The motivation in these works is similar to ours. Namely by considering modules separately the required machinery or proof can be simplified (to focus on the individual module). In our approach this simplification stems from the fact that individual modules have more general (polymorphic) types. It is the polymorphic type definition for an individual module that captures the essence of how the module manipulates its (possibly more specific) data. For termination of that module we better focus on that more general information. An advantage of our approach is that it then spells out by looking at the instantiation of the polymorphic types in other modules exactly how to consider the combination and reuse of information between the modules.

6 Conclusion

This paper builds on the idea of basing termination analysis on type information. Recent work by the authors is presented in [26] and [14] and illustrates a powerful technique in which a collection of simple type-based norms — one for each data type in the program — are combined together to provide a candidate norm for termination analysis. An implementation based on these two works has proved to work well and avoid the difficulties described in earlier works.

The contribution of the current paper is to extend these previous results to work with polymorphic type information. Type polymorphism is an important abstraction tool and this turns out to be of practical importance when basing termination analysis on type information.

First because there are less types to consider when the definitions are more abstract; and second because even when a module occurs in one or more specific type contexts, all of the relevant information for that context can be obtained from the analysis based on its general, context independent, polymorphic types.

Acknowledgment. We thank John Gallagher and Vitaly Lagoon for the many helpful discussions on the issues presented in this paper.

References

1. K. R. Apt. Introduction to logic programming. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B: Formal Models and Semantics, pages 495–574. Elsevier, Amsterdam and The MIT Press, Cambridge, 1990.
2. A. Bossi, N. Cocco, S. Etalle and S. Rossi. On modular termination proofs of general logic programs. *Theory and Practice of Logic Programming*, 2(3):263–291, 2002.
3. 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.
4. A. Bossi, N. Cocco, and M. 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.
5. M. Bruynooghe, M. Codish, S. Genaim, and W. Vanhoof. Reuse of results in termination analysis of typed logic programs. Technical Report, July, 2002.
6. M. Bruynooghe, W. Vanhoof, and M. Codish. Pos(t): Analyzing dependencies in typed logic programs. In *Perspectives of System Informatics, 4th International Andrei Ershov Memorial Conference, (PSI)*. Lecture Notes in Computer Science, vol. 2244. Springer-Verlag, 406–420.
7. 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.
8. D. De Schreye, C. Verschaetse, and M. 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.
9. S. Decorte, D. De Schreye, and H. Vandecasteele. Constraint based automatic termination analysis of Logic Programs. *ACM Trans. Program. Lang. Syst.*, 21(6):1137–1195, November 1999.
10. S. Decorte and D. De Schreye. Demand-driven and constraint-based automatic left-termination analysis for logic programs. In Naish [24], pages 78–92.
11. S. Decorte, D. De Schreye, and M. Fabris. Automatic inference of norms: A missing link in automatic termination analysis. In D. Miller, editor, *Logic Programming - Proceedings of the 1993 International Symposium*, pages 420–436, Vancouver, Canada, 1993. MIT Press.
12. S. Decorte, D. De Schreye, and M. Fabris. Integrating types in termination analysis. Technical Report CW 222, K.U.Leuven, Department of Computer Science, January 1996.
13. 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.
14. S. Genaim, M. Codish, J. Gallagher, and V. Lagoon. Combining norms to prove termination. In Agostino Cortesi, editor, *Third International Workshop on Verification, Model Checking and Abstract Interpretation*, volume 2294 of *Lecture Notes in Computer Science*, pages 126–138. Springer-Verlag, January 2002.

15. R. Giacobazzi, S. K. Debray, and G. Levi. Generalized semantics and abstract interpretation for constraint logic programs. *The Journal of Logic Programming*, 25(3):191–247, Dec. 1995.
16. A. King, K. Shen, and F. 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.
17. 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.
18. N. Lindenstrauss and Y. Sagiv. Automatic termination analysis of logic programs. In Naish [24], pages 63–77.
19. J.W. Lloyd. *Foundations of Logic Programming*. Springer-Verlag, Berlin, 2^{nd} edition, 1987.
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. J. Martin and A. 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. A. Mycroft and R.A. O'Keefe. A Polymorphic Type System for Prolog. *Artificial Intelligence* 23(3): 295-307 1984.
24. L. Naish, editor. *Proceedings of Fourteenth International Conference on Logic Programming*, Leuven (Belgium), 1997. The MIT press.
25. Z. Somogyi, F. Henderson, and T. Conway. The execution algorithm of Mercury, an efficient purely declarative logic programming language. *Journal of Logic Programming*, 29(1–3):17–64, October–November 1996.
26. W. Vanhoof and M. Bruynooghe. When size does matter - Termination analysis for typed logic programs. *Logic Based Program Synthesis and Transformation, LOPSTR 2001, Revised Papers*, LNCS, Springer 2002. To appear.