

Exploiting the power of typed norms in automatic inference of interargument relations

Stefaan Decorte, Danny De Schreye, Massimo Fabris

Report CW 246, January 1997



Katholieke Universiteit Leuven
Department of Computer Science

Celestijnenlaan 200A – B-3001 Heverlee (Belgium)

Exploiting the power of typed norms in automatic inference of interargument relations

Stefaan Decorte, Danny De Schreye, Massimo Fabris

Report CW 246, January 1997

Department of Computer Science, K.U.Leuven

Abstract

Recently, the introduction of type formalisms for statically analysing logic programs has become a commonly applied technique. More specifically in termination analysis, typed norms have been introduced as a refined way to measure the sizes of terms and atoms. While the expressiveness of these typed norms makes them practically indispensable for a refined termination analysis - particularly when proving termination of queries with partially instantiated terms -, it was not clear how relations between the sizes of (differently measured) arguments of predicates could be derived, and what sense they would make. We present a technique based on abstract interpretation for computing such relations and indicate its role in termination analysis. The resulting analysis extends the power of some of the more refined automatic termination analysis techniques presented previously.

Exploiting the Power of Typed Norms in Automatic Inference of Interargument Relations

Stefaan Decorte* Danny De Schreye†

Department of Computer Science, K.U.Leuven,
Celestijnenlaan 200A, B-3001 Heverlee, Belgium.
e-mail : {stefaan,dannyd}@cs.kuleuven.ac.be

Massimo Fabris‡

Dip. di Matematica Pura e Applicata, Universita di Padova
Via Belzoni 7, 35131 Padova - Italy
e-mail : massimo@hilbert.math.unipd.it

Abstract

Recently, the introduction of type formalisms for statically analysing logic programs has become a commonly applied technique. More specifically in termination analysis, typed norms have been introduced as a refined way to measure the sizes of terms and atoms. While the expressiveness of these typed norms makes them practically indispensable for a refined termination analysis - particularly when proving termination of queries with partially instantiated terms -, it was not clear how relations between the sizes of (differently measured) arguments of predicates could be derived, and what sense they would make. We present a technique based on abstract interpretation for computing such relations and indicate its role in termination analysis. The resulting analysis extends the power of some of the more refined automatic termination analysis techniques presented previously.

1 Introduction

Types are fulfilling an increasingly important role in state of the art approaches to Logic Program termination analysis (e.g. [4], [17], [3], [7]). The main reason for this is that one wants to be able to study termination properties of programs for queries containing partially instantiated terms. Most of the earlier work in this area (e.g. [16], [13]) was restricted to queries with ground input arguments and relied on mode information to specify the queries of interest and the way the input data propagates throughout the computation.

Depending on the actual strategy of the termination analysis there could be various reasons and proposed functionalities for integrating type information. But, the following two very related purposes stand out :

*supported by GOA, "Non-standard applications of abstract interpretation", DPWB, Belgium.

†senior research associate of the Belgian National Fund for Scientific Research.

‡partially supported by "Progetto Finalizzato Sistemi Informatici e Calcolo Parallelo" of CNR under grant 89.00026.69.

1) By providing accurate information on the structure of terms occurring in derivations, types allow to select refined ways to measure the size of these terms. Functions from terms to natural numbers, interpreted as measuring the size of each term, are referred to as *norms* and form the basis of the well-founded orderings used to prove termination in many techniques (e.g. [15], [13], [14], [20], [2], [18], [6], [16]).

2) Types allow to select the norm in such a way that the norm of those terms in the derivations that are taken into account for the well-founded ordering (which can be thought of as the 'input' terms) is guaranteed not to grow unboundedly in size due to subsequent derivation steps. The point here is that, if we adopt a norm counting the number of elements in a list, although the norm of a term $[a, b|x]$, denoted $||[a, b|x]||$, is equal to 2, subsequent derivation steps could instantiate x to $[c, d|y]$, increasing the norm of the (more instantiated) term to 4. If such increases are not delimited by some upperbound, then the value of the term under the norm can hardly be useful to set up a finite well-founded order for the derivation. Such upperbounds are typically derived from the term, the type of all possible further instantiation that can occur on it in future derivation steps and the norm itself.

In [7] we propose a technique for inferring appropriate norms from type information. The inferred norms are slightly more general than the ones mentioned above. In the setting of [7], a *typed norm* is a function which maps every pair (t, τ) , consisting of a term t and a type τ such that t belongs to τ , to a natural number. The type system is that of normal rigid types introduced in [9]. Under a typed norm, a same term can be measured in different ways depending on the type it is considered to belong to. These different type memberships will typically result from the different argument positions of predicates in which that same term may occur.

To illustrate this, consider the following one-level flattening program.

Example 1.1 (flatten)

```
flatten([], []).
flatten([L|Z], U) ← flatten(Z, U'), append(L, U', U).
```

Assume that we know that the queries of interest are $\leftarrow \text{flatten}(t_1, t_2)$, where the type of t_1 is the set of lists of lists of any terms τ_1 , and the type of t_2 is the set of lists of any terms τ_2 . A type inference system (e.g. [9]) allows to derive that any descending call has the same types. Assume that typed norms $||\cdot||_{\tau_1}$ and $||\cdot||_{\tau_2}$ are defined as

$$\begin{aligned} ||[t_1|t_2]||_{\tau_1} &= ||t_2||_{\tau_2} + ||t_1||_{\tau_1} \\ ||t||_{\tau_1} &= 0, \text{ for any other term} \end{aligned}$$

and

$$\begin{aligned} ||[t_1|t_2]||_{\tau_2} &= 1 + ||t_2||_{\tau_2} \\ ||t||_{\tau_2} &= 0, \text{ for any other term} \end{aligned}$$

Clearly, the term $[[1, 2]]$ belongs to the type τ_1 as well as to τ_2 , and $||[[1, 2]]||_{\tau_1} = 2$, while $||[[1, 2]]||_{\tau_2} = 1$. ■

The purpose of allowing the generality of typed norms is both to facilitate the automatic inference of "suitable" norms from type information (see [7]) and to improve the expressivity of *interargument relations*. Here, an interargument relation refers to a relation that holds between the (typed) norms of the terms in different argument positions (measured according to each argument's type) of success instances of a predicate. To

illustrate the latter point, notice that in the flatten example the interargument relation $\|t_1\|_{\tau_1} = \|t_2\|_{\tau_2}$ holds for any success atom $\text{flatten}(t_1, t_2)$. No "non-typed" norm is capable of representing this, since even expressing this relation requires norms that *do* assign different sizes to a same term, depending on the type it is considered to belong to.

The main property of the typed norms inferred in [7] is rigidity. For any term t_1 of type τ_1 and for any substitution $\theta : \|t_1\theta\|_{\tau_1} = \|t_1\|_{\tau_1}$. Note that normal rigid types are substitution closed, so that $\|t_1\theta\|_{\tau_1}$ is well defined. Rigidity of (typed) norms is a conservative way of ensuring that no unbounded growth of the norm of a term can occur due to further derivation steps. In general, it is also a practical way of ensuring it, since for typed norms it can be syntactically verified (see [7]). In fact, this syntactic verification forms the key criterion for inferring the norms from the given types.

As argued in [7], typed norms can easily be integrated into existing termination analysis techniques, such as [6], [20], [2]. Only one aspect of the termination analysis is an exception in this respect : the inference of the interargument relations. As we reported in [7], a rather straightforward extension of the technique proposed in [19] to the case of typed norms will be sound, but the loss in precision is most often unacceptable. We refer to subsection 2.3 for a discussion of the problems.

In this paper, we propose an automated technique for inferring interargument relations for typed norms. As in [19], 1) we infer systems of linear relations to approximate the interargument relations, and 2) we formulate the technique as an instance of an abstract interpretation framework. A main distinction with respect to [19] - besides of the focus on typed norms and the improved expressivity this entails - is the use of the bottom-up abstract interpretation framework of [1], which reduces the technical complexity of the analysis. The main advantage of this framework being that we can abstract away from the operational semantics of SLD-resolution, used in top-down abstract interpretation frameworks.

The remainder of the paper is organised as follows. In section 2 we recall some preliminary notions regarding normal rigid types and typed norms, termination analysis and interargument relations. In section 3 we introduce a notion of a *back-propagated success-typed* program, which is a key notion for overcoming the problems in inferring interargument relations for typed norms. Section 4 presents the abstract domain and the abstract analysis. In section 5 we present a detailed example. Finally, we discuss the merits of the technique and relate it to other work.

2 Preliminaries

2.1 (labelled) rigid types

The type formalism chosen in this paper is the one of [9]. The reasons for this choice are 1) rigid types provide sufficient precision for the application, 2) automatic inference of rigid types through abstract interpretation has been fully developed and described in [9], 3) the tools for inferring them are available at our site, allowing for extensive experiments with the proposed techniques. Due to space restrictions and since rigid types are not part of the contribution of this paper, we refer to [9] and [12] for the underlying intuitions. Here, we only recall the basic definitions and give an example.

There exist a number of *primitive types* (e.g. INT, REAL), which represent subsets of the set of constants in the language. We denote the set of all primitive types by \mathcal{P} , and we

assume that there exists a function $Denote : \mathcal{P} \rightarrow 2^{Const_P}$, mapping each primitive type to a corresponding set of constants. We use $Const_P$ and Fun_P to denote respectively the set of constants and the set of function symbols of P .

Rigid types are formally defined by means of *type graphs*, which are a particular instance of directed graphs. We assume that the reader is familiar with the basics of graph theory.

Definition 2.1 (labelled rigid type graph; adapted from [9])

A *labelled rigid type graph* T is a 6-tuple, $(Nodes, ForArcs, BackArcs, Label, ArgPos, TypeLabel)$, where

1. $Nodes$ is a finite, non-empty set of nodes,
2. $ForArcs \subseteq Nodes \times Nodes$ such that $(Nodes, ForArcs)$ is a *tree*,
3. $BackArcs \subseteq Nodes \times Nodes$ such that for each arc $(m, n) \in BackArcs$, node n is an ancestor of node m in $ForArcs$,
4. $Label$ is a function $Nodes \rightarrow \mathcal{P} \cup Const_P \cup Fun_P \cup \{MAX, OR\}$, and
5. $ArgPos$ is a function: $\bigcup_{k>0} (\{m \in Nodes \mid Label(m) = f/k \in Fun_P\} \times \{1, \dots, k\}) \rightarrow Nodes \setminus \{root\}$, such that for each $m \in Nodes$, with $Label(m) = f/k$, $ArgPos(m, \cdot) : \{1, \dots, k\} \rightarrow Nodes$ is a bijection from $\{1, \dots, k\}$ onto $\{n \in Nodes \mid (m, n) \in ForArcs \cup BackArcs\}$.
6. $TypeLabel$ is a function $Nodes \rightarrow TypeNames$ and $TypeNames$ is a set of unique identifiers.

Each node n such that $Label(n) = f/k$ has k immediate descendants, each node labelled OR has at least two immediate descendants, and the other nodes have no descendants and are called *terminal nodes*. Descendants of OR-nodes or functor-nodes are found using the *Desc* function. ■

Definition 2.2

$Desc : \{n \in Nodes \mid Label(n) \in Fun_P \cup \{OR\}\} \rightarrow 2^{Nodes} : Desc(n) = \{n' \mid (n, n') \in ForArcs \cup BackArcs\}$. ■

A rigid type graph T describes a (possibly infinite) set of finite terms. This set of finite terms is found by means of the denotation function, \mathcal{D} . We refer to ([12], Def. 2.3.4 and following) for formal definition.

In [7], the function *TypeLabel* was introduced to regard a rigid type graph as a specification of a set of rigid types : every node in the type graph can be considered as the root of a new type graph, defined by the subgraph rooted at that node. In the sequel we will use rigid types as a shorthand for labelled rigid types.

The next definition enables us to associate sets of terms to *TypeLabels*.

Definition 2.3 (denotation of a TypeLabel)

Let $L = TypeLabel(n)$ be the *TypeLabel* of a node n in a labelled type graph T . We extend the notion of denotation to *TypeLabels* as : $\mathcal{L}(L) = \mathcal{D}(n)$. ■

A labelled rigid type is now presented by means of its graphical representation. In figure 1 the set of all nil-terminated lists which take nil-terminated lists of any terms as elements is depicted as a type graph. It could be used to specify the call type of the first argument of the flatten predicate in Example 1.1.

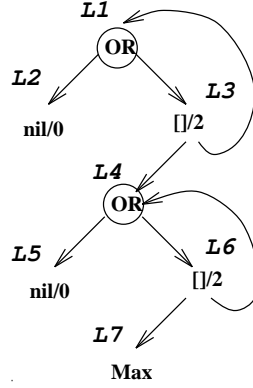


Figure 1: An example of a labelled graph

In [9], rigid type graphs are used as a component of an abstract domain. For reasons of efficiency, rigid type graphs must be further restricted to normal type graphs.

Definition 2.4 (principal nodes; see [9]: definition 4.5)

Let $T=(Nodes, ForArcs, BackArcs, Label, ArgPos, TypeLabel)$ be a rigid type graph. The *principal nodes* of a node $n \in Nodes$ are found by means of the function $Principal : Nodes \rightarrow 2^{Nodes}$:

$$\begin{aligned}
 Principal(n) &= \bigcup_{n' \in Desc(n)} Principal(n') && \text{if } Label(n) = OR \\
 Principal(n) &= \{n\} && \text{otherwise.}
 \end{aligned}$$

■

The main limitation imposed by *normal* type graphs is that principal nodes of any given node should have distinct function symbols. Note that the type in figure 1 satisfies this normality criterion.

2.2 Typed norms

OR-nodes define a type whose denotation is the union of the denotation of some other types. Moreover, if we work with normal type graphs this is a union of disjoint sets of terms. Under the assumption that we deal with normal types and given such a term, the graph enables us to identify in an unambiguous way its TypeLabel. The next definition collects for some OR-node all TypeLabels which are relevant (in the above sense) for it.

Definition 2.5 (Successor type label set of an OR-node)

Let T be a labelled type graph and n a node in T such that $label(n) = OR$. The *successor type label set of n* , $succ_label(n)$ is defined as

$$succ_label(n) = \{L \mid \exists (n, m) \in ForArcs \cup BackArcs \wedge m' \in Principal(m) \wedge TypeLabel(m') = L\}.$$

■

Given a normal type T , in [7] the authors faced the problem of deriving a norm which makes all terms in the denotation of the type rigid.

2.3 Termination analysis and interargument relations

Although typed norms offer higher precision in measuring sizes of terms, they have the disadvantage of complicating the termination analysis. In particular, inference of interargument relations needs special consideration. One may intuitively expect complications, because norms form the basic building blocks to associate well-founded orders to derivations in termination proofs. So, measuring terms of different types in different ways (especially, measuring a same term differently depending on the considered type) can be expected to obscure the termination argument.

In this paper we follow the termination analysis framework of [6]. This is not a limitation in the sense that most of the results in the following sections can easily be integrated in other frameworks, such as e.g. [2]. To simplify the discussion, we recall the key notions of [6] in the context of directly recursive programs only. In what follows, \sim denotes the variant relation on both atoms and terms.

Definition 2.10 (call set)

Let P be a definite program and S a set of atomic queries with a fixed predicate symbol. The *call set*, $Call(P, S)$, is the set of all atoms A such that a variant of A is a selected atom in some derivation for (P, Q) , for some $Q \in S$ and under the left-to-right selection rule. ■

In our setting we will assume that S is specified in terms of the predicate symbol p/n and an n -tuple of normal rigid call types (T_1, \dots, T_n) for its argument. In other words, $S = \{\leftarrow p(t_1, \dots, t_n) \mid t_i \in \mathcal{D}(T_i)\}$ for some given T_1, \dots, T_n . Furthermore, we assume that an upper approximation of $Call(P, S)$ is provided using the type inference of [9]. Thus, for each predicate q/m in P , we assume that an m -tuple of normal rigid types (S_1, \dots, S_m) has been inferred, such that if $q(s_1, \dots, s_m) \in Call(P, S)$, then $s_i \in \mathcal{D}(S_i), \forall i = 1, \dots, m$.

Definition 2.11 (level mapping)

A level mapping is a function $f : Call(P, S)/\sim \rightarrow \mathbb{N}$. ■

In this paper we only consider level mappings defined through the following scheme. For any predicate q/m in P , let (S_1, \dots, S_m) be the associated call types generated by the type inference. Let $\|\cdot\|_{S_1}, \dots, \|\cdot\|_{S_m}$ be any sequence of associated typed norms generated for S_1, \dots, S_m following [7]. Then for any $q(s_1, \dots, s_m) \in Call(P, S)$, $f(q(s_1, \dots, s_m))$ is some positive linear combination of $\|s_1\|_{S_1}, \dots, \|s_m\|_{S_m}$, where the coefficients depend only on q/m .

Definition 2.12 (acceptability wrt S)

Let S be a set of atomic queries with a fixed predicate and P a definite directly recursive program. P is *acceptable wrt S* if there exists a level mapping f and a model I for P , such that

- for any $A \in Call(P, S)$
- for any clause $A' \leftarrow B_1, \dots, B_n$ in P , such that $mgu(A, A') = \theta$ exists,
- for any atom B_i , such that $I \models B_j\theta, 1 \geq j > i$ and having the same predicate symbol as A :

$$f(A) > f(B_i\theta) \quad \blacksquare$$

The following proposition is one of the key results in [6].

Proposition 2.13 A directly recursive program P terminates for any query in S if and only if P is acceptable wrt S . \blacksquare

From the above it becomes clear that the extension from norms to typed norms has no impact on the termination conditions as such. The reason is that, although different terms may be measured using different norms depending on their types, the two atoms A and $B_i\theta$ in the acceptability condition are measured in the same way. This is because they have the same predicate, there is only one tuple of call types associated to each predicate and the level mapping is defined as a fixed combination of the typed norms associated to the call types. Thus, the call and descending recursive call are measured in the same way, and Definition 2.12 remains to capture well-founding and termination.

The problem of providing the model I in the definition of acceptability is a quite different matter. In [6] this is done by computing an interargument relation for the intermediate atoms B_1, \dots, B_{i-1} . This in turn is achieved through abstract interpretation, as discussed in [18]. In the setting of typed norms the notion of an interargument relation needs to be specialised wrt the types of the argument positions. In the following definition, we assume that for any type T a corresponding typed norm $\|\cdot\|_T$ generated as in [7] has been fixed.

Definition 2.14 (interargument relation wrt a tuple of types)

Let p/n be a predicate of P and (T_1, \dots, T_n) an n -tuple of normal rigid types. An interargument relation for p/n wrt (T_1, \dots, T_n) is a relation $R \subseteq \mathcal{N}^n$ such that for any $p(t_1, \dots, t_n)$ satisfying $P \models p(t_1, \dots, t_n)$ and $t_i \in \mathcal{D}(T_i), \forall i = 1, \dots, n$, $R(\|t_1\|_{T_1}, \dots, \|t_n\|_{T_n})$ holds. \blacksquare

Notice that the notion is more specific than that of an interargument relation in the sense that the relation is not required to hold for the entire success set for p in P . In particular, given a clause

$$q(s_1, \dots, s_m) \leftarrow p(t_1, \dots, t_n), (1) r(u_1, \dots, u_k), (2) q(s'_1, \dots, s'_m)$$

we will usually be interested in an interargument relation for p/n and r/k which holds wrt the types that the arguments of these predicates have when point 2 in the derivation is reached. Notice that these types may be more specific than the success types for p/n in general in P . The reasons are : 1. The clause may possibly be activated with a specific call type, 2. the success substitutions for r/k may further specialise the success types of p/n .

One of the main problems we now need to solve is to produce a set of type tuples, as input for the definition of the interargument relation, that is sufficiently precise and consistent to provide a useful model for definition 2.12. Here, the following considerations are important. We illustrate them with the above example.

1. A simple solution would be to apply the type inference of [9] and to collect all success types that hold at execution point 2 for the intermediate atoms (p/n and

r/k) and for all other predicates on which these depend in P . The problem is that, given an additional clause

$$p(t'_1, \dots, t'_n) \leftarrow s(v_1, \dots, v_l) \quad (3)$$

of P , a method such as the one in [9] will only provide success types for s/l which hold locally at point 3, without incorporating the additional instantiations caused on $s(v_1, \dots, v_l)$ by solving $r(u_1, \dots, u_k)$ (at point 2). In theory, this is no problem, in the sense that the success types at point 3 will provide a correct interargument relation for s/l . However, the interargument relations for p/n and r/k (at point 2) need to be computed on the basis of the interargument relations for the predicates on which they depend in P . The fact that the latter relations are computed with respect to quite different success types (e.g. those that hold at point 3) causes enormous consistency problems : the same terms are measured with respect to different norms, making propagation of relations over different clause levels virtually impossible.

2. An alternative is to use the bottom-up type inference of [1]. However, here the problem is that such an approach does not easily allow to take the call information, with which a clause is entered, into account.
3. A final option, and essentially the solution we take in the remainder of the paper, is to use the approach of [9] augmented with a reexecution strategy (see e.g. [11]). This means that, once success types for p/n have been computed, the type inference is restarted using the previously obtained success types as the call types for p/n . By doing so, instantiations caused in the final part of a derivation are back-propagated onto previous calls.

3 Back-propagated success-typed programs

In this section we associate to a given definite program P , a predicate symbol p/n of P and an n -tuple $\bar{T} = (T_1, \dots, T_n)$ of normal rigid types a type-annotated program, $P_{p, \bar{T}}$. In light of the discussion ending the previous section, p/n fulfils the role of either p/n or r/k in the example clause, and \bar{T} that of the types of the arguments of p/n (or r/k) at execution point 2. Intuitively, $P_{p, \bar{T}}$ annotates all predicates of P that depend on p/n with a tuple of success types that a type inference system augmented with reexecution would assign to them. As such, all success type annotations will be consistent with the types \bar{T} , in the sense that they take into account any further instantiations produced between the successful refutation of the atom and execution point 2.

We will refer to such programs $P_{p, \bar{T}}$ as *back-propagated success-typed programs*, which reflects this intuition.

Definition 3.1 (type-annotated atom)

For any atom $A = p(t_1, \dots, t_n)$ and n -tuple of types $\bar{T} = (T_1, \dots, T_n)$, the *type-annotated atom* associated to A and \bar{T} is $A^{\bar{T}} = p^{T_1, \dots, T_n}(t_1, \dots, t_n)$. ■

Definition 3.2 (well-typed atom)

A *well-typed atom* is a type-annotated atom $p^{T_1, \dots, T_n}(t_1, \dots, t_n)$ such that $t_i \in \mathcal{D}(T_i), \forall i = 1, \dots, n$. ■

In the sequel we assume that our language is extended with a set of extra built-in predicates, $=^{T,T}$, one for each normal rigid type T . On the level of logic, one may think of these predicates as simple *renamed* versions of the usual unification without any added functionality. The added functionality will only turn up on the level of the abstract interpretation.

Definition 3.3 (type-annotated clause)

For any clause $C : A_0 \leftarrow A_1, \dots, A_n$, such that A_i has arity m_i , $\forall i = 0, \dots, n$, and for any $(n + 1)$ -tuple of tuples of types, $(\bar{T}_0, \dots, \bar{T}_n)$, where $\bar{T}_i = (T_i^1, \dots, T_i^{m_i})$, $i = 0, \dots, n$, $A_0^{\bar{T}_0} \leftarrow A_1^{\bar{T}_1}, \dots, A_n^{\bar{T}_n}$ is a *type-annotated clause*. ■

Definition 3.4 (success-typed clause)

Let P be a definite program, $C : p(t_1, \dots, t_m) \leftarrow A_1, \dots, A_n$ a clause in P and $\bar{T} = (T_1, \dots, T_m)$ an m -tuple of types. A *success-typed clause* associated to P, C and \bar{T} is a *type-annotated clause*

$$p^{T_1, \dots, T_m}(t_1, \dots, t_m) \leftarrow A_1^{\bar{T}_1}, \dots, A_n^{\bar{T}_n}$$

such that for every substitution σ satisfying $p^{T_1, \dots, T_m}(t_1, \dots, t_m)\sigma$ is a well-typed atom and $P \models (A_1, \dots, A_n)\sigma$, we have that $A_i^{\bar{T}_i}\sigma$ is a well-typed atom, $\forall i = 1, \dots, n$. ■

Definition 3.5 (failed type-annotated clause)

A *failed type-annotated clause* is a type-annotated clause

$$A_0^{\bar{T}_0} \leftarrow A_1^{\bar{T}_1}, \dots, A_n^{\bar{T}_n}$$

such that there exists an i , $1 \leq i \leq n$, and a T_i^j in \bar{T}_i , with $T_i^j = \perp$. ■

Definition 3.6 (back-propagated success-typed program)

Let P be a definite program, p a predicate symbol of P with arity m and $\bar{T} = (T_1, \dots, T_m)$ an m -tuple of types. A *back-propagated success-typed program* associated to P, p and \bar{T} is a definite program $P_{p, \bar{T}}$ such that :

1. for each clause $C : p(t_1, \dots, t_m) \leftarrow A_1, \dots, A_n$ in P either there exists a failed success-typed clause associated to P, C and \bar{T} , or $P_{p, \bar{T}}$ contains a unique success-typed clause associated to P, C and \bar{T} .
2. for every predicate Q^{S_1, \dots, S_r} in $P_{p, \bar{T}}$, different from $=^{S_1, S_1}$, $P_{p, \bar{T}}$ is a back-propagated success-typed program associated to P, q and (S_1, \dots, S_r) . ■

Example 3.7 (permute)

The following program (in normal form) is a back-propagated success-typed program

associated to the well-known permute (with delete) program, permute/2 and (τ_2, τ_2) . We denote by

$$\begin{aligned}
\text{permute}^{\tau_2, \tau_2}(X, Y) &\leftarrow X =^{\tau_2, \tau_2} [], Y =^{\tau_2, \tau_2} []. \\
\text{permute}^{\tau_2, \tau_2}(X, Y) &\leftarrow Y =^{\tau_2, \tau_2} [E|L], \\
&\quad \text{delete}^{\tau_2, MAX, \tau_2}(X, E, R), \\
&\quad \text{permute}^{\tau_2, \tau_2}(R, L). \\
\text{delete}^{\tau_2, MAX, \tau_2}(Y, X, L) &\leftarrow Y =^{\tau_2, \tau_2} [X|L]. \\
\text{delete}^{\tau_2, MAX, \tau_2}(Y, X, L) &\leftarrow Y =^{\tau_2, \tau_2} [H|T_1], \\
&\quad L =^{\tau_2, \tau_2} [H|T_2], \\
&\quad \text{delete}^{\tau_2, MAX, \tau_2}(T_1, X, T_2).
\end{aligned}$$

■

Essentially, a back-propagated success-typed program (in short, success-typed program) $P_{p, \bar{T}}$ associated to P, p and \bar{T} has essentially the same clauses as P , except that : 1) $P_{p, \bar{T}}$ may have multiple annotated versions of the same clause in P , 2) all explicit unifications are replaced by typed unifications, 3) some clauses of P which lead to failure for the considered types may have no counterpart in $P_{p, \bar{T}}$.

Due to this relation between P and $P_{p, \bar{T}}$, we have the following proposition :

Proposition 3.8 (completeness of success-typed programs)

Let P, p and \bar{T} be as above and let $P_{p, \bar{T}}$ be an associated success-typed program. For any atom $p(t_1, \dots, t_m)$ such that $P \models p(t_1, \dots, t_m)$ and $t_i \in \mathcal{D}(T_i), i = 1, \dots, m$, we have $P_{p, \bar{T}} \models p^{T_1, \dots, T_m}(t_1, \dots, t_m)$. ■

This property is crucial in the next section. Here, the inference of an interargument relation for p wrt \bar{T} in P will be performed by inferring an interargument relation for p^{T_1, \dots, T_m} wrt \bar{T} in $P_{p, \bar{T}}$. Proposition 3.8 states that the latter interargument relation is also an interargument relation for p wrt \bar{T} in P .

Throughout the remainder of the paper we assume that the program P is in normal form (see e.g. [9]). On the highest level this means that all unifications are made explicit using the unification builtin $=/2$. Furthermore, all arguments of atoms with user defined predicates are variables and, for each clause, all such variables in the clause are mutually distinct. Non-variable terms may only occur at the right-hand side of $=/2$ and can only be of the type $f(X_1, \dots, X_n), n \geq 0$ (0 corresponds to a constant), with X_1, \dots, X_n distinct variables. Any program can be transformed into an equivalent program in normal form. Note that if P is in normal form, then $P_{p, \bar{T}}$ is in normal form as well, (remember that we do not regard typed unification as different from the usual explicit unification), for any p and \bar{T} .

We conclude the section with one further refinement on the notion of a success-typed program. In essence, its purpose is to avoid that multiple occurrences of a same variable in a clause get different annotations.

Definition 3.9 (consistently, success-typed program)

Let P be a program in normal form, p/m a predicate in P and \bar{T} an m -tuple of types. A success-typed program $P_{p, \bar{T}}$ associated to P, p and \bar{T} is *consistent* if for each clause C in P and for each variable X occurring in C there exists a unique type T such that :

- if X occurs as the i -th argument of an atom $q^{S_1, \dots, S_r}(t_1, \dots, t_r)$, then $S_i = T$,
- if X occurs as the i -th argument of $f(t_1, \dots, t_r)$ in an atom $Y =^{S, S} f(t_1, \dots, t_r)$, then S is a union type with subtype $f(S_1, \dots, S_r)$ and $S_i = T$.

■

The success-typed program in Example 3.7 is consistent.

4 Derivation of linear interargument relations with respect to typed norms

For this section, we assume that P is a consistently success-typed program and that n predicates (different from the annotated unification) are defined in it. Subsection 4.4 describes how to compute bottom-up for each of these predicates a linear interargument relation with respect to the types with which they are adorned. First we provide a solid base for the procedure by describing the concrete and abstract domain, abstraction (and concretisation) function and some useful operations.

4.1 Concrete domain

Obviously the concrete domain consists of sets of annotated atoms. All atoms in such a set obey their annotations, i.e. the atoms are well-typed. We can equip D with a partial order relation \subseteq defined as the subset relation on sets.

4.2 Abstract domain

Analogously to [19], we abstract size relations as systems of linear equations. A linear equation is an equation of the form $a_1 X_1 + \dots + a_m X_m = b$ where $a_i, b \in \mathbb{Q}, \forall i \in \{1, \dots, m\}$. Geometrically interpreted, (the set of solutions of) such an equation corresponds to a hyperplane of dimension $m - 1$ in m -dimensional space. Then, a system of such equations corresponds to the intersection of the hyperplanes associated to each equation. Such systems can be represented by a matrix of the form $\bar{A} \cdot \bar{X} = \bar{c}$, where \bar{A} is an $m \times n$ matrix of real numbers, \bar{X} is an $n \times 1$ matrix of variables and \bar{c} is a $1 \times m$ matrix of real numbers. Because different systems can have the same set of solutions, thus can denote the same hyperplane, a canonical form is introduced: the so-called *row echelon form* of a system. A system of equations $\bar{A} \cdot \bar{X} = \bar{c}$ is in row echelon form if its augmented matrix (i.e. the matrix $[\bar{A} | \bar{c}]$) is in reduced row echelon form. Three conditions must be satisfied for such a matrix $[\bar{A} | \bar{c}] = [a_{ij}]$: 1) the first non-zero entry in a row is 1, 2) for any row i_0 let j_0 be the first column with a non-zero entry, then for all $i > i_0, j \leq j_0, a_{ij} = 0$, 3) for any row i_0 , let j_0 be the first non-zero entry, then for all $i < i_0, a_{ij_0} = 0$.

This form can always be obtained by repeated application of one of three basic operations: multiplication of a row by a scalar, addition of one row to another and permutation of two rows.

Because there exist multiple ways to represent unsolvable systems, we adopt the convention to denote such a system by the symbol \perp . We refer to [19] for more details.

In the abstract interpretation procedure, the following operations on systems of equations will occur frequently.

- **Intersection.** Obviously, a tuple is a solution of the intersection of two systems if it is a solution of both systems. Thus, the intersection of two systems of equations corresponds to the set of solutions of the combined systems. Computing the intersection of two systems $\bar{A}_1 \cdot \bar{X} = \bar{c}_1$ and $\bar{A}_2 \cdot \bar{X} = \bar{c}_2$ corresponds to reducing to row echelon form the augmented matrix

$$\left[\begin{array}{c|c} \bar{A}_1 & \bar{c}_1 \\ \bar{A}_2 & \bar{c}_2 \end{array} \right]$$

- **Disjunction.** Given two systems of equations S_1 and S_2 , their disjunction is a new system which has as solutions the solutions of both systems. Of course, the union of two hyperplanes is very seldom a new hyperplane. We refer to [10] for the presentation of a technique which computes the most precise system of equations whose solution set comprises the two given hyperplanes.
- **Restriction.** Let \bar{A} be an $m \times n$ -matrix and let \bar{X} be the transposition of $[X_1, \dots, X_n]$. The restriction of a system $\bar{A} \cdot \bar{X} = \bar{c}$ to variables X_{k+1}, \dots, X_n is a system $\bar{A}_r \cdot \bar{X}_r = \bar{c}_r$ (\bar{A}_r has dimension $m' \times (n - k)$) such that

$$\begin{aligned} \exists x_1, \dots, x_n \in \mathbb{R} : (x_1, \dots, x_k, x_{k+1}, \dots, x_n) \text{ is a solution of } \bar{A} \cdot \bar{X} = \bar{c} \\ \Downarrow \\ (x_{k+1}, \dots, x_n) \text{ is a solution of } \bar{A}_r \cdot \bar{X}_r = \bar{c}_r \end{aligned}$$

[19] describes how to compute this operation for a system in reduced row-echelon form.

- **Extension.** This is the converse of the restriction operation. The operation maps a system $\bar{A} \cdot \bar{X} = \bar{c}$ to a system $\bar{A}_e \cdot \bar{X}_e = \bar{c}_e$ such that

$$\begin{aligned} (x_1, \dots, x_k) \text{ is a solution of } \bar{A} \cdot \bar{X} = \bar{c} \\ \Downarrow \\ \forall x_{k+1}, \dots, x_n \in \mathbb{R} : (x_1, \dots, x_k, x_{k+1}, \dots, x_n) \text{ is a solution of } \bar{A}_e \cdot \bar{X}_e = \bar{c}_e \end{aligned}$$

In practice, this is performed by adding $(n - k)$ columns of 0's to \bar{A} and adding k rows of a single 0 to \bar{c} .

A partial order can be established on these systems of linear equations as follows: Let $S_1 = \bar{A}_1 \cdot \bar{X} = \bar{c}_1$ and $S_2 = \bar{A}_2 \cdot \bar{X} = \bar{c}_2$. Then $S_1 \leq S_2$ iff $S_1 = \perp$, or $S_2 = \top$, or the intersection of S_1 and S_2 is S_1 . We have $S_1 \equiv S_2$ iff $S_1 \leq S_2$ and $S_2 \leq S_1$. Obviously, \perp and \top are minimal and maximal for this order.

Two extremely useful operations on systems with the same domain, are the *least upper bound* and the *greatest lower bound* operation. They are defined as follows. Let S, S_1 and S_2 denote systems of linear equations with $S_1, S_2 \notin \{\perp, \top\}$:

- $\text{lub}(S, \top) = \text{lub}(\top, S) = \top$.
- $\text{lub}(S, \perp) = \text{lub}(\perp, S) = S$.
- $\text{lub}(S_1, S_2) = \text{disjunct}(S_1, S_2)$.

The greatest lower bound is defined as:

- $\text{glb}(S, \top) = \text{glb}(\top, S) = S$.
- $\text{glb}(S, \perp) = \text{glb}(\perp, S) = \perp$.
- $\text{glb}(S_1, S_2) = \text{intersect}(S_1, S_2)$.

As [19] indicates, there do not exist infinitely ascending chains of systems $S_0 < S_1 < \dots$. The maximum length of a strictly ascending chain is equal to the dimension of the domain.

We are now in a position to specify our abstract domain. If we denote the set of all possible systems of linear equations over k variables by E_q^k , then we fix our abstract domain to be $D^\alpha = (E_q^{k_1}, E_q^{k_2}, \dots, E_q^{k_n})$, for some $k_1, \dots, k_n \in \mathbb{N}$, i.e. elements of our abstract domain correspond to n -tuples of systems of linear equations of fixed dimensions. The order on systems of equations naturally induces an order relation \sqsubseteq on our abstract domain: $\forall \langle S_1, \dots, S_n \rangle, \langle S'_1, \dots, S'_n \rangle \in D^\alpha : \langle S_1, \dots, S_n \rangle \sqsubseteq \langle S'_1, \dots, S'_n \rangle$ iff $S_i \leq S'_i, \forall i \in \{1, \dots, n\}$. Similarly, we can lift the equivalence \equiv on systems of equations to this domain.

Proposition 4.1 D^α is a complete lattice with as bottom element $\langle \perp, \dots, \perp \rangle$ and as top element $\langle \top, \dots, \top \rangle$. ■

4.3 Abstraction and concretisation function

We first restrict our attention to a single system of equations. Let S be a set of atoms with predicate symbol p^{T_1, \dots, T_m} . We define the abstraction of $S, \dot{\alpha}(S)$, to be a representative of the least \equiv -equivalence class of E_q^m having all tuples $(\|t_1\|_{T_1}, \dots, \|t_m\|_{T_m})$, $p^{T_1, \dots, T_m}(t_1, \dots, t_m) \in S$ as solutions. This also fixes the concretisation function: given a set of linear equations S over m variables and a predicate p^{T_1, \dots, T_m} , then the concretisation for that predicate, $\dot{\gamma}(S, p^{T_1, \dots, T_m}) = \{p^{T_1, \dots, T_m}(t_1, \dots, t_m) \mid (\|t_1\|_{T_1}, \dots, \|t_m\|_{T_m}) \text{ is a solution of } S \text{ and } t_i \in \mathcal{D}(T_i)\}$. It can easily be proved that $S_1 \leq S_2$ iff $\dot{\gamma}(S_1, p) \subseteq \dot{\gamma}(S_2, p)$, for any p with appropriate arity.

In general, the abstraction function is defined as follows. Let us first fix an order on the predicate symbols of the program by the use of a bijection $\nu : \text{Pred}(P) \rightarrow \{1, \dots, n\}$, where $\text{Pred}(P)$ is the set of all predicate symbols defined in P (annotated unification excluded). We assume that the arity of $\nu^{-1}(i)$ is $k_i, \forall i \in \{1, \dots, n\}$. Given a set of atoms $A = \cup_{i=1}^n A_i$, where A_i is a set of atoms with predicate symbol p^{T_1, \dots, T_m} and $\nu(p^{T_1, \dots, T_m}) = i$, then, $\alpha(A) = \langle S_1, \dots, S_n \rangle$ where $S_i = \dot{\alpha}(A_i)$. This definition also allows to fix our concretisation function $\gamma: \gamma(\langle S_1, \dots, S_n \rangle) = \cup_{i=1}^n \dot{\gamma}(S_i, \nu^{-1}(i))$.

Proposition 4.2 The partial orders \subseteq on D and \sqsubseteq on D^α and the concretisation function (abstraction function) establish a Galois connection between both domains. ■

Proposition 4.3 The abstract domain does not contain infinitely ascending chains. ■

Proof Trivial, because there does not exist an infinite sequence of systems of linear equations $E_0 \sqsubset E_1 \sqsubset \dots$. The maximal length of strictly ascending tuples of systems is bounded by the sum of the number of variables in each system. ■

4.4 Bottom-up abstract interpretation procedure

The goal of abstract interpretation is to approximate the semantics of programs. In this paper we consider the T_P semantics and we adopt the bottom-up abstract interpretation of [1]. The basic operation is the abstract interpretation of unification. In [19], using a top-down framework, given a call to unification, $X = t$, and an abstract call substitution, E_q^k (a system of linear equations) the abstract success substitution is $E_q^k \cup \{X = \text{abs}_{||\cdot||}(t)\}$. Here, $\text{abs}_{||\cdot||}$ is called the abstract norm and it is a function which maps each term t to a linear arithmetic expression, which can best be described as a partial evaluation of the concrete norm on the term. In our setting, the notion is generalised to type-annotated unification and typed norms as follows:

Definition 4.4 (size expression induced by a typed norm $||\cdot||_L$)

Let T be a labelled normal type, let $L = \text{TypeLabel}(n)$ be the TypeLabel of a node n in T , and $||\cdot||_L$ the associated norm. A term t is mapped to the *size expression* $\text{abs}_L(t)$ induced by the norm $||\cdot||_L$ by means of the following function $\text{abs}_L : \text{Term}_P / \sim \rightarrow \mathcal{L}_{<0,1;+;\leq}$ defined as

$$\begin{array}{ll} \text{if } \text{label}(n) = OR & \text{then } \text{abs}_L(X) = X \\ & \text{otherwise } \text{abs}_L(t) = \sum_{L_i \in \text{SuccLabel}(n)} \text{abs}_{L_i}(t) \\ \text{else if } \text{label}(n) = f/k & \text{then } \text{abs}_L(X) = X \\ & \text{abs}_L(f(t_1, \dots, t_k)) = c_{f,L} + \sum_{i=1}^n \text{abs}_{L_i}(t_i) \\ & \quad \text{with } L_i = \text{TypeLabel}(\text{ArgPos}(n, i)) \\ & \text{otherwise } \text{abs}_L(t) = 0 \\ \text{else } & \text{abs}_L(t) = 0 \end{array}$$

■

We use $\text{abs}_T(t)$ as a shorthand for $\text{abs}_L(t)$ for a type T with root r and $\text{TypeLabel}(r) = L$.

The abstract T_P^α operator is defined in three stages. First, we introduce an auxiliary function \mathcal{A} which, given an atom (either user defined or unification) and the currently obtained element E of the abstract domain, selects from E the information relevant to this atom.

Definition 4.5 \mathcal{A}

Let p^T/k be a predicate defined in P and let $E \in D^\alpha$ be a vector of systems of equations, one system for each of the n predicates occurring in P . By $E[p^T]$ we denote the system of equations corresponding to p^T/k . Then the operator \mathcal{A} abstracts atoms as follows

$$\mathcal{A} : \bar{B}_P \times D^\alpha \rightarrow \cup_{i \in \mathbb{N}} E_q^i :$$

$$\mathcal{A}(X =^{T,T} t, E) = \{X = \text{abs}_T(t)\}$$

$$\mathcal{A}(p^T(X_1, \dots, X_k), E) = E[p^T]\theta \text{ where } \theta \text{ is a renaming of the variables in } E[p^T] \text{ according to } X_1, \dots, X_k$$

■

Next we define a T_P^α operator which, for each $E \in D^\alpha$, computes the separate entries in the final n -tuple, $T_P^\alpha(E) \in D^\alpha$. More in particular, $T_P^\alpha : \text{Pred}(P) \times D^\alpha \rightarrow \cup_{i=1}^n E_q^{k_i}$, with $T_P^\alpha(\nu^{-1}(i), E) \in E_q^{k_i}$.

Definition 4.6 T_P^α

Let $p^{T_1, \dots, T_k} \in \text{Pred}(P)$ and assume that p^{T_1, \dots, T_k} is defined by m clauses C_1, \dots, C_m in P . We further assume that the heads of all clauses $C_i, i = 1, \dots, m$ are identical, i.e.

$p^{T_1, \dots, T_k}(X_1, \dots, X_k)$ for a fixed tuple of variables (X_1, \dots, X_k) . Finally, let $(Z_1^i, \dots, Z_{l_i}^i) = (X_1, \dots, X_k, Y_{k+1}^i, \dots, Y_{l_i}^i)$ be the tuple of all variables occurring in C_i , $i = 1, \dots, m$, where $(Y_{k+1}^i, \dots, Y_{l_i}^i)$ are the local variables.

$$T_P^\alpha(p^T, E) = \text{lub}(\mathcal{A}(p^T(X_1, \dots, X_k), E), \text{lub}_{i=1, \dots, m}(E_i))$$

where

$$E_i = \text{restrict}((X_1, \dots, X_k), E_{\text{Body}_i})$$

$$E_{\text{Body}_i} = \text{glb}_{B \in \text{Body}(C_i)}(E_B)$$

$$E_B = \text{extend}((Z_1^i, \dots, Z_{l_i}^i), \mathcal{A}(B, E)).$$

■

Lemma 4.7 $\mathcal{A}(p^T(X_1, \dots, X_k), E) \leq T_P^\alpha(p^T, E)$.

■

Proof Trivial.

■

From T_P^α , we can now easily define $\mathcal{T}_P^\alpha : D^\alpha \rightarrow D^\alpha$.

Definition 4.8 \mathcal{T}_P^α

$$\mathcal{T}_P^\alpha : D^\alpha \rightarrow D^\alpha : E \mapsto \langle T_P^\alpha(\nu^{-1}(1), E), \dots, T_P^\alpha(\nu^{-1}(n), E) \rangle.$$

■

Lemma 4.9 \mathcal{T}_P^α is monotonic.

■

Proof A proof has been included in the appendix.

■

Definition 4.10 ω -first powers of \mathcal{T}_P^α

The ω -first powers of the operator are defined as usual:

$$\mathcal{T}_P^\alpha \uparrow 0 = \langle \perp, \dots, \perp \rangle.$$

$$\mathcal{T}_P^\alpha \uparrow (i+1) = \mathcal{T}_P^\alpha(\mathcal{T}_P^\alpha \uparrow i).$$

$$\mathcal{T}_P^\alpha \uparrow \omega = \text{lub}_{i \in \mathbb{N}} \mathcal{T}_P^\alpha \uparrow i.$$

■

Proposition 4.11 There exists an $n \in \mathbb{N}$ such that $\mathcal{T}_P^\alpha \uparrow n$ is a least fixpoint.

■

Proof This is an immediate consequence from:

- 1) $E \sqsubseteq \mathcal{T}_P^\alpha(E)$ (because of Lemma 4.7)
- 2) $\mathcal{T}_P^\alpha(E)$ is monotonic
- 3) D^α has no infinite ascending chains.

■

Proposition 4.12 \mathcal{T}_P^α safely approximates the T_P -operator.

■

Proof We again refer to the appendix where the proposition is proved.

■

5 An example

Consider our flatten program of example 1.1 and suppose we would like to derive an interargument relation for flatten with respect to the types τ_1 and τ_2 of the introduction. First we rewrite the flatten program as a consistently success-typed program associated to P , flatten/2 and $\bar{T} = (\tau_1, \tau_2)$ which for simplicity we denote as P_T .

$$\begin{aligned}
\text{flatten}^{\tau_1, \tau_2}(X, Y) &\leftarrow X =^{\tau_1, \tau_1} [], Y =^{\tau_2, \tau_2} []. \\
\text{flatten}^{\tau_1, \tau_2}(X, Y) &\leftarrow X =^{\tau_1, \tau_1} [L|Z], \\
&\quad \text{flatten}^{\tau_1, \tau_2}(Z, U), \\
&\quad \text{append}^{\tau_2, \tau_2, \tau_2}(L, U, Y). \\
\text{append}^{\tau_2, \tau_2, \tau_2}(X, Y, Z) &\leftarrow X =^{\tau_2, \tau_2} [], Y =^{\tau_2, \tau_2} Z. \\
\text{append}^{\tau_2, \tau_2, \tau_2}(X, Y, Z) &\leftarrow X =^{\tau_2, \tau_2} [H|T_1], \\
&\quad Z =^{\tau_2, \tau_2} [H|T_2], \\
&\quad \text{append}^{\tau_2, \tau_2, \tau_2}(T_1, Y, T_2).
\end{aligned}$$

As a convenience, let us number these clauses TC_1 till TC_4 . As P_T defines two predicates with arity 2 and 3 respectively, $T_{P_T}^\alpha$ is defined on $\langle E_q^2, E_q^3 \rangle$ i.e. we adopt the convention that in a couple $\langle S_f, S_a \rangle$, S_f corresponds to a system of equations for $\text{flatten}^{\tau_1, \tau_2}$ and S_a to one for $\text{append}^{\tau_2, \tau_2, \tau_2}$. In order to enhance the readability of this example we do not work with the matrix representation of the systems of equations. Neither do we show where the extension operation comes in.

Then:

$$T_{P_T}^\alpha \uparrow 0 = \langle \perp, \perp \rangle.$$

$T_{P_T}^\alpha \uparrow 1$: Here we have to compute both $T_{P_T}^\alpha(\text{flatten}^{\tau_1, \tau_2}/2, \langle \perp, \perp \rangle)$ and $T_{P_T}^\alpha(\text{append}^{\tau_2, \tau_2, \tau_2}/3, \langle \perp, \perp \rangle)$. Starting with $\text{flatten}^{\tau_1, \tau_2}$, we compute for both of its clauses TC , $(\text{glb}_{B \in \text{Body}(TC)} \mathcal{A}(B, \langle \perp, \perp \rangle))|_{\{X, Y\}}$. $TC1$ gives $\text{glb}(\{X = 0\}, \{Y = 0\})|_{\{X, Y\}} = \{X = 0, Y = 0\}$.

For $TC2$ we obtain $\text{glb}(\{X = L + Z\}, \perp, \perp)|_{\{X, Y\}} = \perp$. Hence $\text{lub}(\perp, \{X = 0, Y = 0\}, \perp) = \{X = 0, Y = 0\}$.

For $TC3$, for $\text{append}^{\tau_2, \tau_2, \tau_2}/3$, $\text{glb}(\{X = 0\}, \{Y = Z\})|_{\{Y, X, Z\}} = \{X = 0, Y = Z\}$ is computed. $TC4$ gives $\text{glb}(\{X = 1 + T_1\}, \{Z = 1 + T_2\}, \perp)|_{\{Y, X, Z\}} = \perp$.

$\text{lub}(\perp, \{X = 0, Y = Z\}, \perp) = \{X = 0, Y = Z\}$.

Thus, $T_{P_T}^\alpha \uparrow 1 = \langle \{X = 0, Y = 0\}, \{X = 0, Y = Z\} \rangle$.

$T_{P_T}^\alpha \uparrow 2$: We again consider all clauses, now making use of the information computed in the previous iteration. For $TC1$, we obtain $\text{glb}(\{X = 0\}, \{Y = 0\})|_{\{X, Y\}} = \{X = 0, Y = 0\}$. $TC2$ gives $\text{glb}(\{X = L + Z\}, \{Z = 0, U = 0\}, \{L = 0, U = Y\})|_{\{X, Y\}} = \{X = 0, Y = 0\}$. Then $T_{P_T}^\alpha(\text{flatten}^{\tau_1, \tau_2}/2, T_{P_T}^\alpha \uparrow 1) = \text{lub}(\{X = 0, Y = 0\}, \{X = 0, Y = 0\}, \{X = 0, Y = 0\}) = \{X = 0, Y = 0\}$.

$TC3$ results in $\text{glb}(\{X = 0, Y = Z\})|_{\{Y, X, Z\}} = \{X = 0, Y = Z\}$ and $TC4$ in $\text{glb}(\{X = 1 + T_1\}, \{Z = 1 + T_2\}, \{T_1 = 0, Y = T_2\})|_{\{Y, X, Z\}} = \{X = 1, Z = 1 + Y\}$. $\text{lub}(\{X = 0, Y = Z\}, \{X = 0, Y = Z\}, \{X = 1, Z = 1 + Y\}) = \{X + Y = Z\}$.

Thus, $T_{P_T}^\alpha \uparrow 2 = \langle \{X = 0, Y = 0\}, \{X + Y = Z\} \rangle$.

$T_{P_T}^\alpha \uparrow 3$: The information of $T_{P_T}^\alpha \uparrow 2$ is now available for computing both $T_{P_T}^\alpha(\text{flatten}^{\tau_1, \tau_2}/2, \langle \{X = 0, Y = 0\}, \{X + Y = Z\} \rangle)$ and $T_{P_T}^\alpha(\text{append}^{\tau_2, \tau_2, \tau_2}/3, \langle \{X = 0, Y = 0\}, \{X + Y = Z\} \rangle)$. This gives the following : for $TC1$: $\text{glb}(\{X = 0\}, \{Y = 0\})|_{\{X, Y\}} =$

$\{X = 0, Y = 0\}$ while *TC2* gives $glb(\{X = L + Z\}, \{Z = 0, U = 0\}, \{L + U = Y\})|_{\{X, Y\}} = \{X = Y\}$. Then $lub(\{X = 0, Y = 0\}, \{X = 0, Y = 0\}, \{X = Y\}) = \{X = Y\}$. For $T_{P_T}^\alpha$ ($\text{append}^{\tau_2, \tau_2, \tau_2} / 3, T_{P_T}^\alpha \uparrow 2$) we look at its two clauses: *TC3* yields $glb(\{X = 0\}, \{Y = Z\})|_{\{X, Y, Z\}} = \{X = 0, Y = Z\}$. From *TC4* we derive $glb(\{X = 1 + T_1\}, \{Z = 1 + T_2\}, \{T_1 + Y = T_2\})|_{\{X, Y, Z\}} = \{X + Y = Z\}$. Thus $upb(\{X + Y = Z\}, \{X = 0, Y = Z\}, \{X + Y = Z\}) = \{X + Y = Z\}$.

As a result, $T_{P_T}^\alpha \uparrow 3 = \langle \{X = Y\}, \{X + Y = Z\} \rangle$.

$T_{P_T}^\alpha \uparrow 4$: This results in no new information, so that the least fixpoint is reached in $T_{P_T}^\alpha \uparrow 3$.

The resulting size relation says that whenever an atom $\text{flatten}(t_1, t_2)$ is a logical consequence of the flatten program, and its arguments satisfy $t_1 \in \mathcal{D}(\tau_1), t_2 \in \mathcal{D}(\tau_2)$, then $\|t_1\|_{\tau_1} = \|t_2\|_{\tau_2}$. We have derived a size relation expressing that both lists contain the same number of basic elements.

6 Conclusions

The contribution of this paper is to fill in the one technical gap needed to extend automatic termination analysis techniques such as the ones in [20] and [17] to be able to take advantage of typed norms: the inference of interargument relations for such norms. At first sight, this can be regarded as a minor, technical point. It should be observed however that, although [20] is one of the more powerful automatic termination techniques available, the method was still based on ordinary norms. In [7], [3] and [4] it has been clearly pointed out that termination analysis for non-ground queries can be significantly improved *only* if type information is taken into account. Furthermore, [7] convincingly shows that types allow for improved automatic inference of norms and that the expressivity of the typed norms is a significant enhancement.

From these observations, we are convinced that any state-of-the-art approach to termination analysis needs to take type information into account and that any future work in this direction will need to address the same problems raised by type-specialised norms that we dealt with in this paper.

On the level of practicality, our approach can deal with any of the programs successfully studied in [17], since typed norms extend non-typed ones. Moreover, practically all the examples mentioned in [7], which fall outside the scope of [17], can successfully be handled by our extension. Flatten is just one representative of this class. Others are the tree-to-list conversion and the change predicate. See [3] for a more extensive list.

In view of full automation, the only component of our approach of which automation was not described in this paper is the generation of consistently success-typed programs. In [8] we give a constructive transformation that assigns a consistently success-typed program to any given program, predicate and tuple of success types. The transformation relies on an abstract interpretation with reexecution.

One might worry that the different layers of abstract interpretation in this approach become - as a whole - computationally unfeasible and needlessly complex. We are currently investigating the problem of merging the different interpretations into a single one-traversal analysis. The main trade-off in this attempt is to avoid too much decrease in precision.

With respect to related work on inference of interargument relations, we refer to [5]

for an overview. That paper also discusses the differences in expressivity, e.g. in relation to the use of systems of linear equations. With respect to [19] the main differences are the improved expressivity by integrating types and the use of a conceptually simpler abstract interpretation scheme.

References

- [1] R. Barbuti, R. Giacobazzi, and G. Levi. A general framework for semantics-based bottom-up abstract interpretation of logic programs. *ACM Transactions on Programming Languages and Systems*, 15(1):133–181, 1991.
- [2] A. Bossi, N. Cocco, and M. Fabris. Proving termination of logic programs by exploiting term properties. In *Proc. CCPSD-TAPSOFT'91*, pages 153–180. Springer-Verlag, LNCS 494, 1991.
- [3] A. Bossi, N. Cocco, and M. Fabris. Typed norms. In B. Krieg-Brueckner, editor, *Proc. ESOP'92*, pages 73–92. Springer-Verlag, LNCS 582, 1992.
- [4] F. Bronsard, T.K. Lakshman, and U.S. Reddy. A framework of directionality for proving termination of logic programs. In Krzysztof Apt, editor, *Proc. JICSLP '92*, pages 321–335. MIT Press, 1992.
- [5] D. De Schreye and S. Decorte. Termination of logic programs: the never-ending story. *Journal of Logic Programming*, (Special anniversary edition), 1994. Accepted for publication.
- [6] D. De Schreye, K. Verschaetse, and M. Bruynooghe. A framework for analysing the termination of definite logic programs with respect to call patterns. In *Proc. FGCS'92*, pages 481–488, ICOT Tokyo, 1992. ICOT.
- [7] S. Decorte, D. De Schreye, and M. Fabris. Automatic inference of norms : a missing link in automatic termination analysis. In D. Miller, editor, *Proceedings ILPS'93*, pages 420–436, Vancouver, Canada, 1993.
- [8] S. Decorte, D. De Schreye, and M. Fabris. Exploiting the power of typed norms in automatic inference of interargument relations. Technical report, Department of Computer Science, K.U.Leuven, Belgium, 1994.
- [9] G. Janssens and M. Bruynooghe. Deriving descriptions of possible values of program variables by means of abstract interpretation. *Journal of Logic Programming*, 13(2 & 3):205–258, 1992.
- [10] M. Karr. Affine relationships among variables of a program. *Acta Informatica*, 6:133–151, 1976.
- [11] B. Le Charlier and P. Van Hentenryck. Reexecution in abstract interpretation of prolog. In Krzysztof Apt, editor, *Proceedings of the Joint International Conference and Symposium on Logic Programming*, pages 750–764. MIT Press, 1992.
- [12] A. Mulkers. *Deriving Live Data Structures in Logic Programs by means of Abstract Interpretation*. Number 675 in LNCS. Springer-Verlag, 1993.

- [13] L. Plümer. *Termination proofs for logic programs*. Number 446 in LNAI. Springer-Verlag, 1990.
- [14] L. Plümer. Automatic termination proofs for Prolog programs operating on non-ground terms. In *Proc. ILPS'91*, pages 503–517, San Diego, October 1991. MIT Press.
- [15] K. Sohn and A. Van Gelder. Termination detection in logic programs using argument sizes. In *Proceedings 10th symposium on principles of database systems*, pages 216–226. ACM Press, May 1991.
- [16] J.D. Ullman and A. Van Gelder. Efficient tests for top-down termination of logical rules. *Journal ACM*, 35(2):345–373, April 1988.
- [17] K. Verschaeetse. *Static Termination Analysis for Definite Horn Clause Programs*. PhD thesis, Dept. Computer Science, K.U.Leuven, 1992.
- [18] K. Verschaeetse and D. De Schreye. Deriving termination proofs for logic programs, using abstract procedures. In *Proc. ICLP'91*, pages 301–315, Paris, June 1991. MIT Press.
- [19] K. Verschaeetse and D. De Schreye. Derivation of linear size relations by abstract interpretation. In M. Bruynooghe and M. Wirsing, editors, *Proceedings PLILP'92*, LNCS 631, pages 296–310. Springer-Verlag, 1992.
- [20] K. Verschaeetse, S. Decorte, and D. De Schreye. Automatic termination analysis. In *Proc. LOPSTR'92*, LNCS. Springer-Verlag, 1993.

A Proofs

Lemma A.1 T_P^α is monotonic: ■

Proof Let $E = \langle E_q^{k_1}, \dots, E_q^{k_n} \rangle, E' = \langle E_q^{l_1}, \dots, E_q^{l_n} \rangle \in D^\alpha$ and $E \sqsubseteq E'$. We need to prove $T_P^\alpha(E) \sqsubseteq T_P^\alpha(E')$. By definition of \sqsubseteq and T_P^α it is sufficient to prove that for all $i = 1, \dots, n, E \sqsubseteq E'$ implies $T_P^\alpha(p_i, E) \leq T_P^\alpha(p_i, E')$. Obviously, $\mathcal{A}(p_i(X_1, \dots, X_n), E) \leq \mathcal{A}(p_i(X_1, \dots, X_n), E')$, so that it remains to be shown that $e \rightarrow \text{lub}_{i=1, \dots, m}(\text{restrict}((X_1, \dots, X_k), \text{glb}_{B \in \text{Body}(C_i)}(\text{extend}((Z_1^i, \dots, Z_l^i), \mathcal{A}(B, e))))))$ is monotonic.

First note that $e \rightarrow \mathcal{A}(B, e)$ is monotonic. If B is a unification, then $\mathcal{A}(B, e)$ is independent of e . Else, let $B = p_j(\bar{U})$ and assume that $E \sqsubseteq E'$. We have: $\mathcal{A}(B, E) = E[p_j]$ renamed to $\bar{Y} = E_q^{k_j}$ renamed to $\bar{Y} \leq E_q^{l_j}$. Finally, all of the operations extend, restrict, lub and glb are monotonic, which concludes the proof. ■

Proposition A.2 T_P^α safely approximates the T_P -operator. ■

Proof We must prove: $\forall E = \langle E_1, \dots, E_n \rangle, T_P^\alpha(E) \supseteq \alpha(T_P(\gamma(E)))$. $\gamma(E)$ corresponds to a set of atoms $\{p^{T_1, \dots, T_k}(t_1, \dots, t_k)\}$. We know that $(\|t_1\|_{T_1}, \dots, \|t_k\|_{T_k})$ is a solution of $E[p^{T_1, \dots, T_k}]$ and that $t_i \in \mathcal{D}(T_i)$.

$T_P(\gamma(E))$ corresponds to a set of ground atoms $\{p^{T_1, \dots, T_n}(t_1, \dots, t_n)\}$ and for each of these atoms there exists a ground clause $p^{T_1, \dots, T_n}(t_1, \dots, t_n) \leftarrow B_1^{T_1, \dots, T_{m_1}}(t_1^1, \dots, t_{m_1}^1), \dots, B_l^{T_1, \dots, T_{m_l}}(t_1^l, \dots, t_{m_l}^l)$. Also, $\forall i, B_i^{T_1, \dots, T_{m_i}}(t_1^i, \dots, t_{m_i}^i) \in \gamma(E)$. This implies that $\forall j, t_j^i \in \mathcal{D}(T_j^i)$ and that $(\|t_1^i\|_{T_1^i}, \dots, \|t_{m_i}^i\|_{T_{m_i}^i})$ a solution is of $E[B_i^{T_1^i, \dots, T_{m_i}^i}]$. Let us now look at the abstraction of this set. It is a systems of equations which is a representative of the *least* equivalence class having the tuples $(\|t_1\|_{T_1}, \dots, \|t_n\|_{T_n})$ as solutions, i.e. $\alpha(T_P(\gamma(E))) = \langle E_1, \dots, E_k \rangle$ and $(\|t_1\|_{T_1}, \dots, \|t_n\|_{T_n})$ is a solution of $E[p^{T_1, \dots, T_n}]$.

Now, $T_P^\alpha(E) = \langle E_1', \dots, E_k' \rangle$. It is sufficient to prove that $\forall i, E_i' \geq E_i$ or $\dot{\gamma}(E_i') \supseteq \dot{\gamma}(E_i) (= T_P(\gamma(E)))$.

Suppose that $T_P(\gamma(E))$ contains the atom $p^{T_1, \dots, T_n}(t_1, \dots, t_n)$ and that $(\|t_1\|_{T_1}, \dots, \|t_n\|_{T_n})$ is not a solution of $E'[p^{T_1, \dots, T_n}]$. Then either, it is not a solution of $E[p^{T_1, \dots, T_n}]$ or not of $\text{lub}_{i=1, \dots, m}(\text{restrict}((X_1, \dots, X_k), \text{glb}_{B \in \text{Body}(C_i)}(\text{extend}((Z_1^i, \dots, Z_l^i), \mathcal{A}(B, E))))))$.

If the atom is in $\gamma(E)$, our assumption is false because the the tuple is a solution of $E[p^{T_1, \dots, T_n}]$. In the other case, it must be such that for all clauses C_i , the tuple is not a solution of $\text{restrict}((X_1, \dots, X_k), \text{glb}_{B \in \text{Body}(C_i)}(\text{extend}((Z_1^i, \dots, Z_l^i), \mathcal{A}(B, E))))$. Because the restriction operator does not remove solutions, $\forall i$, the tuple may not be a solution of $\text{glb}_{B \in \text{Body}(C_i)}(\text{extend}((Z_1^i, \dots, Z_l^i), \mathcal{A}(B, E)))$. We now fix C_i to the clause specified above. Then, considering the glb operation, this condition is equivalent to $\exists j$ such that the tuple $(\|t_1^j\|_{T_1^j}, \dots, \|t_{m_j}^j\|_{T_{m_j}^j})$ is not a solution of $\mathcal{A}(B^{T_1^j, \dots, T_{m_j}^j}, E)$. This is a contradiction because for that clause we know that $(\|t_1^j\|_{T_1^j}, \dots, \|t_{m_j}^j\|_{T_{m_j}^j})$ is a solution of $E[B_j^{T_1^j, \dots, T_{m_j}^j}]$, $\forall j$. ■