Typed Norms for Typed Logic Programs

Jonathan Martin[*] Andy King[†] and Paul Soper[*]

1 Introduction

Such is the complex nature of termination that ad hoc methods for its automatic detection in logic programs are giving way to techniques more firmly based on theory. Many of these approaches relate to the early theoretical result [3] which showed that a logic program terminates for *bounded* goals if and only if it is *recurrent*. Definitions of recurrency and boundedness are formulated in terms of level mappings which assign natural numbers, or levels, to ground atoms.

A predicate is recurrent with respect to some level mapping if the level of its head is greater than the level of each of its body atoms. The termination of bounded goals, whose level cannot increase, then follows from the well-foundedness of the natural numbers.

Level mappings are often defined in terms of *norms* which measure the size of terms. For example, the norm $|.|_{list\text{-}length}$ defined to measure the length of a list, can be used as the basis for a level mapping for the `Delete/3` predicate below. Comparing the size of the second argument in the head of the recursive clause with the size of the second argument in the recursive call, and using list length as a measure for size, we see that the size of this argument decreases by one on each recursive call. Thus the predicate is recurrent with respect to the level mapping $|.|$ defined by $|\texttt{Delete}(t_1, t_2, t_3)| = |t_2|_{list\text{-}length}$ and terminates for all goals bounded with respect to $|.|$. Note that the predicate is also recurrent with respect to other level mappings and indeed termination can be proved for other goals by choosing a different mapping.

```
Delete(x, [x|y], y).
Delete(x, [y|z], [y|w]) <-
   Delete(x, z, w).
```

Deducing termination for programs which are not structurally recursive is more complex, requiring the derivation of inter-argument relationships [2]. Inter-argument relationships express how the sizes of an atom's arguments are related. In the case of `Delete/3`, for example, the length of the second argument is one plus the length of the third argument. The `Perm/2` predicate defined below is one example where an inter-argument relationship is needed to prove termination.

```
Perm([], []).
Perm([h|t], [a|p]) <-
   Delete(a, [h|t], l) &
   Perm(l, p).
```

In fact it can be shown that this program is not recurrent and will not terminate for all ground queries – recurrency implies that a program terminates for *all* computation rules and here there exists a computation rule which selects non-ground `Delete/3` goals which lead to infinite derivations. It can be shown however to be *acceptable* [1], an analogous concept to recurrency for programs executed using a left-to-right computation rule. A key step in the proof is to show that the size of the first argument in the head of the recursive clause is strictly greater than the size

[*]Department of Electronics and Computer Science, University of Southampton, Southampton, SO9 5NH, UK. {jcm93r, pjs}@ecs.soton.ac.uk

[†]Computing Laboratory, University of Kent at Canterbury, Canterbury, CT2 7NF, UK. a.m.king@ukc.ac.uk

of the first argument in the recursive call, that is $|l|_{list\text{-}length} < |[h|t]|_{list\text{-}length}$. This can only be inferred by deducing the inter-argument relationship for `Delete/3` given above.

Choosing the right norm is crucial in deducing termination and deriving inter-argument relationships. Furthermore, different norms are often needed for each case. As an example, consider the predicate `FlattenAndLength/3` defined below which flattens a list of lists and computes the length of the original list. The norm which sums the lengths of the sublists of the first argument can be used to deduce termination and is also needed to infer a useful inter-argument relationship between the first and second arguments. To derive a precise relationship between the first and third arguments, however, the norm $|.|_{list\text{-}length}$ is also needed.

```
FlattenAndLength([], [], 0).
FlattenAndLength([e|x], r, Succ(z)) <-
   Append(e, y, r) &
   FlattenAndLength(x, y, z).
```

Early work on termination relied on the user to provide the necessary norms. As this had limited usefulness a method to automatically generate norms from a program was proposed in [6]. The approach focuses on deriving norms from type graphs that have previously been inferred by an analysis of the program. The technique is effective in generating norms for proving termination of many of the programs found in the termination literature. The approach is clearly inappropriate, however, in the context of a typed language such as Gödel [11] when the types are already known.

As typed logic programming becomes more mainstream, system building tools like partial deduction systems will need to be mapped from untyped languages to typed ones. SAGE [9] is one example of a partial deduction system developed for the typed language Gödel. Although SAGE does well to demonstrate the effectiveness of self-application and how the overheads of the ground representation in meta-programs can be removed, there is much potential for improvement [10]. Its main weakness lies in a rather rudimentary termination analysis which would benefit considerably from the well developed techniques found in the termination literature. Inevitably, norms will play a crucial role in such an analysis. It is important, however, when mapping techniques across from the untyped setting that the new techniques should exploit the new type system as much as possible. In the case of automatic norm derivation the approach in [6] clearly would not take advantage of the prescribed types. As a result of this and since "any state-of-the-art approach to termination analysis needs to take type information into account" [7], new techniques are needed to derive norms directly from these types and avoid the overhead of type graph generation. We present one such technique.

In this paper we show how norms can be generated from the prescribed types of a program written in a language which supports parametric polymorphism, e.g. Gödel [11]. Interestingly, the types highlight restrictions of earlier norms and suggest how these norms can be extended to obtain some very general and powerful notions of norm which can be used to measure any term in an almost arbitrary way. We see our work on norm derivation as a contribution to the termination analysis of typed logic programs which, in particular, forms an essential part of partial deduction systems such as SAGE.

The paper is structured as follows. The next section introduces polymorphic, many-sorted languages and programs. Section 3 defines linear, semi-linear and hierarchical typed norms and discusses the problem of rigidity in a polymorphic many-sorted context. Section 4 describes how to infer the norms of section 3 from the prescribed types of a program. Related work is addressed in the penultimate section and we conclude with some directions for future work.

2 Theoretical foundations

2.1 Polymorphic many-sorted languages

Let Σ_τ (resp. Σ_f) be an alphabet of type constructor (resp. typed function) symbols which includes at least one base (resp. constant) and let Σ_p be an alphabet of predicate symbols. Let U denote a countably infinite set of type parameters so that the term structure $T(\Sigma_\tau, U)$ represents the set of parametric types. Let $V = \{V_\tau \mid \tau \in T(\Sigma_\tau, U)\}$ denote a family of countably infinite, disjoint sets of variables for polymorphic (and monomorphic) formulae, where each $v_\tau \in V_\tau$ has type τ. Variables will be denoted by the letters v, w, x, y, and z, whereas parameters will be denoted by the letter u. Each $f_\sigma \in \Sigma_f$ (resp. $p_\sigma \in \Sigma_p$) is assigned a unique[1] type (modulo renaming) $\sigma = \langle \tau_1 \ldots \tau_n, \tau \rangle$ (resp. $\sigma = \tau_1 \ldots \tau_n$) where $\tau_1 \ldots \tau_n \in T(\Sigma_\tau, U)^\star$ and $\tau \in T(\Sigma_\tau, U) \setminus U$. We call τ the range type of $f_{\langle \tau_1 \ldots \tau_n, \tau \rangle} \in \Sigma_f$ when $n > 0$. Types are unique in the sense that if $f_{\langle \sigma_1 \ldots \sigma_n, \sigma \rangle}$, $f_{\langle \tau_1 \ldots \tau_n, \tau \rangle} \in \Sigma_f$ (resp. $p_{\sigma_1 \ldots \sigma_n}$, $p_{\tau_1 \ldots \tau_n} \in \Sigma_p$) then $\sigma_i = \tau_i$ and $\sigma = \tau$. A symbol will often be written without its type if it is clear from the context. The triple $L = \langle \Sigma_p, \Sigma_f, V \rangle$ defines a polymorphic many-sorted first-order language.

Terms, atoms and formulae are defined in the usual way [11]. We denote by $var(o)$ (resp. $par(o)$) the set of variables (resp. parameters) in a syntactic object o. The set of term (resp. type) substitutions is denoted by Sub (resp. Sub_τ). The set of all instances of Σ_f is denoted by $\Sigma_f^\dagger = \{f_{\psi(\sigma)} | f_\sigma \in \Sigma_f \wedge \psi \in Sub_\tau\}$.

2.2 Polymorphic many-sorted programs

Let $P = \langle \Delta, S \rangle$ be a polymorphic many-sorted logic program where Δ is a triple $\langle \Delta_\tau, \Delta_f, \Delta_p \rangle$ of type declarations and S is a set of statements of the form $\forall(a \leftarrow w)$ where a is an atom and w is either absent or a polymorphic many-sorted formula. The type declarations Δ_τ, Δ_f and Δ_p define respectively Σ_τ, Σ_f and Σ_p.

Each function declaration $f : \tau_1 \times \ldots \times \tau_n \rightarrow \tau \in \Delta_f$ (resp. constant declaration $c : \tau \in \Delta_f$) where $\tau_1, \ldots, \tau_n \in T(\Sigma_\tau, U)$ and $\tau \in T(\Sigma_\tau, U) \setminus U$ implies $f_{\langle \tau_1 \ldots \tau_n, \tau \rangle} \in \Sigma_f$ (resp. $c_{\langle \epsilon, \tau \rangle} \in \Sigma_f$). Similarly, each predicate declaration $p : \tau_1 \times \ldots \times \tau_n \in \Delta_p$ (resp. proposition declaration $p \in \Delta_p$) where $\tau_1, \ldots, \tau_n \in T(\Sigma_\tau, U)$ implies $p_{\tau_1 \ldots \tau_n} \in \Sigma_p$ (resp. $p_\epsilon \in \Sigma_p$). Δ_f (resp. Δ_p) is assumed to be universal, that is, each symbol has exactly one declaration in Δ_f (resp. Δ_p) so that Σ_f (resp. Σ_p) is well-defined.

Given a language $L = \langle \Sigma_p, \Sigma_f, V \rangle$ defined by a program P, we define a family of extended Herbrand domains as follows. Each $ED_{Herb,\tau}$ is the least set such that if $v_\tau \in V_\tau$ then $v_\tau \in ED_{Herb,\tau}$; if $f_{\langle \epsilon, \sigma \rangle} \in \Sigma_f$ and $\tau = \psi(\sigma)$ then $f_{\langle \epsilon, \psi(\sigma) \rangle} \in ED_{Herb,\tau}$; and if $f_{\langle \sigma_1 \ \ldots \ \sigma_n, \sigma \rangle} \in \Sigma_f$ and $t_i \in ED_{Herb,\tau_i}$ with $par(\sigma_i) \cap par(\tau_j) = \emptyset$ for all i, j and $par(\tau_j) \cap par(\tau_k) = \emptyset$ for all $j \neq k$ and $\psi \in mgu(\{\sigma_1 = \tau_1, \ldots, \sigma_n = \tau_n\} \cup \{\rho_i = \rho_j \mid v_{\rho_i} \in t_i \wedge v_{\rho_j} \in t_j\})$ then $\psi(f(t_1, \ldots, t_n)) \in ED_{Herb,\psi(\sigma)}$.

3 Norms for typed logic programs

A norm is a mapping that measures the size of a term. The norm list length, for example, might typically count the number of $Cons$ symbols that occur in a list.

Example 3.1 The length of a list of integers can be expressed as

$$\begin{aligned} |Nil| &= 0 \\ |Cons(t_1, t_2)| &= 1 + |t_2| \end{aligned}$$

□

The mapping is partial since it is only defined for closed, that is Nil-terminated, lists. To define norms as total mappings we introduce the alphabets $\Sigma_\tau = \{Lin\}$ and $\Sigma_f = \{+_{\langle Lin.Lin, Lin \rangle}$, $0_{\langle \epsilon, Lin \rangle}$, $1_{\langle \epsilon, Lin \rangle}\}$ so that $ED_{Herb,Lin}$ represents the class of linear expressions on V_{Lin} where terms such as $x_{Lin} + y_{Lin} + y_{Lin} + 1_{\langle \epsilon, Lin \rangle} + 1_{\langle \epsilon, Lin \rangle} + 1_{\langle \epsilon, Lin \rangle}$ are abbreviated to $x + 2y + 3$.

It is usually too restrictive to use a single norm to measure the size of any term in a program. Different terms need to be measured according to their structure or, equivalently, according to their type. This motivates the introduction of a typed norm $|.|_\tau$ which only measures terms of type τ.

[1] For overloaded symbols, for example $+$, we assume the symbol is uniquely renamed for each of its types.

Definition 3.1 (typed norm I) A *typed norm* for a polymorphic type τ is a mapping $|.|_\tau : ED_{Herb,\tau} \rightarrow ED_{Herb,Lin}$. □

Example 3.2 The typed norm $|.|_{List(Int)} : ED_{Herb,List(Int)} \rightarrow ED_{Herb,Lin}$ defined below measures the length of both open and closed lists of integers.

$$\begin{aligned} |v|_{List(Int)} &= v \\ |Nil|_{List(Int)} &= 0 \\ |Cons(t_1, t_2)|_{List(Int)} &= 1 + |t_2|_{List(Int)} \end{aligned}$$

□

It is appropriate at this point to review the important concept of rigidity. This idea was originally introduced in [5] in order to prove termination for a class of goals with possibly non-ground terms. A rigid term is one whose size, as determined by a norm, is not affected by substitutions applied to the term.

Definition 3.2 (rigid term) Let $|.|_\tau$ be a typed norm for τ and t be a term of type τ. Then t is *rigid* with respect to $|.|_\tau$ iff $\forall \theta \in Sub$, $|t|_\tau = |t\theta|_\tau$.

Example 3.3 The term $Cons(x, Cons(y, Nil))$ is rigid wrt the norm $|.|_{List(Int)}$ of example 3.2 since for every substitution $\{x \mapsto t_1, y \mapsto t_2\}$ where t_1 and t_2 are terms $|Cons(t_1, Cons(t_2, Nil))|$ $= 2$. □

By defining level mappings in terms of norms, it is possible to define a class of bounded goals [3] in terms of rigidity. More precisely, an atom is bounded with respect to a level mapping if each argument of the atom whose size is measured in the level mapping is rigid. A problem arises, however, with the typed norms used in level mappings. In measuring the level of an atom, a norm $|.|_\tau$, which can only measure terms of type τ may be applied to a term of type σ, where $\sigma = \psi(\tau)$ for some $\psi \in Sub_\tau$.

Example 3.4 Given that $\Sigma_\tau = \{Int, List\}$, $\Sigma_f = \{Nil_{\langle \epsilon, List(u) \rangle}, Cons_{\langle u.List(u), List(u) \rangle}\}$, $\Sigma_p = \{Traverse_{List(u)}\}$ and $S = \{Traverse(Nil)., Traverse(Cons(x, y)) \leftarrow Traverse(y).\}$ then the norm $|.|_{List(u)}$ defined by

$$\begin{aligned} |v|_{List(u)} &= v \\ |Nil|_{List(u)} &= 0 \\ |Cons(t_1, t_2)|_{List(u)} &= 1 + |t_2|_{List(u)} \end{aligned}$$

can be used to define a level mapping $|.|$ for the $Traverse/1$ predicate as follows

$$|Traverse(t)| = |t|_{List(u)}$$

The problem is that in trying to prove recurrency with respect to the level mapping $|.|$ for $Traverse/1$, the level mapping can be applied to atoms such as $Traverse(Cons(1, Nil))$, yet the type of the argument of $Traverse$ in this instance, $List(Int)$, is not the type $List(u)$ for which the mapping is defined. □

This problem arises due to the polymorphism in our typed language and is not difficult to remedy. The domain of the norm must be changed and a constraint imposed to ensure that the rigidity property still holds.

Definition 3.3 (typed norm II) A *typed norm* for a polymorphic type τ is a mapping $|.|_\tau : \cup_{\psi \in Sub_\tau} ED_{Herb,\psi(\tau)} \rightarrow ED_{Herb,Lin}$ where

$$\forall \psi \in Sub_\tau, \quad |f_{\langle \tau_1 \ldots \tau_n, \tau \rangle}(t_1, \ldots, t_n)|_\tau = |f_{\langle \psi(\tau_1) \ldots \psi(\tau_n), \psi(\tau) \rangle}(t_1, \ldots, t_n)|_\tau$$

□

To see why the constraint is required, suppose that the term t is rigid wrt the type II norm $|.|_\tau$, then, by the definition of rigidity

$$\forall \theta \in Sub, \quad |t|_\tau = |t\theta|_\tau \tag{1}$$

Now applying a variable substitution to a term often has the effect of further instantiating the type of the term. For example the type of the term $Cons(x, Nil)$ is $List(u)$, but the type of $Cons(x, Nil)\{x \mapsto 1\} = Cons(1, Nil)$ is $List(Int)$. Hence we constrain the equations defining $|.|_\tau$ so that equation (1) holds.

The following proposition provides us with a (weak) syntactical characterisation of rigid terms. This can be strengthened to the if and only if version by imposing some rather natural conditions on the way norms are defined. Unfortunately space restrictions do not allow us to give the details here. We only remark that these conditions do not restrict the norms in any way.

Proposition 3.1 (rigid term – weak) Let $|.|_\tau$ be a typed norm for τ and t be a term of type τ. Then t is *rigid* with respect to $|.|_\tau$ if $var(|t|_\tau) = \emptyset$. □

Throughout the remainder of this paper we will only be concerned with type II norms. Henceforth $|.|_\tau$ will only denote a type II norm whose domain is unambiguously defined by definition 3.3. In view of the constraint on type II norms, we will write $|f(t_1, \ldots, t_n)|_\tau$ where f represents $f_{\langle \psi(\tau_1) \ldots \psi(\tau_n), \psi(\tau) \rangle}$ for all $\psi \in Sub_\tau$. Although each norm is annotated with its type, the following example illustrates that several norms may exist for the same type.

Example 3.5 The typed norm $|.|^{len}_{List(List(Int))}$ measures the length of a list whose elements are lists of integers. The typed norm $|.|^{sum}_{List(List(Int))}$ sums the lengths of the elements of such a list.

$$\begin{aligned}
|v|^{len}_{List(List(Int))} &= v \\
|Nil|^{len}_{List(List(Int))} &= 0 \\
|Cons(t_1, t_2)|^{len}_{List(List(Int))} &= 1 + |t_2|^{len}_{List(List(Int))}
\end{aligned}$$

$$\begin{aligned}
|v|^{sum}_{List(List(Int))} &= v \\
|Nil|^{sum}_{List(List(Int))} &= 0 \\
|Cons(t_1, t_2)|^{sum}_{List(List(Int))} &= |t_1|^{sum}_{List(Int)} + |t_2|^{sum}_{List(List(Int))}
\end{aligned}$$

where $|.|^{sum}_{List(Int)}$ is equal to the norm $|.|_{List(Int)}$ of example 3.2. Note that the norm $|.|^{len}_{List(List(Int))}$ is characterised by a weight of 1 in its recursive equation and the selection of the second argument position only, whereas the norm $|.|^{sum}_{List(List(Int))}$ is characterised by a weight of 0 in its recursive equation and the selection of both argument positions. □

To uniquely characterise a norm we introduce a pair $s = \langle w_s, I_s \rangle$ of partial mappings where $w_s : \Sigma^\dagger_f \to \mathbb{N}$ assigns a weight to each function symbol and $I_s : \Sigma^\dagger_f \to \wp(\mathbb{N})$ selects a subset of the argument positions for each function symbol. The definition of a norm for a type τ depends on s and therefore we denote the norm by $|.|^s_\tau$.

Example 3.6 In example 3.5 $len = \langle w_{len}, I_{len} \rangle$ where

$$\begin{aligned}
w_{len} &= \{Nil \mapsto 0, \quad Cons \mapsto 1\} \\
I_{len} &= \{Nil \mapsto \{\}, Cons \mapsto \{2\}\}
\end{aligned}$$

□

We are now in a position to define a notion of linear and semi-linear norms [4, 13] for typed programs.

Definition 3.4 (linear typed norm) A typed norm $|.|^s_\tau$ is *linear* iff

$$\begin{array}{ll}
\forall v \in V_{\psi(\tau)}, \forall \psi \in Sub_\tau & |v|^s_\tau = v \\
\forall f_{\langle \tau_1 \ldots \tau_n, \tau \rangle} \in \Sigma^\dagger_f & |f(t_1, \ldots, t_n)|^s_\tau = w_s(f_{\langle \tau_1 \ldots \tau_n, \tau \rangle}) + \sum_{i \in I_s(f_{\langle \tau_1 \ldots \tau_n, \tau \rangle})} |t_i|^s_\tau
\end{array}$$

where $I_s(f_{\langle \tau_1 \ldots \tau_n, \tau \rangle}) = \{1, \ldots, n\}$. □

Note that the types highlight an inherent restriction of linear norms, that is, these norms are only defined when $\tau_i = \tau$ for $i = 1, \ldots, n$. Such norms have limited applicability.

Example 3.7 Given $\Sigma_\tau = \{Tree\}$ and $\Sigma_f = \{Leaf_{\langle\epsilon,Tree\rangle}, Node_{\langle Tree.Tree,Tree\rangle}\}$, the linear typed norm for $Tree$ that counts the number of function symbols in a term is defined by

$$\begin{aligned} |v|^{size}_{Tree} &= v \\ |Leaf|^{size}_{Tree} &= 1 \\ |Node(t_1,t_2)|^{size}_{Tree} &= 1 + |t_1|^{size}_{Tree} + |t_2|^{size}_{Tree} \end{aligned}$$ □

The following definition generalises linear typed norms by allowing $I_s(f_{\langle\tau_1\ldots\tau_n,\tau\rangle}) \subseteq \{1,\ldots,n\}$. In the special case when $I_s(f_{\langle\tau_1\ldots\tau_n,\tau\rangle}) = \{1,\ldots,n\}$ the two definitions are equivalent.

Definition 3.5 (semi-linear typed norm) A typed norm $|.|^s_\tau$ is *semi-linear* iff

$$\begin{array}{ll} \forall v \in V_{\psi(\tau)}, \forall\psi \in Sub_\tau & |v|^s_\tau = v \\ \forall f_{\langle\tau_1\ldots\tau_n,\tau\rangle} \in \Sigma^\dagger_f & |f(t_1,\ldots,t_n)|^s_\tau = w_s(f_{\langle\tau_1\ldots\tau_n,\tau\rangle}) + \sum_{i\in I_s(f_{\langle\tau_1\ldots\tau_n,\tau\rangle})} |t_i|^s_\tau \end{array}$$

where $I_s(f_{\langle\tau_1\ldots\tau_n,\tau\rangle}) \subseteq \{1,\ldots,n\}$. □

Example 3.8 Given $\Sigma_\tau = \{Int, List\}$ and $\Sigma_f = \{Nil_{\langle\epsilon,List(u)\rangle}, Cons_{\langle u.List(u),List(u)\rangle}\}$, then the norm $|.|^{len}_{List(List(Int))}$ defined in example 3.5 is semi-linear. □

Semi-linear norms are not expressive enough to measure the sizes of terms that can be defined in a typed language such as Gödel. To quote [4, pp. 72, paragraph 2] "The recursive structure of a semi-linear norm gets into the term structure by only one level. Moreover so far it is not defined how different semi-linear norms can be linked to work together. The definition of a semi-linear norm is recursively based only onto itself and it is easy to understand that this is a severe restriction." Again the types highlight where the essential problem lies: the norm applied to t_i is $|.|_\tau$ whereas the type of t_i is τ_i. The following definition overcomes this limitation of semi-linear norms.

Definition 3.6 (hierarchical typed norm) A typed norm $|.|^s_\tau$ is *hierarchical* iff

$$\begin{array}{ll} \forall v \in V_{\psi(\tau)}, \forall\psi \in Sub_\tau & |v|^s_\tau = v \\ \forall f_{\langle\tau_1\ldots\tau_n,\tau\rangle} \in \Sigma^\dagger_f & |f(t_1,\ldots,t_n)|^s_\tau = w_s(f_{\langle\tau_1\ldots\tau_n,\tau\rangle}) + \sum_{i\in I_s(f_{\langle\tau_1\ldots\tau_n,\tau\rangle})} |t_i|^s_{\tau_i} \end{array}$$

where $I_s(f_{\langle\tau_1\ldots\tau_n,\tau\rangle}) \subseteq \{1,\ldots,n\}$ and $|t_i|^s_{\tau_i}$ are hierarchical typed norms. □

Example 3.9 Given the alphabets of example 3.8, the norm $|.|^{sum}_{List(List(Int))}$ defined in example 3.5 is hierarchical and, in fact, cannot be expressed as a semi-linear norm. □

Note that definition 3.6 is closely related to definition 4.5 of [6]. Both generalise the definition of a type norm proposed in [13]. In [6] the relationship between typed norms and semi-linear norms is not made explicit, but our presentation makes the relationships between the various norms clear. In particular, we see that every linear typed norm is semi-linear and every semi-linear typed norm is hierarchical.

Although hierarchical norms allow us to inspect the structure of terms at a deeper level than in the semi-linear case, the pair of mappings s maps a functor of a given type to the same pair of values regardless of its depth in the term. In certain (pathological) circumstances this can impede the detection of a well-founded ordering.

Example 3.10 With Σ_τ and Σ_f as defined in example 3.9, consider the hierarchical typed norm $|.|^s_{Tree}$ defined by

$$\begin{aligned} |v|^s_{Tree} &= v \\ |Leaf|^s_{Tree} &= w_s(Leaf) \\ |Node(t_1,t_2)|^s_{Tree} &= w_s(Node) + \textstyle\sum_{i\in I_s(Node)} |t_i|^s_{Tree} \end{aligned}$$

There is no definition of s which will satisfy the inequality

$$|Node(Node(w, Node(x,y)), z)|^s_{Tree} > |Node(Node(Node(w,x), y), z)|^s_{Tree} \tag{2}$$

needed to prove recurrency for the predicate $Shift/1$ defined by

```
Shift(Node(Node(_, Leaf), Leaf)).
Shift(Node(Node(w, Node(x, y)), z)) <-
  Shift(Node(Node(Node(w, x), y), z)).
```

The following table illustrates that for all values of $I_s(Node)$ and $w_s(Node)$ and for every variable assignment for w, x, y, z the left-hand side is always less than or equal to the right-hand side.

$I_s(Node)$	$\|Node(Node(w, Node(x,y)), z)\|^s_{Tree}$	$\|Node(Node(Node(w,x), y), z)\|^s_{Tree}$
$\{1,2\}$	$3w_s(Node) + w + x + y + z$	$3w_s(Node) + w + x + y + z$
$\{1\}$	$2w_s(Node) + w$	$3w_s(Node) + w$
$\{2\}$	$w_s(Node) + z$	$w_s(Node) + z$
$\{\}$	$w_s(Node)$	$w_s(Node)$

The inequality can be satisfied, however, by substituting in (2) the norm $|.|^{left}_{Tree}$ defined by

$$\begin{array}{rlrl} |v|^{left}_{Tree} &= v & |v|^{right}_{Tree} &= v \\ |Node(t_1,t_2)|^{left}_{Tree} &= 1 + |t_1|^{right}_{Tree} & |Node(t_1,t_2)|^{right}_{Tree} &= 1 + |t_2|^{right}_{Tree} \end{array}$$

□

The definition of a hierarchical typed norm can be generalised further to accommodate such examples by replacing $|t_i|^s_{\tau_i}$ in the definition with $|t_i|^{s_i}_{\tau_i}$ where each s_i is a new pair of mappings. This additional expressiveness allows a term to be measured in a very flexible way, though in practice it is unlikely that such generality will be needed and besides which the complexity introduced is mind-boggling.

4 Automatic generation of norms

We show how the typed norms of the previous section can be derived directly from the prescribed types of a program. For a program P, we require a finite set of norms which will enable us to measure the size of any term occurring in P. The norms needed will be determined by the types that can occur in P. In the following we consider two types to be equivalent if one is a renaming of the other.

Definition 4.1 (argument types) Suppose that P defines the language $\langle \Sigma_p, \Sigma_f, V\rangle$. The set of argument types for P is denoted by $T_{arg} = \{\tau_i \mid p_{\tau_1 \ldots \tau_n} \in \Sigma_p \wedge 1 \le i \le n\}$. □

The set T_{arg} represents the types of all terms occurring as arguments of atoms in P, in that if the type of an argument of some atom is τ, then either $\tau \in T_{arg}$ or $\exists \psi \in Sub_\tau$, $\exists \sigma \in T_{arg}$ such that $\tau = \psi(\sigma)$. The following definition captures the types of subterms of arguments.

Definition 4.2 (argument subtypes) For each $\tau \in T_{arg}$ we define T^τ_{sub} the set of subtypes of τ to be the least set such that $\tau \in T^\tau_{sub}$ and if $\sigma \in T^\tau_{sub}$, $f_{\langle \rho_1 \ldots \rho_n, \rho\rangle} \in \Sigma_f$ and $\sigma = \psi(\rho)$, then $\psi(\rho_i) \in T^\tau_{sub}$, for all $i = 1, \ldots, n$. □

Example 4.1 If $\Sigma_f = \{Nil_{\langle \epsilon, List(u)\rangle}, Cons_{\langle u.List(u), List(u)\rangle}\}$ and $\Sigma_p = \{P_{List(List(u))}, Q_{List(u)}\}$, then $T_{arg} = \{List(List(u)), List(u)\}$, $T^{List(List(u))}_{sub} = \{List(List(u)), List(u), u\}$ and $T^{List(u)}_{sub} = \{List(u), u\}$. □

By defining a norm $|.|_\tau$ for each $\tau \in T_{arg}$, we are able to measure the size of any argument occurring in the program. The sets $T_{sub}^{\tau_i}$ are used to facilitate the definitions of these norms. It will often be the case that some of the arguments in a program have the same type and different norms may be required to measure the sizes of such arguments. We thus define for each $\tau \in T_{arg}$ a norm parameterised by a pair s as in the preceding section. Later, s can be defined for individual arguments.

Before defining the induction process we first make an important observation which has an effect on the definition of the norms. We first note that the type of a constant or the range type of a function must be either a base type or a type with a top-level constructor. A consequence of this is that any term whose type is a parameter is a variable. The term structure of any term assigned to this variable cannot be accessed or altered in any way within the local computation, since if it could, the type of the term would be known and thus the variable would be of a more specific type. Thus the term (and its size measured wrt to any norm) never changes and hence has no effect on termination at the local level. This means that when defining the norm $|.|_u$ where $u \in U$, the value of $|t|_u$ for any term t should be constant. To simplify the definition we assume the constant value is zero. Furthermore, the norm $|.|_u$ can be removed from any definition which depends on it.

Definition 4.3 (induced typed norm) For each $\tau \in T_{arg}$ we define the hierarchical typed norm $|.|_\tau^s : \cup_{\psi \in Sub_\tau} ED_{Herb,\psi(\tau)} \to ED_{Herb,Lin}$ as the least set of equations E_τ^s as follows. If $\tau \in U$ then $E_\tau^s = \{|.|_\tau^s = 0\}$, else

$$E_\tau^s = \left\{ |v|_\nu^s = v \mid \nu \in T_{sub}^\tau \right\} \cup \left\{ |f(t_1,\ldots,t_n)|_\nu^s = w_s(f_{\psi(\sigma)}) + \textstyle\sum_{i \in I_s(f_{\psi(\sigma)})} |t_i|_{\psi(\rho_i)}^s \;\middle|\; \begin{array}{l} f_\sigma \in \Sigma_f \wedge \sigma = \langle \rho_1 \ldots \rho_n, \rho \rangle \wedge \\ \nu \in T_{sub}^\tau \wedge \nu = \psi(\rho) \wedge \psi \in Sub_\tau \end{array} \right\}$$

A pair $s = \langle w_s, I_s \rangle$ is partially defined for each $\tau \in T_{arg}$ as follows. For each $\nu \in T_{sub}^\tau$ and $f_\sigma \in \Sigma_f$, $\sigma = \langle \rho_1 \ldots \rho_n, \rho \rangle$ such that $\nu = \psi(\rho)$ for some $\psi \in Sub_\tau$, we add the mapping $f_{\psi(\sigma)} \mapsto w \in \mathbb{N}$ to w_s and the mapping $f_{\psi(\sigma)} \mapsto I \subseteq \{1,\ldots,n\}$ to I_s with the constraint that $i \notin I$ for all $\rho_i \in U$. □

Note that due to the definition of T_{sub}^τ each $|.|_{\psi(\sigma_i)}^s$ is defined in E_τ^s. Thus each E_τ^s is well defined pending a complete definition of the pair $s = \langle w_s, I_s \rangle$.

Example 4.2 Given T_{arg} as defined in example 4.1, we partially define a pair $s = \langle w_s, I_s \rangle$ for the type $List(List(u))$ and a pair $t = \langle w_t, I_t \rangle$ for the type $List(u)$ as follows:

$$\begin{array}{rl}
w_s = & \left\{ \begin{array}{ll} Nil_{\langle \epsilon, List(List(u)) \rangle} \mapsto w_1, & Cons_{\langle List(u).List(List(u)), List(List(u)) \rangle} \mapsto w_2, \\ Nil_{\langle \epsilon, List(u) \rangle} \mapsto w_3, & Cons_{\langle u.List(u), List(u) \rangle} \mapsto w_4 \end{array} \right\} \\
I_s = & \left\{ \begin{array}{ll} Nil_{\langle \epsilon, List(List(u)) \rangle} \mapsto \{\}, & Cons_{\langle List(u).List(List(u)), List(List(u)) \rangle} \mapsto I_1 \subseteq \{1,2\} \\ Nil_{\langle \epsilon, List(u) \rangle} \mapsto \{\}, & Cons_{\langle u.List(u), List(u) \rangle} \mapsto I_2 \subseteq \{2\} \end{array} \right\} \\
w_t = & \left\{ Nil_{\langle \epsilon, List(u) \rangle} \mapsto w_5, Cons_{\langle u.List(u), List(u) \rangle} \mapsto w_6 \right\} \\
I_t = & \left\{ Nil_{\langle \epsilon, List(u) \rangle} \mapsto \{\}, Cons_{\langle u.List(u), List(u) \rangle} \mapsto I_3 \subseteq \{2\} \right\}
\end{array}$$

where $w_1, w_2, w_3, w_4, w_5, w_6 \in \mathbb{N}$.

Choosing for example $w_1 = w_2 = w_3 = w_5 = 0, w_4 = w_6 = 1$, $I_1 = \{1,2\}$ and $I_2 = I_3 = \{2\}$ we derive the following equation sets

$$E_{List(List(u))}^s = \left\{ \begin{array}{ll}
|v|_{List(List(u))}^s & = v, \\
|Nil|_{List(List(u))}^s & = 0, \\
|Cons(t_1,t_2)|_{List(List(u))}^s & = |t_1|_{List(u)}^s + |t_2|_{List(List(u))}^s, \\
|v|_{List(u)}^s & = v, \\
|Nil|_{List(u)}^s & = 0, \\
|Cons(t_1,t_2)|_{List(u)}^s & = 1 + |t_2|_{List(u)}^s
\end{array} \right\}$$

$$E_{List(u)}^t = \left\{ \begin{array}{ll}
|v|_{List(u)}^t & = v, \\
|Nil|_{List(u)}^t & = 0, \\
|Cons(t_1,t_2)|_{List(u)}^t & = 1 + |t_2|_{List(u)}^t
\end{array} \right\}$$

□

Note that the sets of terms for which the norms are defined are not disjoint. For example, the domain of the norm $|.|^s_{List(List(u))}$ of example 4.2 is a subset of the domain for the norm $|.|^t_{List(u)}$. There is no confusion, however, when deciding which norm to use on a particular argument of an atom since the choice is determined by the atom's predicate symbol.

Example 4.3 Consider the atom $Q_{List(u)}(Cons(Cons(1, Nil), Nil))$ which may appear as part of a goal for the predicate $Q_{List(u)}$. Although the type of the atom's argument is $List(List(Int))$, the correct norm to use would be $|.|^t_{List(u)}$ for some t since the type of the predicate is $List(u)$. □

All that remains now to complete the definitions of our derived norms is to define suitable weight and index functions. This in itself is a non-trivial problem.

4.1 Defining the weight and index functions

Most of the approaches to termination analysis based on norms essentially use a simple generate-and-test method for deducing termination. Norms are generated (either automatically or otherwise) and used to form level mappings which are then applied to the program for which a termination proof is sought. Inequalities are then derived whose solubility indicates the success or failure of the termination proof.

The main difficulty with this approach is the potentially infinite number of norms that can be generated. To reduce the complexity of this problem a number of heuristics can be used. Decorte et al. [6], for example, propose the following (adapted) heuristics for deriving typed norms.

- A weight of one is assigned to all functors of arity $n > 0$.
- A weight of zero is assigned to all constants.
- Any argument position whose type is not a parameter is selected.

Applying these heuristics to our partially derived norms allows us to obtain the same norms that would be derived by [6] given the same type information in the form of a type graph. Although this approach works well on a large number of examples, there are occasions when it will fail to generate norms that can be used in a termination proof. The naive reverse program with an accumulating parameter [6] is one example where a reduced number of arguments needs to be selected. In that paper a solution to this problem is sketched using *symbolic norms* which effectively define an argument index function through an exhaustive search. Also, below we give an example of where constants must be assigned weights other than zero.

Example 4.4 If each constant occurring in the program below is assigned a weight of zero then the interargument relation derived for `Path(`x`, `y`)` would be $|x| = |y| = 0$. With this relationship, termination cannot be proved since we require that $|x| > |z|$ in the recursive `TransitiveClosure/2` clause. To prove termination each constant must take on a different value.

```
TransitiveClosure(x, y) <- Path(x, y).
TransitiveClosure(x, y) <- Path(x, z) & TransitiveClosure(z, y).

Path(a, b).
Path(b, c).
```
□

This example seems to suggest that the determination of weights must take place as an integral part of a termination analysis – the variety of the weights occurring indicates the futility of a generate and test approach in this instance.

In summary, we see that there are several approaches to the problem of deriving the weight and index functions. We do not advocate any particular method here since it is necessary to further investigate and compare suitable methods. We believe that the open-ended definitions of our derived norms should facilitate such a study.

5 Related work

One weakness of [6] is that its norms are derived from type graphs. Type graph analyses, however, have not always been renowned for their tractability. Even for small programs, the prototype analyser of [12], used in [6], is typically 15 times slower than the optimising PLM compiler [15]. Recently, type graph analysis has been shown to be practical for *medium*-sized Prolog programs [14] when augmented with an improved widening and compacting procedure. In addition, Gallagher and de Waal have shown how type graphs can be efficiently represented as unary logic programs in [8]. Clearly, however, any approach which avoids the costs of inferring type graphs is preferable.

Bossi et al. [4] define a very general concept of norm in terms of type schemata which describe structural properties of terms. Their typed norms for termination analysis are very similar to the ones presented in this paper, though they are able to define some norms which cannot be inferred using our present framework.

Example 5.1 Consider the following program from [4]

```
Check(Cons(x, xs)) <- Check(xs).
Check(Cons(x, Nil)) <- Nat(x).
Nat(Succ(x)) <- Nat(x).
Nat(0).
```

We would like to define a norm $|.|_{List(Nat)}$ so that we can prove termination for goals `<- Check(x)` where `x` is rigid wrt $|.|_{List(Nat)}$. The following norm adapted from [4] satisfies this criterion.

$$\begin{array}{rlrlrl}
|v|_{List(Nat)} &= v & |v|_{Nat} &= v & |v|_{Empty} &= v \\
|Cons(t_1, t_2)|_{List(Nat)} &= 1 + |t_2|_{List(Nat)} & |0|_{Nat} &= 0 & |Nil|_{Empty} &= 0 \\
|Cons(t_1, t_2)|_{List(Nat)} &= |t_1|_{Nat} + |t_2|_{Empty} & |Succ(t)|_{Nat} &= 1 + |t|_{Nat} & &
\end{array}$$

This norm cannot be inferred automatically using our method (nor that of [6]) since it relies on the functor $Cons$ having two distinct types, namely $\langle Nat.List(Nat), List(Nat)\rangle$ and $\langle Nat.Empty, List(Nat)\rangle$, but this is forbidden in languages like Gödel where the declarations are universal. Note that this is not a limitation of our framework but rather a limitation of the type system on which it is based. Given a more flexible system it would be possible to infer such norms as the above directly from the prescribed types. □

We note that the typed norms of [4] are not derived automatically. By contrast, our norms, are simple enough to be easily derived using only the type declarations of a program.

6 Conclusions and future work

In this paper, we have presented a flexible method for inferring a number of norms from the type declarations of a program which are sufficient to measure the size of any Herbrand term occurring in the program in an almost arbitrary way. The norms are intended for use in termination analysis and the derivation of inter-argument relationships, though we believe that their applicability is not restricted to these areas. The definition of each derived norm is parameterised by a weight function and an argument index function. This open-ended definition allows the norms to be incorporated into a wide range of analyses which define these functions in different ways. We believe that defining weight and index functions in an efficient and intelligent way is a non-trivial problem in itself. Our definitions of norms provide a useful framework in which to study this problem.

It is our intention to examine exactly how these norms can be integrated into a termination analysis for typed logic programs. With a working termination analysis we will be able to assess the usefulness of the prescribed types in inferring norms. In particular, it would be interesting to quantify how much faster the typed (Gödel) approach is against the untyped (Prolog) approach. We will investigate how to define the weight and index functions such that a minimal number of useful norms are generated and we suspect that analysis can be used to achieve this.

Acknowledgements

We gratefully acknowledge Florence Benoy, John Gallagher, Corin Gurr, John Lloyd, Peter Holst Andersen and David Sands for their stimulating and useful discussions. This work was supported, in part, by the Nuffield grant SCI/180/94/417/G. Jonathan Martin is supported by EPSRC studentship ref. no. 93315269.

References

[1] K.R. Apt and D. Pedreschi. Studies in pure Prolog: Termination. In *Proceedings Esprit symposium on computational logic*, pages 150–176, Brussels, November 1990. Springer-Verlag.

[2] F. Benoy and A. King. Inferring argument size relations with CLP($\mathcal{R}$). In *LOPSTR'96*, 1996.

[3] M. Bezem. Characterizing termination of logic programs with level mappings. In Ewing L. Lusk and Ross A. Overbeek, editors, *Proceedings of the North American Conference on Logic Programming*, pages 69–80, Cleveland, Ohio, USA, 1989.

[4] A. Bossi, N. Cocco, and M. Fabris. Typed norms. In *ESOP'92*, pages 73–92, 1992.

[5] A. Bossi, N. Cocco, and M. Fabris. Norms on terms and their use in proving universal termination of a logic program. *Theoretical Computer Science*, 124:297–328, 1994.

[6] S. Decorte, D. de Schreye, and M. Fabris. Automatic inference of norms: A missing link in automatic termination analysis. In *ILPS'93*, pages 420–436, 1993.

[7] S. Decorte, D. de Schreye, and M. Fabris. Exploiting the power of typed norms in automatic inference of interargument relations. Technical report, Dept. computer science, K.U.Leuven, 1994.

[8] J. Gallagher and A. de Waal. Fast and precise regular approximations of logic programs. In *ICLP'94*, pages 599–613, 1994.

[9] C. Gurr. *A Self-Applicable Partial Evaluator for the Logic Programming Language Gödel*. PhD thesis, University of Bristol, January 1994.

[10] C. Gurr. Personal communication on the literature on termination analyses. September 1995.

[11] P. M. Hill and J. W. Lloyd. *The Gödel Programming Language*. MIT Press, 1994.

[12] G. Janssens and M. Bruynooghe. Deriving descriptions of possible values of program variables by means of abstract interpretation. *J. Logic Programming*, 13:205–258, 1992.

[13] L. Plümer. *Termination Proofs for Logic Programs*. Springer-Verlag, 1990.

[14] P. Van Hentenryck, A. Cortesi, and B. Le Charlier. Type Analysis of Prolog Using Type Graphs. In *PLDI'94*, pages 337–348. ACM Press, 1994.

[15] P. Van Roy. A Prolog Compiler for the PLM. Master's thesis, Computer Science Division, University of California, Berkeley, 1984.