Constraint-Based Termination Analysis of Logic Programs

STEFAAN DECORTE, DANNY DE SCHREYE, and HENK VANDECASTEELE
Katholieke Universiteit Leuven

Current norm-based automatic termination analysis techniques for logic programs can be split up into different components: inference of mode or type information, derivation of models, generation of well-founded orders, and verification of the termination conditions themselves. Although providing high-precision results, these techniques suffer from an efficiency point of view, as several of these analyses are often performed through abstract interpretation. We present a new termination analysis which integrates the various components and produces a set of constraints that, when solvable, identifies successful termination proofs. The proposed method is both efficient and precise. The use of constraint sets enables the propagation of information over all different phases while the need for multiple analyses is considerably reduced.

Categories and Subject Descriptors: I.2.2 [**Artificial Intelligence**]: Automatic Programming; I.2.3 [**Artificial Intelligence**]: Deduction and Theorem Proving

General Terms: Languages, Verification

Additional Key Words and Phrases: Constraint solving, logic programming, termination analysis

1. INTRODUCTION

As we see it, the norm or level mapping-based approach to termination analysis for logic programs has gone through a number of research stages. First important realizations in this area were by Ullman and Van Gelder [1988] and Plümer [1990]. In these works, highly efficient termination analysis procedures were proposed. These methods used fairly simple well-founded orderings, based on *norms* that measure sizes of terms according to the list-length or term-size measures. They also were much "demand-driven," starting from a desired decrease in the well-founded

S. Decorte was supported by GOA, "Non-Standard Applications of Abstract Interpretation," DPWB, Belgium. D. De Schreye is senior research associate of FWO Flanders.
Authors' address: Department of Computer Science, K. U. Leuven, Celestijnenlaan 200A, B-3001 Heverlee, Belgium; email: {stefaan; dannyd; henkv}@cs.kuleuven.ac.be.

order and traversing dependency graphs to actually prove the validity of such a decrease through relations that were set up for intermediate calls to predicates occurring to the left of recursive calls.

These works have been crucial for the entire development of the termination analysis area. Even today, very few methods can compete with them, given that the efficiency-versus-precision trade-off is fully taken into account.

The main drawback of these methods—as can be expected from nicely engineered approaches—is that they have a somewhat ad hoc flavor. At the least, they were less targeted at gaining conceptual insight into the termination problem as a whole, at clarifying the importance of certain subproblems, and at providing conceptually clear frameworks for studying the different trade-offs in termination analysis in general.

A second line of work has filled this gap. The work of Apt, Bezem, and Pedreschi (e.g., see Apt and Bezem [1991], Apt and Pedreschi [1990; 1991], and Bezem [1992]), but also work of others, such as Bossi et al. [1991; 1992], studied the problem from a more mathematical perspective, providing more insight into the various components of a termination analysis proof. This highlighted the need for the well-founded ordering on atoms (usually referred to as *level mappings*), (in the context of termination under the left-to-right selection rule of Prolog) the need for models for the programs, the need for boundedness or rigidity constraints (with respect to the considered level mappings) on the sets of atoms for which termination is proved, and the identification of the well-founding conditions (in terms of decreases of the level mapping on atoms in the given classes) themselves.

This second line of research has clarified more than the early work the difficulties involved:

—the selection of the level mapping, and, in many practical cases, in particular the norm, that measures terms, that gives rise to this level mapping,
—the inference of appropriate models, in automatic termination analysis often referred to as the inference of interargument or size relations over the selected norm,
—the role of the boundedness or rigidity constraints,
—the actual syntactic termination/well-foundedness constraints themselves.

The identification of these different components in the analysis has led to a third line of work. Here, the issue was to revisit the solutions proposed for all these components in the initial termination analysis approaches and to improve on them, at least in terms of precision. This line of research includes work on inferring norms and level mappings from mode and type information [Verschaetse 1992; Decorte et al. 1993; Martin et al. 1996], automatic inference of interargument and size relations [De Schreye and Verschaetse 1995; Brodsky and Sagiv 1991; Lindenstrauss and Sagiv 1997],

automatic verification of the well-founded constraints (for instance, through CLP techniques) [De Schreye et al. 1992; Mesnard 1996].

At least on the level of automatic termination analysis, the current status of all this work can be evaluated as follows. With respect to precision, recent works outperform the results obtainable with the early approaches. More refined well-foundings are proposed. More precise interargument relations are computed. More programs can be proved to terminate. At the level of efficiency, the situation is reverse. More recent automated termination analysis techniques, such as Decorte et al. [1993], De Schreye et al. [1992], and Lindenstrauss and Sagiv [1997], require several stages in the analysis: to infer mode and type information, to infer appropriate norms and level mappings (using the modes and types) that satisfy rigidity constraints, to infer interargument relations, and finally to prove the well-foundedness conditions themselves. This results in very time-consuming analysis, which may be unrealistic in view of developing practical program verification tools. Moreover, there is still a lack of precision. Due to sequentializing the analysis in distinguished steps: mode/type inference, norm/level mapping inference, interargument inference, termination conditions verification, the analysis ceases to be demand-driven. The actual "well-founding" termination conditions (e.g., the actual decrease of the level mapping between the head and the body atoms of the different clauses) is *not* taken into account for the generation of the appropriate norm, level mapping, and interargument relations. The analysis has become bottom-up instead of top-down.

In this article, we revise the original demand-driven termination analysis methods, taking the conceptual insights on the different layers and their optimization into account. This leads to a new analysis in which, starting from the well-foundedness constraints, from the rigidity constraints, from our current understanding of useful norms and level mappings, and from our understanding of interargument relations, we automatically set up

—a symbolic, parametrized form of all the concepts involved (the norm, the level mapping, the interargument relations),
—a set of constraints on the parameters in these concepts, sufficient to obtain a termination proof.

By solving the generated constraints, we get a demand-driven solution for all the intermediate concepts involved in the analysis. More precisely, if a norm, level mapping, and interargument relation of the given generic forms exist such that the program can be proved to terminate, then our analysis will generate the required instance of the generic object that proves the termination.

The new approach is both precise and efficient. The efficiency results from the fact that information over the different stages of the analysis is now propagated at once, starting from a general constraint set. Due to this general propagation scheme, the enforced decisions at one component of the analysis quickly narrow the options for other components, which signifi-

cantly speeds up the analysis as a whole. Moreover, the analysis is now performed in one single step, eliminating the need for a number of different (abstract interpretation) phases.

The precision is due to the fact that we no longer need to fix certain choices (e.g., which norm to use or which interargument relation to select) before other stages of the analysis are started. Choices are postponed until the constraint propagation forces the parameters to take specific values. As a result, we avoid making the wrong choices. In theory, this is another efficiency gain: we avoid the need for backtracking over previous choices. In practice, backtracking over all possible choices is too inefficient, and practical systems only consider a limited number of all possible options (e.g., they might only consider the term-size and the list-length norm). In this respect, our approach is more precise.

The structure of this article is as follows. In the next section, we present some preliminaries on norms, level mappings, interargument relations, and acceptability. In Section 3, we provide the basic motivation for our research. In Section 4, we illustrate the main intuitions of our analysis through an example. All introduced concepts are formalized in Section 5, leading to a symbolic termination condition. In Section 6, we propose a technique to derive simple constraints from the symbolic condition, and we discuss how to solve them. We illustrate the practicality of the approach on several examples in Section 7. In Section 8 we comment on experiments performed on a prototype system. We discuss related work in Section 9, and we end with a discussion.

2. PRELIMINARIES

2.1 Notational Conventions and Terminology

In the following, we assume familiarity with the terminology, the basic concepts, and the main results of logic programming, as they are, for instance, presented in Apt [1990] or Lloyd [1987].

By $\mathcal{L}_P$, we denote the first-order language associated to a program P. We use $Const_P$, Fun_P, $Pred_P$, and Var_P to denote respectively the set of constants, the set of function symbols (without the constants), the set of predicate symbols, and the set of variables of this language. In this article we extend the definition of Fun_P to also include all functor symbols which might occur in queries for P. The same holds for $Const_P$.

We use the following notational conventions. Variables, functors, and predicate symbols start with a lowercase character. Constants start with an uppercase character. The Prolog notation p/n is used to represent a predicate with symbol p and arity n. If there is no risk of confusion, we simply write p. We also use the Prolog conventions for representing lists. We represent substitutions as sets of bindings $\{x_1/t_1, \cdots, x_n/t_n\}$, and we denote them by Greek characters. N-tuples of indexed objects—e.g., variables or terms—$(x_1, x_2, \ldots, x_n)$ are denoted as $\bar{x}$.

Given a program P, we denote its Herbrand Universe by U_P and its Herbrand Base by B_P. The extended Herbrand Universe, U_P^E, and the extended Herbrand Base, B_P^E, associated to a program P, were introduced in Falaschi et al. [1989]. They are defined as follows. Let $Term_P$ and $Atom_P$ denote the sets of, respectively, all terms and all atoms that can be constructed from the alphabet underlying to P. The variant relation, denoted $\sim$, defines an equivalence. U_P^E and B_P^E are respectively the quotient sets $Term_P/\sim$ and $Atom_P/\sim$. For any term t (or atom A), we denote its class in U_P^E (B_P^E) as $[t]$ ($[A]$). However, in order to reduce notational complexity, we drop the brackets when no real confusion is possible.

If E and F are two expressions (a term, atom, n-tuple of terms, or n-tuple of atoms) of the same type and the expressions unify, then $mgu(E, F)$ denotes their most general unifier. $\models$ denotes logical consequence in the first-order language. For any formula F, $\forall F$ and $\exists F$ denote the universal, respectively existential, closure over the free variables occurring in F.

From this section onward, we denote a query as the sequence of its literals, separating literals by commas. This is convenient, since it allows us to view sets of atomic queries as sets of atoms. We take the convention that a program *does not* include a query. A pair (P, Q) consisting of a program P and a query Q is referred to as a *queried program*.

Whenever we talk about termination, we mean *left-termination*, which is termination with respect to the left-to-right selection rule only.

2.2 Norms and Level Mappings

An essential component of any termination analysis framework is the ability of measuring and comparing atoms and terms. The "size" of a term is found by using *norms*.

Definition 2.1 (*Norm*). A *norm* is a mapping $\|.\| : U_P^E \to \mathbb{N}$.

Observe that norms are defined on the *extended Herbrand Universe*. With slight abuse of notation, we will often write $\|t\|$, with $t \in Term_P$.

Several examples of norms can be found in the literature. Two well-known instances are the list-length norm, which maps lists to their length, and the term-size norm, which counts the number of functors in a term.

Definition 2.2 (*List-Length*, *Term-Size*). The *list-length* norm, denoted $\|.\|_l$, is defined in the following way:

$$\begin{aligned} \|[t_1|t_2]\|_l &= 1 + \|t_2\|_l, && \text{with } t_1 \text{ and } t_2 \text{ any term,} \\ \|t\|_l &= 0 && \text{otherwise.} \end{aligned}$$

The *term-size* norm, denoted $\|.\|_t$, is defined in the following way:

$$\begin{aligned} \|f(t_1, \ldots, t_n)\|_t &= 1 + \textstyle\sum_{1 \le i \le n} \|t_i\|_t, && \text{with } f \text{ any function symbol and } n > 0, \\ \|t\|_t &= 0 && \text{otherwise.} \end{aligned}$$

Studying the termination of queries containing nonground terms is known to cause problems in the analysis. As restricting to ground queries is too limited, automatic termination analysis has resorted to concepts as boundedness [Bezem 1992] or rigidity [Bossi et al. 1991]. Only the latter property will be of further relevance.

Definition 2.3 (*Rigid Term* [*Bossi et al. 1991*]). Let $\|.\|$ be a norm and t be a term. We say that t is *rigid* with respect to $\|.\|$ if, for any substitution σ, $\|t\sigma\| = \|t\|$.

If a term is known to be rigid with respect to a given norm, it can be considered a ground term: further substitutions can no longer affect its size. Notice, however, that this property depends on the selected norm. In light of the preceding discussion, the following class of norms, called semilinear norms, are especially relevant for practical applications of termination analysis. They allow us to detect syntactically whether or not a term is rigid with respect to a norm.

Definition 2.4 (*Semilinear Norm* [*Bossi et al. 1991*]). A norm $\|.\|$ is *semilinear* if it is recursively defined by means of the following schema:

$$\begin{aligned} \|v\| &= 0 \text{ if } v \text{ is a variable, and} \\ \|f(t_1, \ldots, t_n)\| &= c + \|t_{i_1}\| + \cdots + \|t_{i_m}\|, \\ &\quad \text{with } c \in \mathbb{N},\ \{i_1, \ldots, i_m\} \subseteq \{1, \ldots, n\} \\ &\quad \text{and } c, i_1, \ldots, i_m \text{ depending only on } f/n,\ n \geq 0. \end{aligned}$$

Of the examples above, both the list-length and the term-size norm satisfy this property. In Bossi et al. [1991] it is shown how semilinear norms allow us to detect rigid terms syntactically. In this article we will consider norms which are a slight variant of semilinear ones. All norms in this article take the following form:

$$\begin{aligned} \|t\| &= 0 \text{ if } t \text{ is a variable or a constant, and} \\ \|f(t_1, \ldots, t_n)\| &= f_0 + \sum_{i=1}^{n} f_i \|t_i\| \\ &\quad \text{where } f_i \in \mathbb{N},\ i \in \{0, \cdots, n\} \text{ and depend only on} \\ &\quad f/n,\ n > 0. \end{aligned}$$

To measure the size of atoms, level mappings were introduced.

Definition 2.5 (*Level Mapping*). A level mapping is a function $|.| : B_P^E \rightarrow \mathbb{N}$.

A level mapping assigns to each atom a natural number which can be interpreted as its *size*. Analogously to the concept of a rigid term, we define an atom A to be rigid with respect to a level mapping $|.|$ if its size under $|.|$ is invariant under substitution. We say that a level mapping $|.|$ is rigid on a set of atoms S if each atom in S is rigid with respect to $|.|$.

In practice, level mappings are most often defined as a linear combination of the sizes of the terms on fixed (per-predicate) argument positions. That is, it is defined as

$$|(p(t_1, \cdots, t_n)| = \sum_{i \in I_p} \|t_i\|, \quad \text{for some } I_p \subseteq \{1, \cdots, n\} \text{ and for some norm } \|.\|.$$

The set I_p is frequently selected to be the set of input positions. Verschaetse [1992] describes how to automatically propose the set I_p and calls the resulting level mappings *natural level mappings*. We return to the concept with more detail in Section 4.

2.3 Acceptability with Respect to a Set of Atoms

In automatic termination analysis, it is common practice that the behavior of a program must be investigated for some set of possible calls to the defined predicates. To simplify the discussion, we make the assumption that this set of queries consists of atomic queries only. Obviously, this is not a limitation, as conjunctive goals can be reduced to the atomic case by adding definitions for new predicates to the program with the conjunction in their body.

In this subsection we briefly describe the main ideas of the termination analysis framework of De Schreye et al. [1992]. We present the key notions of De Schreye et al. [1992] here only in the context of directly recursive programs. Full details on the framework can be found in De Schreye and Verschaetse [1992].

Definition 2.6 (*Call Set*). Let P be a definite program and S a set of atomic queries. 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 practice, the set of queries S will be specified as a call pattern, using abstraction. Commonly, mode or type abstraction mechanisms are used.

The following definition is a key concept in De Schreye et al. [1992].

Definition 2.7 (*Acceptability with Respect to S*). Let S be a set of atomic queries and P a definite directly recursive program. P is *acceptable with respect to S* if there exists a level mapping $|.|$ such that

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

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

The following proposition of De Schreye et al. [1992] characterizes left-termination in terms of acceptability.

PROPOSITION 2.8. *A directly recursive program P terminates under the left-to-right selection rule for any query in S if and only if P is acceptable with respect to S.*

Because of the undecidability of the termination problem, verifying whether a program is acceptable with respect to a set of atoms is, in general, undecidable too. The main difficulty is in providing a suitable level mapping. Still, the proposition will hand us a practical method for checking the termination of a program with respect to a fixed set of queries, as shown later in Proposition 4.1.

2.4 Interargument Relations

A final point to be addressed by automatic termination analysis techniques concerns the computed answers which appear in Definition 2.7. In several practical approaches these are provided under the form of *interargument relations*.

Definition 2.9 ((*Valid*) *Interargument Relation*). Let P be a definite program and p/n a predicate in P. An *interargument relation for p/n* is a relation $R_p \subseteq \mathbb{N}^n$. R_p is a *valid interargument relation for p/n with respect to a norm* $\|.\|$ if and only if for every $p(t_1, \cdots, t_n) \in Atom_P$: if $P \models p(t_1, \cdots, t_n)$ then $(\|t_1\|, \cdots, \|t_n\|) \in R_p$.

Example 2.10 (*Delete*). Consider the following well-known delete program:

$delete(X, [X|T], T).$
$delete(X, [H|T], [H|T']) \leftarrow delete(X, T, T').$

A few interargument relations are

$$\begin{aligned}
(R^1_{delete}) &: \{(x_1, x_2, x_3) | x_2 \geq x_1 + x_3\} \\
(R^2_{delete}) &: \{(x_1, x_2, x_3) | x_1 = x_2 + x_3\} \\
(R^3_{delete}) &: \{(x_1, x_2, x_3) | x_2 = x_3 + 1\} \\
(R^4_{delete}) &: \{(x_1, x_2, x_3) | x_2 < x_3\}
\end{aligned}$$

The first interargument relation is valid with respect to the term-size norm. R^2_{delete} is a valid interargument relation for delete with respect to the constant zero-norm $\|.\|_{zero} : U^E_P \rightarrow \mathbb{N} : t \rightarrow 0$. Interargument relation R^3_{delete} is valid with respect to the list-length norm. Finally, it is impossible to find a norm $\|.\|$ such that the last interargument relation is valid with respect to $\|.\|$, as the following reasoning illustrates.

Let us suppose that R^4_{delete} is valid with respect to a norm $\|.\|$. As $\|.\|$ is a mapping from terms to natural numbers, we can find a term $t_2 \in Term_P$ such that $\|t_2\|$ is minimal. Then taking any term $t_1 \in Term_P$, the atom

$delete(t_1, [t_1|t_2], t_2)$ is entailed by the delete program. But then, $\|[t_1|t_2]\| < \|t_2\|$, which is in contrast to our assumption of minimality of $\|.\|$ on t_2.

In the remainder of this article, we restrict ourselves to interargument relations which express an inequality relation (it will turn out that in most of the cases these are exactly the kind of relations we need for a successful termination proof).

2.5 Termination Analysis Illustrated

Let us illustrate all these concepts by means of an example. We will also use this example throughout the following sections. In order not to obscure the issues through the complexity of the example, and to facilitate comparison with other approaches, we select the very "standard" permute example.

Example 2.11 (*Permute*).

$$\begin{aligned} permute(Nil, Nil) &\leftarrow \\ permute(l, [el|t]) &\leftarrow delete(el, l, l_1), permute(l_1, t). \\ delete(x, [x|t], t) &\leftarrow \\ delete(x, [h|t], [h|t']) &\leftarrow delete(x, t, t'). \end{aligned}$$

Suppose we would like to prove that permute terminates whenever it is called with a nil-terminated list of any terms as its first argument and a free variable on the second position. Note that calling permute in this way issues calls to delete where the first argument is any term, where the second argument is a nil-terminated list, and where the third argument is a free variable.

In an (automatic) norm-based termination proof, one typically starts from a fixed norm $\|.\|$, which could in our example be the list-length norm $\|.\|_l$, which was defined in the previous section. From this norm a level mapping $|.|$ is proposed. As the input to the permute and delete atoms is located in respectively their first and second argument, one would naturally select a level mapping:

$$|permute(t_1, t_2)| = \|t_1\|_l$$

$$|delete(t_1, t_2, t_3)| = \|t_2\|_l$$

Proving termination for delete is straightforward. Take any call to delete satisfying the above-specified type of calls and which unifies with the head of the second clause. Let us say $delete(t_1, t_2, t_3)$, and let $\theta = mgu(delete(t_1, t_2, t_3), delete(x, [h|t], [h|t']))$. It is then easy to see that $|delete(t_1, t_2, t_3)| > |delete(x, t, t')\theta|$. Because of rigidity, $|delete(t_1, t_2, t_3)| = |delete(t_1, t_2, t_3)\theta|$ and $|delete(t_1, t_2, t_3)\theta| = |delete(x, [h|t], [h|t'])\theta| = \|[h|t]\theta\|_l = 1 + \|t\theta\|_l = 1 + |delete(x, t, t')\theta|$.

For the case of permute, a valid interargument relation for delete/3 with respect to the list-length norm must be generated. Using abstract interpre-

tation, for example, the following valid relation (with respect to the list-length norm) can be derived:

$$\{(n_1, n_2, n_3) | n_2 = n_3 + 1\}$$

At this point, all ingredients to show termination of permute are available. Take any permute call $permute(t_1, v)$, where t_1 is a nil-terminated list, and which unifies with the recursive clause, and let $\theta \equiv mgu(permute(t_1, v), permute(l, [el|t]))$. Then $|permute(t_1, v)| = |permute(t_1, v)\theta| = |permute(l, [el|t])\theta| = \|l\theta\|_l$, and we must prove that $\|l\theta\|_l > \|l_1\theta\sigma\|_l = |permute(l_1, t)\theta\sigma|$, for any computed answer substitution σ for $delete(el, l, l_1)\theta$. Here comes the interargument relation in play, as for such delete atoms, $\|l\theta\sigma\|_l = \|l_1\theta\sigma\|_l + 1$.

3. REFINED MOTIVATION

Reconsider the permute program of Example 2.11, and let us summarize the important observations. When using a traditional norm-based termination analysis, typically three sources of information must be provided. One normally starts from a fixed norm $\|.\|$, which, in our example, was the list-length norm $\|.\|_l$. The next step involves proposing a level mapping $|.|$ from this norm. A useful level mapping is one that is rigid on all possible calls. In addition, for the case of permute, a valid interargument relation for delete/3 with respect to the list-length norm must be generated.

The sketch above immediately highlights three major problems with this kind of automatic proof method. Each of these issues is an immediate consequence of considering these inputs as unrelated to each other: feeding the wrong inputs results in a failed termination proof. First of all, if the analysis had been started from another norm, like the term-size norm, termination could never have been proved. Apart from the trivial zero-mapping, there exist no other rigid level mappings with respect to the term-size norm and the above-proposed set of calls.

Similarly, on the level of the interargument relations, the selection of an appropriate norm can have a major impact. Consider the following example of Verschaetse [1992], where it is used as part of a program to compute the power-set of a given set.

Example 3.1 (*Combine*). Combine adds its first argument at the beginning of all lists appearing in the term occurring on the second argument position:

$$\begin{aligned} combine(l, Nil, Nil) &\leftarrow \\ combine(x, [h|t], [[x|h]|u]) &\leftarrow combine(x, t, u) \end{aligned}$$

When trying to derive a linear interargument relation for combine from the list-length norm, the relation.

$$\{(k, n, m) \in \mathbb{N}^3 | n = m\}$$

is valid. However, if term-size was selected, the analysis would be unable[1] to infer any sensible interargument relation.

Several techniques, such as Ullman and Van Gelder [1988], are tailored to the use of one specific norm (for Ullman and Van Gelder [1988], list-length). Others, such as Verschaetse [1992] and Lindenstrauss and Sagiv [1997] provide the user with the possibility of specifying a desired norm, tuned to the program and query at hand. However, the above discussion severely underlines the impact that the selection of a norm has on the different components in the termination proof. Fixing a norm from which to start the analysis has a constraining effect on all other components.

The strong dependency on the choice (the definition) of the norm to measure terms for success or failure of a termination proof has previously been observed in several works, such as Bossi et al. [1992], Bronsard et al. [1992], and Verschaetse [1992]. The solution proposed has been to increase the precision of the norms through the integration of type information. Still, the (even extensive) use of type information does not facilitate to make a justified guess in favor of, for example, term-size or list-length norms when ground information is concerned.

Second, even if a correct selection of a norm has been performed, a successful termination proof can be missed because the level mapping defined from it is not adequate. The following example illustrates this point:

Example 3.2. Consider the following *reverse* predicate, which reverses lists through the use of an accumulating parameter appearing as the third argument of the *revacc* predicate.

$$\begin{aligned} reverse(l, l_r) &\leftarrow revacc(l, l_r, Nil). \\ revacc(Nil, l, l) &\leftarrow \\ revacc([el|t], r, a) &\leftarrow revacc(t, r, [el|a]). \end{aligned}$$

It is not difficult to see that any calls to *reverse* terminate whenever the predicate is used to reverse a ground list (on first argument position). In this case, a perfectly natural level mapping is the following:

$$|revacc(l_1, l_2, l_3)| = \|l_1\|_t + \|l_3\|_t$$

However, such a level mapping does not allow to prove a decrease between a call $revacc([el|l_1], l_2, l_3)$ and the corresponding recursive call $revacc(l_1, l_2, [el|l_3])$. The accumulator is responsible for this behavior, and a level mapping which does not consider this accumulator is necessary:

$$|revacc(l_1, l_2, l_3)| = \|l_1\|_t$$

[1]Our course, $\{(k, n, m) \in IN^3 | m \leq n\}$ is a valid interargument relation for combine with respect to the term-size norm, but it cannot be inferred in the setting of De Schreye and Verschaetse [1995]. Only linear interargument relations expressing an equality relation can.

Finally, even in cases where a reasonable norm and level mapping have been proposed, the analysis may fail because of the derivation of unsuitable interargument relations. To illustrate this point, reconsider the permute program of Example 2.11, and this time let us investigate termination for calls to permute with both arguments ground. As both arguments are ground, a good candidate norm to base the proof on is the term-size norm. To fix a level mapping on this norm, we choose to measure permute calls by regarding their first argument only. A perfectly natural and detailed valid interargument relation for delete with respect to the term-size norm is the following:

$$\{(x_1, x_2, x_3) | x_1 + x_2 \geq x_3 + 1\} \tag{1}$$

However, the relation takes too much information into account, and proving termination becomes impossible. There are two solutions: either the level mapping must be extended on permute atoms to measure both arguments, or the interargument relation must be weakened:

$$\{(x_1, x_2, x_3) | x_2 \geq x_3 + 1\}$$

From the examples it should be clear that there exists a lot of interdependency between the three central concepts of a termination analysis. It might, therefore, not be such a good idea to provide the level mapping, norm, and interargument relations more or less independently from one another. This is the key idea for the rest of this article where we work the other way around. At the beginning of the analysis, we still do not assume anything known about the level mapping. Neither do we make any assumptions about the norm or the interargument relations which depend on such a norm. Instead, we work at a higher level by providing generic forms of all three concepts. In the remainder of this article, we then propose a method for automatically deriving a suitable norm, level mapping, and the necessary interargument relations from which to start the analysis. In the following, the framework is formulated for direct recursive programs only. This allows us to focus on the essence of the approach, free from the resulting notational complexity. We immediately want to emphasize that this is just a presentation matter and does not impose any restriction upon the applicability of the proposal. To begin with, most of the time a mutually recursive program can be transformed into a directly recursive one. We refer to the extensive discussion in Plümer [1990] on how termination analysis for indirectly recursive programs can most often be solved using techniques for directly recursive ones. Next, the concepts proposed for the technique can be reformulated for the mutually recursive case by moving from the clause level to more general syntactic structures, as there are the cyclic collections of the program (see De Schreye et al. [1992]) or its strongly connected components.

4. TERMINATION ANALYSIS REVISITED

In the remainder of this article, we make the assumption, for each predicate in the program, that each of its argument positions is fixed as being an input or an output position. A generic view on input/output applies here: input arguments do not have to correspond to ground terms, neither do output arguments have to be free. This is completely in the line of reasoning of well-typed programs, as introduced by Bronsard et al. [1992]. If the program is not well-moded nor well-typed, the positions in a predicate corresponding to these generalized notions of input/output can be inferred through an inspection of its call types. We refer to Decorte et al. [1993] where the problem has been discussed in the context of type information, and heuristics for appropriate selection of both sets are given.

First, the following notion of acceptability, derived from the notion in Definition 2.7, is very useful.

PROPOSITION 4.1 (RIGID ACCEPTABILITY WITH RESPECT TO S). Let S be a set of atomic queries and P a definite, directly recursive program. Let $\|.\|$ be a norm, and, for each predicate p in P, let R_p be a valid interargument relation for p with respect to $\|.\|$. If there exists a level mapping $|.|$ which is rigid on $Call(P, S)$ such that

—for any clause $H \leftarrow B_1, \cdots, B_n \in P$, and
—for any atom B_i in its body which has the same predicate symbol as H,
—for substitution θ such that the arguments of the atoms in $(B_1, \cdots, B_{i-1})\theta$ all satisfy their associated interargument relations $R_{B_1}, \cdots, R_{B_{i-1}}$

$$|H\theta| > |B_i\theta|,$$

then P is acceptable with respect to S.

PROOF. Suppose the above condition is satisfied for P. Take any $A \in Call(P, S)$ and any clause $A' \leftarrow B_1, \cdots B_{i-1}, B_i, B_{i+1}, \cdots B_n$ such that $mgu(A, A') = \theta$. Suppose A' and B_i have the same predicate symbol and suppose that $(B_1, \cdots, B_{i-1})\theta\sigma$ is a computed instance of $(B_1, \cdots, B_{i-1})\theta$. Because $A\theta = A'\theta$, $|A\theta\sigma| = |A'\theta\sigma|$. Now, since $A \in Call(P, S)$, and $|.|$ is rigid on $Call(P, S)$, then $|A\theta\sigma| = |A|$. Thus, $|A| = |A'\theta\sigma|$.

Finally, since $(B_1, \cdots, B_{i-1})\theta\sigma$ is a computed instance, $P \models B_j\theta\sigma$, for all $j < i$. Thus, by definition of valid interargument relation, the arguments of $B_j\theta\sigma$ satisfy R_{B_j}, for all $j < i$. Then, by the rigid acceptability assumption, $|A'\theta\sigma| > |B_i\theta\sigma|$. Combined with $|A\theta\sigma| = |A|$, we get $|A| > |B_i\theta\sigma|$. □

Observe that through the rigidity constraint on the level mapping we are no longer forced to reason on "calls" in the termination condition. The condition is now fully at the clause level. Secondly, the condition on computed answers has been generalized to one on instances satisfying the interargument relations.

Also note that the condition is not unreasonably too strong. For example, consider the well-known append program:

$$\begin{aligned} \text{append}(Nil,\ l,\ l) &\leftarrow \\ \text{append}([el|l_1],\ l_2,\ [el|l_3]) &\leftarrow \text{append}(l_1,\ l_2,\ l_3) \end{aligned}$$

A level mapping measuring the list-length of the first or third argument (or both) will satisfy rigid acceptability. Any call set on which this mapping is rigid will be terminating (e.g., append queries where the first or the third argument (or both) is a nil-terminated list). However, in general, the condition is not a necessary one.

In the remainder of this section, we illustrate the underlying intuitions that form the basis for our new approach through an example. Reconsider the permute program of Example 2.11. Suppose that we aim to prove left-termination for all atomic goals in the set

$$S = \{\text{permute}(t_1,\ t_2) |\ \begin{array}{l} t_1 \text{ is a nil-terminated list and} \\ t_2 \text{ is a free variable}\}. \end{array}$$

Initially, a separate analysis is needed to determine the set $Call(P, S)$. In this particular example, type inference (for instance, through the abstract interpretation of Janssens and Bruynooghe [1992]) allows us to determine that $Call(P, S)$ is the set

$$S \cup \{\text{delete}(t_1,\ t_2,\ t_3) |\ \begin{array}{l} t_1 \text{ and } t_3 \text{ are free variables and} \\ t_2 \text{ is a nil-terminated list}\}. \end{array}$$

The analysis starts by setting up symbolic versions of all concepts needed in the analysis, followed by a formulation of all conditions on these symbolic expressions, and finally, by solving the constraints these conditions entail.

We first fix a symbolic semilinear norm:

$$\|[t_1|t_2]\| = \bullet_0 + \bullet_1\|t_1\| + \bullet_2\|t_2\|, \quad \text{for any terms } t_1 \text{ and } t_2.$$

Since the list-constructor is the only functor occurring in the program, we leave $\|.\|$ unspecified elsewhere. The symbols $\bullet_0$, $\bullet_1$, and $\bullet_2$ denote natural numbers. The purpose of the analysis is to determine their value in such a way that a successful termination proof can be based on $\|.\|$.

Similarly, we introduce a symbolic version of the level mapping $|.|$:

$$\begin{aligned} |delete(t_1,\ t_2,\ t_3)| &= d_1\|t_1\| + d_2\|t_2\| + d_3\|t_3\| \text{ and} \\ |permute(t_1,\ t_2)| &= p_1\|t_1\| + p_2\|t_2\| \end{aligned}$$

where t_1, t_2, and t_3 again denote any terms, and d_1, d_2, d_3, p_1, and p_2 denote natural numbers that need to be determined.

Finally, a symbolic interargument relation for delete is needed. In this article, we allow interargument relations which express an inequality relation of the following form:

$$R^{p/n} = \left\{ (x_1, \cdots, x_n) \;\middle|\; \sum_{i \in I_p} p_i^e \, x_i \geq \sum_{j \in J_p} p_j^e \, x_j + p_0^e \right\},$$

with $p_i^e \in \mathbb{N}$, $i \in \{1, \cdots, n\}$, $I_p \cup J_p \subseteq \{1, \cdots, n\}$, and $I_p \cap J_p = \emptyset$. The sets I_p and J_p are the sets of generalized input and output arguments, which were mentioned at the beginning of this section.

Back to the example, the sets I_{delete} and J_{delete} have not yet been fixed. Following the heuristics proposed in Decorte et al. [1993], the heart of which states that only argument positions which are never called with a free variable can be considered as "input," we propose $I_{delete} = \{2\}$ and $J_{delete} = \{1, 3\}$. This gives rise to the symbolic interargument relation

$$\{(n_1, n_2, n_3) | d_2^e \, n_2 \geq d_1^e \, n_1 + d_3^e \, n_3 + d_0^e\}.$$

Next, we formulate the conditions needed for a successful termination proof, and we set up the constraints on the introduced symbolic coefficients from them. In general, there are three conditions, following immediately from the rigid acceptability (Proposition 4.1):

—the atoms in $Call(P, S)$ must be rigid with respect to the level mapping,
—any introduced interargument relation must be valid,
—the rigid acceptability condition, with respect to this level mapping and interargument relations, must hold.

Let us consider the first condition. Recall from Section 2.2 that the theory of semilinear norms was especially designed to characterize rigid terms and atoms in a syntactic way. Specifically, (sub)argument positions which are taken into account by the norm or level mapping should never contain a variable.

For delete(t_1, t_2, t_3) atoms in $Call(P, S)$, variables may occur for t_1 and t_3. Moreover, if t_2 is of the form $[t_1' | t_2']$, then t_1' may also be a variable. Thus, rigidity of the level mapping imposes $d_1 = d_3 = \bullet_1 = 0$. Similarly, for the permute predicate, rigidity enforces the constraint $\bullet_1 = p_2 = 0$. As a result, the symbolic norm and level mapping reduce to

$$\begin{aligned} \|[t_1|t_2]\| &= \bullet_0 + \bullet_2 \|t_2\| \\ |delete(t_1, t_2, t_3)| &= d_2 \|t_2\| \\ |permute(t_1, t_2)| &= p_1 \|t_1\| \end{aligned}$$

Next, the interargument relation for delete must be valid. In several previous works, (e.g., De Schreye and Verschaetse [1995], Bruynooghe and Boulanger [1994], and Cousot and Halbwachs [1978]) inference of valid interargument relations was performed through abstract interpretation

and was computationally rather expensive. Here, we solve this problem by considering our proposed interargument relation(s) as a generalized (nonground) interpretation, I, of the program, and by imposing (as a condition on the symbolic coefficients) that $T_P(I) \subseteq I$ should hold, where T_P is the generalized (nonground) immediate consequence operator of Falaschi et al. [1989].

In the case of delete, this gives rise to the conditions

$$\forall x, t : d_2^e\|[x|t]\| \geq d_1^e\|x\| + d_3^e\|t\| + d_0^e \text{ (for the nonrecursive clause)}$$

and

$$\forall x, h, t, t' : d_2^e\|t\| \geq d_1^e\|x\| + d_3^e\|t'\| + d_0^e \Rightarrow$$

$$d_2^e\|[h|t]\| \geq d_1^e\|x\| + d_3^e\|[h|t']\| + d_0^e.$$

Combining these conditions with those for $\|.\|$ simplifies to

$$\forall x, t : (d_2^e \bullet_2 - d_3^e)\|t\| - d_1^e\|x\| - d_0^e + d_2^e \bullet_0 \geq 0 \text{ and}$$
$$\forall x, h, t, t' : d_2^e\|t\| - d_1^e\|x\| - d_3^e\|t'\| - d_0^e \geq 0 \Rightarrow$$

$$d_2^e \bullet_2\|t\| - d_1^e\|x\| - d_3^e \bullet_2\|t'\| - d_0^e + \bullet_0(d_2^e - d_3^e) \geq 0.$$

Reducing such conditions to explicit constraints on d_0^e, d_1^e, d_2^e, d_3^e, $\bullet_0$ and $\bullet_2$ is a nontrivial matter. As far as we know, no general approach for dealing with such constraints exists. In Section 6 we propose a specific reduction method, based on combining such conditions to infer sufficient preconditions in terms of basic constraints on the symbolic coefficients.

Finally, consider the rigid acceptability condition. For delete, only the recursive clause imposes a condition, namely that

$$\forall x, h, t, t' : |delete(x, [h|t], [h|t'])| > |delete(x, t, t')|.$$

Simplifying both sides we get

$$|delete(x, [h|t], [h|t'])| = d_2\|[h|t]\| = d_2\bullet_0 + d_2\bullet_2\|t\| \text{ and}$$

$$|delete(x, t, t')| = d_2\|t\|$$

Thus the condition is $d_2\bullet_0 + d_2\bullet_2\|t\| > d_2\|t\|$ or $d_2(\bullet_0 + (\bullet_2 - 1)\|t\|) > 0$. Since t ranges over all nil-terminated lists, $\|t\|$ will take infinitely many values in the natural numbers, including 0 (for sensible norms that count the empty list as 0), so that this can be further reduced to $d_2 > 0$, $\bullet_0 > 0$, and $\bullet_2 \geq 1$. Choosing any remaining values for d_2, $\bullet_0$, and $\bullet_2$ already yields a successful termination proof for delete. For instance, $d_2 = \bullet_0 = \bullet_2 = 1$, with list-length as the resulting norm, and $|delete(t_1, t_2, t_3)| = \|t_2\|_l$ will do.

The treatment of predicate permute itself is more complex. Rigid acceptability states, that, if $delete(el, l, l_1)\theta$ satisfies the interargument relation

(under $\|.\|$), then

$$|permute(l, [el|t])\theta| > |permute(l_1, t)\theta|.$$

Using the form of the imposed interargument relation for delete and the definitions of $|.|$ and $\|.\|$, this reduces to

$$d_2^e\|l\theta\| \geq d_1^e\|el\theta\| + d_3^e\|l_1\theta\| + d_0^e \Rightarrow p_1\|l\theta\| > p_1\|l_1\theta\|.$$

Again, the special reduction method of Section 6 will simplify this condition into constraints on the coefficients.

In the remainder of the article, we reexamine the termination condition which was presented in this section, and we reformulate it as a condition in terms of symbolic counterparts of level mapping, norm, and interargument relation. Using two simple rewrite rules, it is shown that the condition can be rewritten into a solved form. For such solved forms, constraints on the symbols can be derived without much difficulty, and algorithms for constraint derivation are available. The termination condition is only one source of constraints. In an analogous way, constraints are derived whenever interargument relations are necessary. An inspection of the call set provides a final subset of constraints. Such a system of constraints identifies sets of suitable level mappings, norms, and interargument relations which can be used in the original termination condition. In other words, if a solution for the constraint system exists, termination can be proved.

5. SETTING UP THE SYMBOLIC CONDITIONS

In this section, we formalize the notions introduced in the example. We first define symbolic versions of semilinear norms, level mappings, and interargument relations. These will then form the basis for a symbolic termination condition. The condition will be formulated as a search for suitable mappings of all introduced symbols. In the next section, we will describe how this search can be automated efficiently.

5.1 Symbolic and Natural Formulae

Definition 5.1 (*Functor*, *Predicate*, *and Extended Predicate Coefficients*). The set of *functor coefficients*, respectively *predicate coefficients*, respectively *extended predicate coefficients*, associated to a program P are the sets of symbols

$$\begin{aligned} FC(P) &= \{f_i|f/n \in Fun_P \wedge i \in \{0, \ldots, n\}\} \\ PC(P) &= \{p_i|p/n \in Pred_P \wedge i \in \{1, \ldots, n\}\} \\ EC(P) &= \{p_i^e|p/n \in Pred_P \wedge i \in \{0, \ldots, n\}\} \end{aligned}$$

Intuitively, $FC(P)$, $PC(P)$, and $EC(P)$ are the sets of parameters we intend to compute for, respectively, the symbolic norm, the symbolic level mapping, and the interargument relations.

Example 5.2. Reconsider the above permute program. It has the following sets of coefficients:

$$\begin{aligned} FC(\text{permute}) &= \{\bullet_0, \bullet_1, \bullet_2\}, \\ PC(\text{permute}) &= \{\text{permute}_1, \text{permute}_2, \text{delete}_1, \text{delete}_2, \text{delete}_3\}, \\ EC(\text{permute}) &= \{\text{permute}_0^e, \text{permute}_1^e, \text{permute}_2^e, \text{delete}_0^e, \\ &\qquad \text{delete}_1^e, \text{delete}_2^e, \text{delete}_3^e \end{aligned}$$

where • stands for the list constructor. To reduce the verbosity of the notation, in the sequel we abbreviate symbols as $permute_1$ and $delete_1$ to p_1 and d_1.

Let $\mathcal{C}$ denote the set of symbols $FC(P) \cup PC(P) \cup EC(P)$. The symbol $L_{\langle \mathcal{C}; +, .; \geq \rangle}$ denotes the language containing the symbols in the set $\mathcal{C}$ as constants, the infix functor +/2, the infix functor ./2, the relation symbol $\geq$/2, and the set of variables in the first-order language of the program P. Terms in that language are defined in the usual way and are denoted by the set $Term_{\langle \mathcal{C}; +, .; \geq \rangle}$. To simplify the notation, we use shorthands like fx for $f.x$ and $3f_1x + 2f_2y$ for a term as $f_1x + f_1x + f_1x + f_2y + f_2y$. The relational symbols = and > are defined in terms of $\geq$ as usual and considered as additional primitive predicates. We denote the set of all possible atoms in that language by $Atom_{\langle \mathcal{C}; +, .; \geq \rangle}$. We call such atoms *symbolic expressions*. A (*linear*) *symbolic equation* is an expression in $Atom_{\langle \mathcal{C}; +, .; \geq \rangle}$ of the form $\Sigma_{i=1}^{n} a_i x_i = \Sigma_{j=1}^{m} b_j y_j + c$. A (*linear*) *symbolic inequality* is an expression in $Atom_{\langle \mathcal{C}; +, .; \geq \rangle}$ of the form $\Sigma_{i=1}^{n} a_i x_i \geq \Sigma_{j=1}^{m} b_j y_j + c$. We denote the set of all logical formulae over $Atom_{\langle \mathcal{C}; +, .; \geq \rangle}$ by $F_{\langle \mathcal{C}; +, .; \geq \rangle}$. Such formulae are called *symbolic formulae*. By *natural formulae*, we denote formulae in $F_{\langle \{0,1\}; +, .; \geq \rangle}$.

As one can apply substitutions to natural expressions and formulae, one can also apply substitutions in the symbolic context.

5.2 From the Natural to the Symbolic Level

Symbolic terms will now be chosen to abstract concrete terms (and atoms). Such symbolic terms abstract concrete terms on two different levels. On one hand, such terms include information about the instantiation level of the term in the sense that they account for those parts of the concrete terms whose size may still change under instantiation. On the other hand, they make abstraction of the way in which the term or atom actually is measured. The following definitions formalize this idea.

Definition 5.3 (*Symbolic Norm* $\|.\|^s$). Let $FC(P)$ be a set of functor coefficients.

$$\begin{aligned} \|.\|^s : Term_P &\rightarrow Term_{\langle \mathcal{C}; +, .; \geq \rangle} \\ t &\rightarrow f_0 + \sum_{i=1}^{n} f_i \|t_i\|^s \quad \text{if } t = f(t_1, \cdots, t_n),\ n > 0, \\ t &\rightarrow x \quad \text{if } t = x \in Var_P, \\ t &\rightarrow 0 \quad \text{if } t \in Const_P \end{aligned}$$

In the same way, we can define a symbolic level mapping by symbolizing its coefficients.

Definition 5.4 (*Symbolic Level Mapping* $|.|^s$). Let $PC(P)$ be a set of predicate coefficients and $\|.\|^s$ a symbolic norm.

$$\begin{aligned} |.|^s : Atom_P &\rightarrow Term_{\langle \mathscr{C}; +, .; \geq \rangle} \\ p(t_1, \cdots, t_n) &\rightarrow \sum_{i=1}^{n} p_i \|t_i\|^s \end{aligned}$$

Finally, we can abstract success sets through a similar scheme.

Definition 5.5 (*Symbolic Size Expression* $\mathscr{A}^s$). Let $EC(P)$ be a set of extended predicate coefficients, $\|.\|^s$ a symbolic norm, and I_p and J_p the sets of argument positions fixed for predicate p/n.

$$\begin{aligned} \mathscr{A}^s: \quad Atom_P &\rightarrow Atom_{\langle \mathscr{C}; +; \geq \rangle} \\ t_1 = t_2 &\rightarrow \|t_1\|^s = \|t_2\|^s \\ p(t_1, \cdots, t_n) &\rightarrow \sum_{i \in I_p} p_i^e \|t_i\|^s \geq \sum_{j \in J_p} p_j^e \|t_j\|^s + p_0^e, \text{ where } p \neq =/2 \end{aligned}$$

Example 5.6. Consider again the delete program of Example 2.11. On lists, the norm $\|.\|^s$ would measure a term like $[1, x|y]$ as

$$\|[1, x|y]\|^s = \bullet_0 + \bullet_2(\bullet_0 + \bullet_1 x + \bullet_2 y).$$

On an atom like $delete(x, [1|y], z)$, the level mapping $|.|^s$ would yield

$$|delete(x, [1|y], z)|^s = d_1 x + d_2(\bullet_0 + \bullet_2 y) + d_3 z.$$

Assuming again that $I_{delete} = \{2\}$ and $J_{delete} = \{1, 3\}$, the size expression would map an atom $delete(x, [x|y], y)$ to

$$\mathscr{A}^s(delete(x, [x|y], y)) = d_2^e(\bullet_0 + \bullet_1 x + \bullet_2 y) \geq d_1^e x + d_3^e y + d_0^e.$$

5.3 Symbolic Termination Condition

Whenever we have a formula over $F_{\langle \mathscr{C}; +, .; \geq \rangle}$, we can associate a natural formula to it by fixing a symbol mapping.

Definition 5.7 (*Symbol Mapping*). A *symbol mapping* is a mapping $s : \mathscr{C} \rightarrow \mathbb{N}$.

Expressions involving only symbols from $\mathscr{C}$ are mapped into the natural numbers by substituting the symbols by their mapped value. With abuse of notation, if F is a symbolic formula, and s a symbol mapping, we denote the associated natural formula as $s(F)$.

Each symbol mapping s induces in a natural way a norm and a level mapping by mapping the symbols in the generic symbolic definitions to their actual value under s. A symbol mapping s also characterizes an interargument relation $R_s^{p/n}$ for each predicate p/n of P from the symbolic size expression as follows:

$$R_s^{p/n} = \{(s(\|t_1\|^s), \cdots, s(\|t_n\|^s)) | t_1, \cdots, t_n \in Term_P \text{ and } s(\mathcal{A}^s(p(t_1, \cdots, t_n))) \text{ holds}\}.$$

Definition 5.8 (*Rigid Symbol Mapping*). A symbol mapping s is *rigid* with respect to a set of atoms S if and only if the level mapping induced by s is rigid with respect to S.

Following the syntactic characterization of rigidity of terms under semi-linear norms of Bossi et al. [1991], this condition entails the following sufficient condition on s:

PROPOSITION 5.9 (RIGID SYMBOL MAPPING: SUFFICIENT CONDITION). Let P be a program, S a set of atoms, and s a symbol mapping on $\mathcal{C}$. Then, s is rigid with respect to S if,

—for any predicate p/n in P, such that $Call(P, S)$ contains an atom, $p(t_1, \cdots, t_i, \cdots, t_n)$, with t_i a free variable: $s(p_i) = 0$, and
—for any term $t = f(t_1, \cdots, t_i, \cdots, t_n)$ occurring (possibly as a subterm) within some argument of an atom in $Call(P, S)$, with t_i a free variable: $s(f_i) = 0$.

PROOF. Follows directly from Bossi et al. [1991, Prop.2.1.13]. □

Example 5.10. Consider permute queried as in Section 4. For permute, the rigidity conditions obtained through Proposition 5.9 are identical to the ones presented in Section 3, up to syntactic renaming:

$$(C_1) s(d_1) = s(d_3) = s(\bullet_1) = 0 \text{ (for delete atoms)},$$
$$(C_2) s(p_2) = s(\bullet_1) = 0 \text{ (for permute atoms)}.$$

In Section 6.5 we show how the above proposition can be further refined when the set $Call(P, S)$ is abstracted by means of mode or type information.

One of the desirable properties a symbol mapping could have is that the interargument relations induced by it be valid. This is the case for valid symbol mappings.

Definition 5.11 (*Valid Symbol Mapping*). A symbol mapping s is *valid* if all interargument relations induced by s are valid with respect to the norm induced by s.

This relates to a symbolic condition following the $T_P(I) \subseteq I$ condition for characterizing models.

PROPOSITION 5.12 (VALIDITY OF A SYMBOL MAPPING). Let P be a program and s a symbol mapping on $\mathcal{C}$. If for each clause $H \leftarrow B_1, \cdots, B_n \in P$ it holds that

$$s(\forall[\mathcal{A}^s(B_1) \wedge \cdots \wedge \mathcal{A}^s(B_n) \Rightarrow \mathcal{A}^s(H)])$$

then s is valid.

PROOF. The union of the relations $R_s^{p/n}$

$$= \{(s(\|t_1\|^s), \cdots, s(\|t_n\|^s)) | t_1, \cdots, t_n \in Term_P \text{ and } s(\mathcal{A}^s(p(t_1, \cdots, t_n))) \text{ holds}\},$$

$p/n \in P$, defines an (nonground) interpretation of P on the domain $\mathbb{N}$. The condition expresses, for this interpretation, that $T_P(I) \subseteq I$ holds. Thus, the interpretation is a (nonground) model, and therefore each $R_s^{p/n}$ is a valid interargument relation. □

Example 5.13. Reconsider the permute example, where we need a model for the delete predicate. The validity condition for the delete predicate states that

$$(C_3)s(\forall[true \Rightarrow d_2^e(\bullet_1 x + \bullet_2 t + \bullet_0) \geq d_1^e\, x + d_3^e\, t + d_0^e])$$

$$(C_4)s(\forall[d_2^e\, t \geq d_1^e\, x + d_3^e\, t' + d_0^e \Rightarrow$$

$$d_2^e(\bullet_1 h + \bullet_2 t + \bullet_0) \geq d_1^e\, x + d_3^e(\bullet_1 h + \bullet_2 t' + \bullet_0) + d_0^e]).$$

We can now formulate a proposition analogue to Proposition 4.1, but which is formulated in symbolic terms.

PROPOSITION 5.14. Let P be a directly recursive program and S a set of atoms. If there exists a *valid* symbol mapping s which is *rigid* with respect to $Call(P, S)$ such that for each clause $A \leftarrow B_1, \cdots, B_{i-1}, A', B_{i+1}, \cdots, B_m \in P$, and such that for each A' having the same predicate symbol as A,

$$s(\forall[\mathcal{A}^s(B_1) \wedge \cdots \wedge \mathcal{A}^s(B_{i-1}) \Rightarrow |A|^s > |A'|^s]),$$

then P is acceptable with respect to S.

PROOF. Suppose that for some program P there exists a symbol mapping s satisfying the above condition. Obviously, if $s(\forall[\mathcal{A}^s(B_1) \wedge \cdots \wedge \mathcal{A}^s(B_{i-1}) \Rightarrow |A|^s > |A'|^s])$ holds, then the condition also holds for any instantiation of it:

$$s(\forall:[\mathcal{A}^s(B_1\sigma) \wedge \cdots \wedge \mathcal{A}^s(B_{i-1}\sigma) \Rightarrow |A\sigma|^s > |A'\sigma|^s]) \quad (1)$$

Now, let ${}^s|.|$ be the level mapping induced by s, ${}^s\|.\|$ the norm induced by s, and, for each predicate p/n in P, $R_s^{p/n}$ the interargument relation induced by s.

Because s is rigid with respect to $Call(P, S)$, ${}^s|.|$ is also rigid with respect to $Call(P, S)$. Because s is valid, each $R_s^{p/n}$ is valid.

Take any clause $A \leftarrow B_1, \cdots, B_{i-1}, B_i, \cdots, B_n$ of P, with B_i having the same predicate as A. Let θ be any substitution such that the arguments of $B_1\theta, \cdots, B_{i-1}\theta$ satisfy $R_s^{B_1}, \cdots, R_s^{B_{i-1}}$ respectively. By definition of the interargument relation induced by s, this means $s(\forall:[\mathcal{A}^s(B_1\theta) \wedge \cdots \wedge$

$\mathcal{A}^s(B_{i-1}\theta)]$) holds. Thus, by (1), $({}^s|A\sigma| > {}^s|A'\sigma|)$ holds, which reduces to ${}^s|A\theta| > {}^s|A'\theta|$. □

Example 5.15. In the case of permute, the symbolic version of the rigid acceptability imposes the conditions

$$(C_5)s(\forall[true \Rightarrow d_1x + d_2(\bullet_1h + \bullet_2t + \bullet_0) + d_3(\bullet_1h + \bullet_2t' + \bullet_0) > d_1x + d_2t + d_3t'])$$

$$(C_6)s(\forall[d_2^e\, l \geq d_1^e\, el + d_3^e\, l_1 + d_0^e \Rightarrow p_1l + p_2(\bullet_1el + \bullet_2t + \bullet_0) > p_1l_1 + p_2t]).$$

For completeness, we recall the other conditions on the symbol mapping s:

$$(C_1)s(d_1) = s(d_3) = s(\bullet_1) = 0$$

$$(C_2)s(p_2) = s(\bullet_1) = 0$$

$$(C_3)s(\forall[true \Rightarrow d_2^e(\bullet_1x + \bullet_2t + \bullet_0) \geq d_1^e\, x + d_3^e\, t + d_0^e])$$

$$(C_4)s(\forall[d_2^e\, t \geq d_1^e\, x + d_3^e\, t' + d_0^e \Rightarrow d_2^e(\bullet_1h + \bullet_2t + \bullet_0) \geq d_1^e\, x + d_3^e(\bullet_1h + \bullet_2t' + \bullet_0) + d_0^e])$$

At this point, we have transformed the general problem of proving termination into the problem of searching for a symbol mapping validating all of the above formulae. Obviously, given a symbol mapping, checking the validity of C_1 and C_2 is a problem of a different complexity than that of verifying the formulae C_3 to C_6. In the next section we develop a method for reducing these complex formulae into basic formulae involving the introduced symbols only. In addition, the reduction method will be of a constructive nature, in the sense that these basic formulae can guide the process of finding a suitable symbol mapping. The resulting system of formulae will in general be easy to solve.

6. THE QUEST FOR CONSTRAINTS

The problem with most of the symbolic conditions derived in the previous section is that they include two different types of variables: the actual symbolic coefficients for which we aim to fix a symbol mapping and the universally quantified variables, which express that the derived conditions should hold for any values for these. The point is to eliminate the latter variables and obtain new conditions in terms of the symbolic coefficients only. First, we need some theory on linear equations and inequalities. We assume that the basic concepts which are related to linear inequalities are familiar.

6.1 Solved Inequalities and Constraint Sets

In the following, the symbol $\succcurlyeq$ stands for either one of the two relational symbols $\geq$ and $>$.

Definition 6.1 (*Positive Inequality*). A linear (natural) inequality $\Sigma_{i=1}^{n} a_i x_i + c_1 \succcurlyeq \Sigma_{j=1}^{k} b_j y_j + c_2$ is *in positive form* if and only if

$$\forall 1 \leq j \leq k : b_j = 0,\ c_2 = 0, \text{ and } \forall 1 \leq i < j \leq n : x_i \neq x_j.$$

Note that each linear inequality can be written in positive form, thus in the form $\Sigma_{i=1}^{n} a_i x_i + c\ 0$.

Definition 6.2 (*Solved Inequality*). A positive inequality $\Sigma_{i=1}^{n} a_i x_i + c \succcurlyeq 0$ is a *solved inequality* if and only if

$$\forall 1 \leq i \leq n : a_i \geq 0 \text{ and } c \succcurlyeq 0.$$

Proposition 6.3. Let $\Sigma_{i=1}^{n} a_i x_i + c \succcurlyeq 0$ be a solved inequality. Then

$$\forall (N_1, \cdots, N_n) \in \mathbb{N}^n : \sum_{i=1}^{n} a_i N_i + c \succcurlyeq 0.$$

PROOF. Trivial. □

Solved inequalities are the keystone of the technique we propose for generating appropriate symbol mappings. The idea is to rewrite each of the conditions obtained in the previous section into a form closely resembling the solved natural form. We then interpret the solved inequality definition as a constraint on the symbol mapping. By doing so, we no longer need to spend attention to the universally quantified variables, as Proposition 6.3 suggests.

First, we show how to (partially) characterize symbol mappings by means of constraint sets. Based on this characterization, we propose a method to rewrite the conditions of the previous section into a solved form.

Definition 6.4 (*Constraint Set*). A *constraint set* is a set of equations and inequalities over the symbols in $\mathscr{C}$.

Definition 6.5 (*Solution of a Constraint Set*). A symbol mapping s is a *solution of a constraint* set S if and only if $\forall E \in S : s(E)$.

The above characterization of symbol mappings as solutions of a constraint set allows us to reformulate the condition in Proposition 5.14 as the search for a convenient constraint set. The main idea of our method for deriving constraint sets consists in transforming the initial symbolic inequality in such a way that the inequality becomes trivial to solve. Above, we have seen that so-called solved inequalities are always satisfied. Therefore, the transformation process is targeted at rewriting the condition in Proposition 5.14 into a solved inequality. Considering the transformation

itself, we use two main rules for rewriting. Although they look simple at first sight, transforming with them allows us to impose valuable constraints on the symbol mapping. Furthermore, similar methods are necessary to express the validity of symbol mappings in terms of constraint sets.

6.2 Constraint Generation

Reexamining the results obtained in Section 5, we observe that all conditions set up from the validity condition or the symbolic rigid acceptability take the following form:

$$s\left(\forall\left[I_1 \wedge \cdots \wedge I_k \Rightarrow \sum_i a_i x_i \ \geqslant \ \sum_j b_j y_j + b_0 \right] \right)$$

where $\forall$ universally quantifies the formula between the square brackets. Here, each I_m is either an inequality of the form $\Sigma\ c_p x_p \geq \Sigma\ d_q y_q + c_0$, or it is an equality $\|t_1\|^s = \|t_2\|^s$. The relation $\geqslant$ stands for either $\geq$ (in the case of a validity condition) or $>$ (in the case of a symbolic rigid acceptability condition). For validity conditions on facts of P, and for symbolic rigid acceptability conditions for recursive clauses without intermediate body-atoms, the left-hand side of the implication reduces to "true." In the case of permute, C_3, C_4, C_5, and C_6 are instances of this general form.

In order to reduce such conditions to constraints, we first rewrite them into a normal form by grouping all subexpressions in $\geqslant$-inequalities to the left-hand side of the $\geqslant$-sign and merging the coefficients of identical variables, obtaining formulae

$$s\left(\forall\left[I'_1 \wedge \cdots \wedge I'_k \Rightarrow \sum_{i=1}^{n} e_i x_i + c \ \geqslant \ 0 \right] \right)$$

where all I'_m are either of the form $\Sigma_j\ f_j y_j + c \geq 0$ or $\|t_1\|^s = \|t_2\|^s$. From this point on, we will refer to such formulae as *normal formulae*.

Example 6.6. In the permute example, we get the following normalized conditions, after substituting (C_1) and (C_2) to reduce the complexity:

$$(C'_3)s(\forall[true \Rightarrow (d_2^e \bullet_2 - d_3^e)t - d_1^e\, x + (d_2^e \bullet_0 - d_0^e) \geq 0)$$

$$(C'_4)s(\forall[d_2^e\, t - d_1^e\, x - d_3^e\, t' - d_0^e \geq 0 \Rightarrow$$

$$d_2^e \bullet_2 t - d_1^e\, x - d_3^e \bullet_2 t' + ((d_2^e - d_3^e) \bullet_0 - d_0^e) \geq 0])$$

$$(C'_5)s(\forall[true \Rightarrow d_2(\bullet_2 - 1)t + d_2 \bullet_0 > 0])$$

$$(C'_6)s(\forall[d_2^e\, l - d_1^e\, el - d_3^e\, l_1 - d_0^e \geq 0 \Rightarrow p_1 l - p_1 l_1 > 0])$$

Since the universally quantified variables range over all natural numbers, the above conditions are guaranteed to hold whenever all coefficients for these variables (occurring at the right-hand side of the implication) are larger than or equal to zero. In addition, constant summands in strict inequalities (e.g., $d_2 \bullet_0$ in C'_5) need to be strictly positive. We can then use the following proposition to distillate constraints on the symbol mapping involving the symbolic coefficients only.

PROPOSITION 6.7 (DERIVATION OF CONSTRAINTS). *Let* $s(F) \equiv s(\forall[I'_1 \wedge \cdots \wedge I'_k \Rightarrow \Sigma_i\ e_i x_i + c \geqslant 0])$ *be a formula normalized along the lines above. Then*

If s is a solution of the set $\{e_i \geq 0 | 1 \leq i \leq n\} \cup \{c \geqslant 0\}$

then $s(F)$ holds.

PROOF. Immediate, since we can prove the stronger proposition that for such a symbol mapping the formula $s(\forall[true \Rightarrow \Sigma_i\ e_i x_i + c \geqslant 0])$ holds. Obviously, the right-hand-side formula becomes a solved formula for such s, and since the universally quantified variables x_i range over the natural numbers, the claim is proved through Proposition 6.3. □

Example 6.8. Consider the condition C'_3. We get the following explicit constraints on the symbol mapping:

$$\begin{cases} d_2^e \bullet_2 - d_3^e \geq 0 & (c_1) \\ d_2^e \bullet_0 - d_0^e \geq 0 & (c_2) \\ -d_1^e \geq 0 & (c_3) \end{cases}$$

From constraint c_3 it becomes clear already that whenever the norm is restricted to exclude the elements of a list (as imposed by the rigidity conditions), then any further valid interargument relation is of the form

$$\{(x_1, x_2, x_3) | d_2^e\, x_2 \geq d_3^e\, x_3 + d_0^e\}.$$

Applying the proposition on condition C'_5 generates the following constraints:

$$\begin{cases} d_2(\bullet_2 - 1) \geq 0 & (c_4) \\ d_2 \bullet_0 > 0 & (c_5) \end{cases}$$

Let us analyze constraint c_5. It says that the only way to prove delete to be terminating with respect to the previously fixed set of queries is to first make the level mapping measure the second argument, which is no surprise, as this is the only remaining argument after rigidity analysis, and, secondly, to let the norm count the list constructor. Again, this is not a surprise, as this is the only nontrivial functor in the program, and for a reduction to appear somewhere, at least one functor must effectively be weighted.

In general, the constraint set derived above will be a too strong condition on the symbol mapping. This is clear, since applying the rule on the initial conditions does not account for success information (the valid interargument relations occurring in the definition of rigid acceptability), which is explicitized through the left-hand side of the implication. In fact, if there exists a symbol mapping satisfying the constraint sets derived in the above way for all validity and rigid acceptability conditions, one can prove that the program is terminating with respect to any possible selection rule. The following example illustrates the above point.

Example 6.9. Consider the condition C'_6, for which the following set of three constraints is generated:

$$\begin{cases} p_1 \geq 0 \\ -p_1 \geq 0 \\ 0 > 0 \end{cases}$$

The first two constraints can be satisfied only by deriving the zero-mapping to measure permute atoms. Moreover, the last constraint can never be satisfied.

In the next section, it is explained how the success information captured in the left-hand side can be exploited in the right-hand side to weaken the derived constraints.

6.3 Rewrite Rules

The following propositions are defined on symbolic formulae and are of extreme importance. They are called the *substitution rule* and the *evaluation rule*. Although simple at first sight, they will allow us to introduce the necessary success information available in the left-hand side of implications into the right-hand side.

PROPOSITION 6.10 (SUBSTITUTION RULE). Let s be any symbol mapping, $\forall[I_1 \wedge \cdots \wedge I_k \Rightarrow \Sigma_{i=1}^{n} e_i x_i + c \succcurlyeq 0]$ a symbolic formula which is quantified over the variables in $\{x_1, \cdots, x_n\}$, and $I_p \equiv (x_j = F)$, for some $1 \leq p \leq k$, $1 \leq j \leq n$, and F of the form $\Sigma_{l=1}^{n} b_l x_l + d$.

Then we have

$$s(\forall[I_1 \wedge \cdots \wedge I_k \Rightarrow (\textstyle\sum_{i=1}^{n} e_i x_i + c)\{x_j/F\}]) \succcurlyeq 0$$

iff

$$s(\forall[I_1 \wedge \cdots \wedge I_k \Rightarrow \textstyle\sum_{i=1}^{n} e_i x_i + c \succcurlyeq 0]).$$

PROOF. If $e_j \equiv 0$, then the proof is trivial. Otherwise, the following holds:

$$s\left(\forall\left[I_1 \wedge \cdots \wedge I_k \Rightarrow \sum_{i=1}^{j-1} e_i x_i + \sum_{i=j+1}^{n} e_i x_i + e_j\left(\sum_{l=1}^{n} b_l x_l + d\right) + c \succcurlyeq 0\right]\right)$$

Now take any $(N_1, \cdots, N_n) \in \mathbb{N}^n$ such that $s((I_1 \wedge \cdots \wedge I_k)\{x_1/N_1, \cdots, x_n/N_n\})$ holds. Then

$$s\left(\sum_{i=1}^{j-1} e_i N_i + \sum_{i=j+1}^{n} e_i N_i + e_j\left(\sum_{l=1}^{n} b_l N_l + d\right) + c \geqslant 0\right)$$

holds. Because of $s(I_p)$, we know that $s(e_j \Sigma_{l=1}^{n} b_l N_l + d) = s(e_j N_j)$ or that $s(\Sigma_{i=1}^{j-1} e_i N_i + \Sigma_{i=j+1}^{n} e_i N_i + e_j N_j + c \geqslant 0)$, which proves our claim.

The reverse is trivial because whenever $s(\forall[I_1 \wedge \cdots \wedge I_k \Rightarrow \Sigma_{i=1}^{n} e_i x_i + c \geqslant 0])$ holds, it also holds for a weaker condition with $x_j = s(F)$. □

This proposition teaches us that we can safely substitute the information of (certain) equality constraints of the left-hand side into the right-hand side of the implication, without restricting the class of applicable symbol mappings.

Example 6.11. To illustrate the information brought in by the rule, consider the delete fact, written in normalized form, making the unification explicit:

$$delete(u, v, w) \leftarrow v = [u|w]$$

The validity condition, C_3, would become

$$s(\forall : v = \bullet_0 + \bullet_2 w \Rightarrow d_2^e v - d_1^e u - d_3^e w - d_0^e \geq 0).$$

Deriving constraints at this point results in the system

$$\begin{cases} d_2^e \geq 0 \\ -d_1^e \geq 0 \\ -d_3^e \geq 0 \\ -d_0^e \geq 0 \end{cases}$$

and thus characterizes the class $\{(x_1, x_2, x_3) | d_2^e x_2 \geq 0\}$ as the only remaining class of valid interargument relations for delete. Obviously, no successful termination proof for permute can be based on such interargument relation, since the model designates *all* delete atoms. Applying the substitution rule yields

$$s(\forall : v = \bullet_0 + \bullet_2 w \Rightarrow (d_2^e \bullet_2 - d_3^e) w - d_1^e u + (d_2^e \bullet_0 - d_0^e) \geq 0).$$

Generating constraints along the lines above then gives

$$\begin{cases} d_2^e \bullet_2 - d_3^e \geq 0 \\ -d_1^e \geq 0 \\ d_2^e \bullet_0 - d_0^e \geq 0 \end{cases}$$

which are the same as those we obtain from the original delete clause, whose associated condition has a "*true*" left-hand side.

Let us restrict to boolean values for a moment. Only considering the delete fact, we are already able to make the following conclusions. A valid interargument relation for delete can only be one of the following possibilities: either

$$\{(x_1, x_2, x_3) | d_2^e\, x_2 \geq 0\}$$

with respect to any possible norm or

$$\{(x_1, x_2, x_3) | d_2^e\, x_2 \geq d_3^e\, x_3 + c \wedge c \in \{0, 1\}\}$$

with respect to any norm $\|[t_h|t_t]\| = \|t_t\| + c$, with $c \in \{0, 1\}$.

The equality relation is not always of the form $x = F$ but may in general be of the form $\|t_1\|^s = \|t_2\|^s$. In such a case, we first reduce the unification to solved form using Martelli and Montanari [1982]. Then the complex unification can be replaced by the separate components of the solved form.

The second rewrite rule is the evaluation rule and applies to inequalities of the left-hand side.

PROPOSITION 6.12 (EVALUATION RULE). *Let s be any symbol mapping and $\forall[I_1 \wedge \cdots \wedge I_k \Rightarrow \Sigma_{i=1}^n e_i x_i + c \geqslant 0]$ a symbolic formula which is quantified over $\{x_1, \cdots, x_n\}$. If $s(\forall[I_1 \wedge \cdots \wedge I_k \Rightarrow \Sigma_{i=1}^n e_i x_i + c \geqslant p])$, with $p \in \mathbb{N}$, then $s(\forall[I_1 \wedge \cdots \wedge I_k \Rightarrow \Sigma_{i=1}^n e_i x_i + c \geqslant 0])$.*

PROOF. Take any $(N_1, \cdots, N_n) \in \mathbb{N}^n$, and suppose $s((I_1 \wedge \cdots \wedge I_k)\{x_1/N_1, \cdots, x_n/N_n\})$ holds. Then we must prove $s(\Sigma_{i=1}^n e_i N_i + c \geqslant 0)$ holds, which is trivial, as we know $s(\Sigma_{i=1}^n e_i N_i + c \geqslant p)$ holds and p is positive. □

This rule will be applicable to implications involving at least one $J \geq 0$ conjunct on the left-hand side:

$$s(\forall[I'_1 \wedge \cdots \wedge J \geq 0 \wedge \cdots \wedge I'_n \Rightarrow \sum_i e_i x_i \geqslant 0])$$

Using the proposition, we can then safely move to the implication

$$s(\forall[I'_1 \wedge \cdots \wedge J \geq 0 \wedge \cdots \wedge I'_n \Rightarrow \sum_i e_i x_i - J \geqslant 0]).$$

Example 6.13. In the example, we can apply the following reductions of this type:

$$(C''_4)s(\forall[d_2^e\, t - d_1^e\, x - d_3^e\, t' - d_0^e \geq 0 \Rightarrow$$

$$d_2^e(\bullet_2 - 1)t + d_3^e(1 - \bullet_2)t' + (d_2^e - d_3^e)\bullet_0 \geq 0])$$

$$(C''_6)s(\forall[d_2^e\ l - d_1^e\ el - d_3^e\ l_1 - d_0^e \geq 0 \Rightarrow$$

$$(p_1 - d_2^e)l + (d_3^e - p_1)l_1 + d_1^e\ el + d_0^e > 0])$$

Now, all necessary information is present in the right-hand side, and we derive the following constraints on the symbol mapping:

$$(C''_4)\begin{cases} d_2^e(\bullet_2 - 1) \geq 0 & (c_6) \\ d_3^e(1 - \bullet_2) \geq 0 & (c_7) \\ (d_2^e - d_3^e)\bullet_0 \geq 0 & (c_8) \end{cases}$$

$$(C''_6)\begin{cases} p_1 - d_2^e \geq 0 & (c_9) \\ d_3^e - p_1 \geq 0 & (c_{10}) \\ d_1^e \geq 0 & (c_{11}) \\ d_0^e > 0 & (c_{12}) \end{cases}$$

Let us examine the strength (in terms of completeness) of the evaluation rule. At first sight, the rule seems pretty weak: mere subtraction of the given equations. Here, we will argue that it gives a good, practical approximation of a complete reduction rule.

We restrict this discussion to the case of the implications resulting from the rigid acceptability condition. The ones resulting from the validity conditions can be discussed in a very similar way.

Just to illustrate the issues, let us reduce the abstraction and complexity of the general case by assuming that we are dealing with a clause of the form

$$p(x) \leftarrow q(x, y), r(y, z), p(z),$$

for which we aim to study the induced constraints:

$$\forall x, y, z : s(\mathcal{A}^s(q(x, y)) \wedge s(\mathcal{A}^s(r(y, z)) \Rightarrow {}^s\|x\| > {}^s\|z\|$$

By making some arbitrary, but reasonable assumptions on the input and output arguments in such a rule, this might give rise to a general constraint of the form

$$\forall x, y, z : q_1^e \cdot {}^s\|x\| \geq q_2^e \cdot {}^s\|y\| + q_0^e \wedge r_1^e \cdot {}^s\|y\| \geq r_2^e \cdot {}^s\|z\| + r_0^e \Rightarrow {}^s\|x\| > {}^s\|z\|.$$

A first important observation in this context is the following. Although the evaluation rule seems to be used as a mere subtraction of the inequalities in the left-hand sides from the right-hand sides of the implications (with the aim of finding a trivial inequality), it actually looks for *any positive linear combination* of the inequalities in the left-hand sides that yield the right-hand side.

The reasons for this are the following:

—Every inequality in the left-hand side (e.g., $q_1^e \cdot {}^s\|x\| \geq q_2^e \cdot {}^s\|y\| + q_0^e$) is equivalent with any positive multiple of itself (e.g., $n.q_1^e \cdot {}^s\|x\| \geq n \cdot q_2^e \cdot$

$^s\|y\| + n \cdot q_0^e$); and, as the coefficients q_1^e, q_2^e, q_0^e are still undetermined, all such multiples are covered by the inequality.

—In Algorithm 6.14 (below), we will apply the evaluation rule to *all available* left-hand-side inequalities.

Thus, we are actually trying to derive the inequality $^s\|x\| > {}^s\|z\|$ as any positive linear combination out of the given assumptions. Even stronger, it is sufficient that the subtraction of a positive linear combination reduces the inequality $^s\|x\| > {}^s\|z\|$ to a positive inequality (e.g., something of the form 2. $^s\|x\| + {}^s\|y\| + 3 \geq 0$). As the norms are all positive, the subtraction is allowed to leave us with a positive linear combination of norms, which remains positive.

This being said, it is not hard to verify, in the case of a single intermediate body atom, e.g., clauses of the form

$$p(x) \leftarrow q(x, y), p(y),$$

that the evaluation rule is complete. Inequalities $\Sigma_{i=1}^n a_i\|x_i\| + a_0 \geq 0$ can geometrically be interpreted as parts of an n-dimensional space bounded by an $(n-1)$-dimensional hyperspace, of which we are only interested in the intersection with $\|x_1\| \geq 0 \wedge \cdots \wedge \|x_n\| \geq 0$. The implication constraint resulting from the rigid acceptability condition corresponds to the requirement that one such space is a subspace of another. It is not hard to verify that this only holds if their subtraction yields an inequality $\Sigma_{i=1}^n p_i\|x_i\| + p_0 \geq 0$, with $p_i \geq 0$, for all $j: 1, \cdots, n$.

In the case of more than one intermediate body atom, the evaluation rule is not complete. Consider the following example.

Example 6.14 (*Incompleteness of Evaluation Rule*).

$$\begin{aligned}
&p(x) && \leftarrow at_least_double(x, y),\\
&&& \quad sum_at_least_1(y, x), p(y).\\
&at_least_double(x, Nil) && \leftarrow\\
&at_least_double([x_1, x_2|t], [y|t]) && \leftarrow at_least_double(t, s).\\
&sum_at_least_1(Nil, [x|t]) && \leftarrow\\
&sum_at_least_1([y|t], s) && \leftarrow
\end{aligned}$$

The strongest interargument relations that hold for *at_least_double* and *sum_at_least_1* are $x - 2y \geq 0$, and $x + y - 1 \geq 0$, respectively. The conjunction of $x - 2y \geq 0$ and $x + y - 1 \geq 0$ implies $x > y$. However, subtracting positive linear combinations of these interargument relations from $x - y - 1 \geq 0$ gives

$$(1 - p - q)x + (2p - q - 1)y + (q - 1) \geq 0, \quad \text{with } p \geq 0 \text{ and } q \geq 0.$$

This reduces to a trivial inequality if

$$\begin{cases} 1 \geq p + q \\ 2p \geq q + 1 \\ q \geq 1 \end{cases}$$

which is unsolvable over the natural numbers.

To conclude the discussion, the evaluation rule is incomplete in general, but it is very useful in practice. For instance, for the benchmarks of Speirs et al. [1997], collected by Lindenstrauss and Sagiv [1997], it never failed (see Section 6). A complete reduction rule for deciding implications of *concrete* inequalities exists (e.g., see Cousot and Halbwachs [1978] for several key references). It is based on computing convex hulls. However, our inequalities have *symbolic* coefficients, thereby moving the problem to one dealing with quadratic inequalities. As far as we know, no general method exists for this case.

We can now use the following algorithm for reducing complex symbolic conditions as those obtained in the previous section into constraints. For simplicity, we assume that all explicit unification occurring in the program is of the form $x = t$, where x is a variable, thus giving rise to constraints of the $x = expression$, although more complex forms of unification impose no problems.

Algorithm 6.15 (*Deriving Constraint Sets*). Let $\mathcal{S}$ be a set of conditions imposed by the validity condition and by the reduction part of the rigid acceptability Proposition 4.1 for some logic program P. If each condition in $\mathcal{S}$ is in normal form, then the following algorithm *Reduce* returns a constraint set.

Reduce($\mathcal{S}$):
$S_{con} \leftarrow \emptyset$
while $\mathcal{S} \neq \emptyset$ **Do**
—Select any $C \equiv (I_1 \wedge \cdots \wedge I_n \Rightarrow J \geqslant 0) \in \mathcal{S}$
—$\mathcal{S} \leftarrow \mathcal{S} \backslash \{C\}$
—$S_C \leftarrow \{I_j | 1 \leq j \leq n\}$
—*Rewriting Phase*
  while $S_C \neq \emptyset$ **Do**
  —select $I \in S_C$
  —$S_C \leftarrow S_C \backslash \{I\}$
  —*Rewrite C using one of the following rules.*
    if I is an equality
    then
    apply substitution

$$\underbrace{C : \forall[I_1 \wedge \cdots \wedge I_n \Rightarrow J \geqslant 0] \qquad I \equiv (x = E)}$$

$$C : \forall[I_1 \wedge \cdots \wedge I_n \Rightarrow J\{x/E\} \geqslant 0]$$

else if I is an inequality
apply evaluation

$$\underbrace{C : \forall[I_1 \wedge \cdots \wedge I_n \Rightarrow J \ \geqslant \ 0] \qquad I \equiv (P \geq 0)}$$

$$C : \forall[I_1 \wedge \cdots \wedge I_n \Rightarrow J - P \ \geqslant \ 0]$$

end if
Regroup all variables.
end while

—*Constraint derivation phase*
Let $C \equiv \forall[I_1 \wedge \cdots \wedge I_n \Rightarrow \Sigma_{i=1}^{n} e_i x_i + c \geqslant 0]$ be the result of the rewriting phase.
$S_{con} \leftarrow S_{con} \cup \{e_i \geq 0 | 1 \leq i \leq n\} \cup \{c \geqslant 0\}$

end while
return S_{con}

PROPOSITION 6.16. *Let $\mathcal{S}$ be any set of normal symbolic formulae. If a symbol mapping s is a solution of Reduce($\mathcal{S}$) then $\forall C \in \mathcal{S} : s(C)$.*

PROOF. First of all, notice that each rewrite step transforms a normal formula into another normal formula.

Let C be any formula of $\mathcal{S}$. After each basic step of the rewriting phase, we know at least that $s(C_{after}) \Rightarrow s(C_{before})$: if we had rewritten using substitution, then $s(C_{after}) \Leftrightarrow s(C_{before})$ because of the substitution rule. If we had rewritten using the evaluation rule, then $s(C_{after}) \Rightarrow s(C_{before})$ because of the evaluation rule.

Thus, after rewriting we can conclude that $s(C_{final}) \Rightarrow s(C_{original})$. Since the constraint set is derived from C_{final} and s is a solution of that constraint set, it follows that $s(C_{original})$ holds. □

6.4 Solving the Constraints

Solving systems such as $\{c_1, \cdots, c_{12}\}$ is rather simple by employing for example, a finite domain solver or a boolean solver. In practice, in most of the cases restricting to boolean values for the variables is sufficient. Then, standard enumeration techniques can be applied. In the case of permute, mapping to boolean values gives a correct solution. To clarify the discussion, we repeat all constraints derived through the previous sections:

$$\begin{cases} d_2^e \bullet_2 - d_3^e \geq 0 & (c_1) \\ d_2^e \bullet_0 - d_0^e \geq 0 & (c_2) \\ -d_1^e \geq 0 & (c_3) \\ d_2(\bullet_2 - 1) \geq 0 & (c_4) \\ d_2 \bullet_0 > 0 & (c_5) \\ d_2^e(\bullet_2 - 1) \geq 0 & (c_6) \\ d_3^e(1 - \bullet_2) \geq 0 & (c_7) \\ (d_2^e - d_3^e) \bullet_0 \geq 0 & (c_8) \\ p_1 - d_2^e \geq 0 & (c_9) \\ d_3^e - p_1 \geq 0 & (c_{10}) \\ d_1^e \geq 0 & (c_{11}) \\ d_0^e > 0 & (c_{12}) \end{cases}$$

The following conclusions can be drawn immediately from individual constraints: from $c_3 : d_1^e = 0$, from $c_5 : d_2 = 1$, $\bullet_0 = 1$, and from $c_{12} : d_0^e = 1$. Notice also that c_{11} is redundant. Next, using this information, c_4 imposes $\bullet_2 = 1$. At this point, c_6 and c_7 become redundant. c_2 then imposes $d_2^e = 1$, making c_1 and c_8 redundant. From c_9, we learn that we must take $p_1 = 1$, and c_{10} then requires $d_3^e = 1$. At this point, all inequalities are satisfied by this mapping. If any inequalities would remain, an enumeration on the remaining variables would have provided a solution. In our case, any remaining variable can be freely mapped onto any value, but as there are not any of them left, this is the only solution of the system.

The resulting norm, level mapping, and interargument relation are

$$\|[t_1|t_2]\| = 1 + \|t_2\|$$

$$|permute(t_1, t_2)| = \|t_1\|$$

$$|delete(t_1, t_2, t_3)| = \|t_2\|$$

$$R^{delete} = \{(x_1, x_2, x_3) | x_2 \geq x_3 + 1\}.$$

Sometimes, restricting to boolean values for all variables is not sufficient, and other solution methods must be adopted. In such a case, the system can be solved by restricting only part of the symbols to boolean values, e.g., like restricting the values of the functors and level mappings to booleans. Then, our system of constraints becomes linear and can be solved through a simplex algorithm.

6.5 Rigidity Constraints

In the previous sections, we have described how to derive symbol mappings that are valid and are suited for a termination proof as they satisfy the condition imposed on each clause. Proposition 5.14 expresses one more

condition on such symbol mappings: they must be rigid with respect to the call set $Call(P, S)$ for some initial set of queries S. For automation purposes, it is convenient to express the initial set S as a pattern of symbolic information. In such a way, infinite sets can be expressed. Several formalisms have been proposed in the literature. One of the simplest is modes. Modes distinguish between ground terms, uninstantiated terms, and any possible term. As this is not part of the main focus of this article, we only present the basics on modes. More detailed information can be found in Mellish [1985] or Debray and Warren [1986], among others.

Definition 6.17 (*Mode of a Predicate*). Let P be a program and p/n a predicate defined in P. A *mode* for p/n is a mapping $m_p : \{1, \cdots, n\} \rightarrow \{g, f, a\}$, where g, f, a are three constant symbols in a language different from $\mathcal{L}_P$.

Frequently, the pattern $p(m_p(1), \cdots, m_p(n))$ is used to denote a mode.

For termination analysis, modes are often used to specify the initial set of atoms S. Frequently, this set is specified in terms of one top call mode m_p: where S is defined as the set of atoms with mode m_p. For the sake of termination, one must be able to derive the set of all possible calls $Call(P, S)$ from S. We refer to Bruynooghe [1991] and Debray [1989], where mode-inferencing algorithms are proposed.

Finally, in order to ensure the rigidity property required in Proposition 5.14, it is straightforward to verify that the following constraints are sufficient:

$$\cup_{p/n \in Pred_P} \{p_i = 0 | i \in \{1, \cdots, n\} \wedge m_p(i) \in \{f, a\}\}$$

Example 6.18. Consider the mode $permute(g, f)$ denoting the set of all permute atoms having a ground first argument and a variable on the second argument position. If we apply a mode-inferencing algorithm, we obtain the following modes:

$$permute(g, f)$$

$$delete(f, g, f)$$

The following constraints are derived:

$$\{p_2 = 0, d_1 = 0, d_3 = 0\}$$

A possible symbol mapping s satisfying the above constraint set defines the following level mapping:

$$permute(t_1, t_2) = {}^s\|t_1\|$$

$$delete(t_1, t_2, t_3) = {}^s\|t_2\|$$

For any possible norm ${}^s\|.\|$, this level mapping is rigid with respect to $Call(P, S)$, where S is the set of atoms denoted by $permute(g, f)$.

Next, let us consider the case in which the call set is represented in the form of types. Types extend the limited precision of modes by taking into account functor information in addition to instantiation information. They allow, for example, to distinguish between lists and other terms, which is impossible by modes.

In the literature, several type formalisms have been proposed (e.g., see Pfenning [1992]). In this article, we pick out one particular instance, called *rigid types*, to illustrate the main ideas. We refer to Janssens and Bruynooghe [1992] for a deeper study on rigid types and their properties. Other formalisms can be used with minor changes only. In the following, we recall the basic ideas, and we 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.

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 6.19 (*Rigid Type Graph*, *Adapted from Janssens and Bruynooghe* [*1992*]). A *rigid* type graph T is a five-tuple, (*Nodes*, *ForArcs*, *BackArcs*, *Label*, *ArgPos*), where

(1) *Nodes* is a finite, nonempty 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 $Nodes \rightarrow \mathcal{P} \cup Const_P \cup Fun_P \cup \{\mathtt{Max}, \mathtt{OR}\}$, and
(5) *ArgPos* is a function $\cup_{k>0}(\{m \in Nodes | Label(m) = f/k \in Fun_P\} \times \{1, \ldots, k\}) \rightarrow Nodes \backslash \{root\}$, such that for each $m \in Nodes$, with $Label(m) = f/k$, $ArgPos(m, \cdot): \{1, \ldots, k\} \rightarrow Nodes$ is a bijection from $\{1, \ldots, k\}$ onto $\{n \in Nodes | (m, n) \in ForArcs \cup BackArcs\}$.

Each node labeled with a function symbol with arity k has k immediate descendants; each node labeled `OR` has at least two immediate descendants; and the other nodes have no descendants and are called *terminal nodes*. For a functor node n, we use the shorthand n/i to denote the $ArgPos(n, i)$. Descendants of `OR`-nodes or functor-nodes are found using the *Desc* function.

Definition 6.20 (*Desc Function*). $Desc: \{n \in Nodes | Label(n) \in Fun_P \cup \{\mathtt{OR}\}\} \rightarrow 2^{Nodes}: Desc(n) = \{n' | (n, n') \in ForArcs \cup BackArcs\}$.

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, $\mathbb{D}$. The

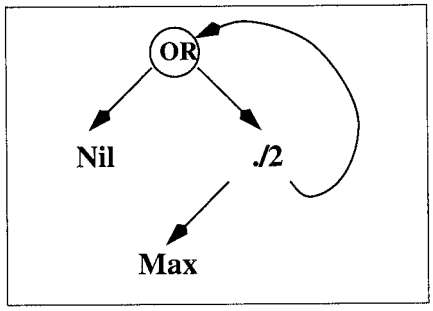

Fig. 1. T_{list}, type graph representing lists of terms.

next couple of definitions were inspired by similar definitions in Mulkers [1993].

Definition 6.21 (*Adapted from Mulkers [1993, Definition 2.3.2]*. Let $\mathbb{T}$ be the function $\mathbb{T}:Nodes \times 2^{(Nodes \times Term_P)} \rightarrow 2^{Term_P}$:

$$
\begin{array}{ll}
\text{if } Label(n) \in Const_P & \text{then } \mathbb{T}(n, I) = \{Label(n)\} \\
\text{if } Label(n) \in \mathcal{P} & \text{then } \mathbb{T}(n, I) = Denote(Label(n)) \\
\text{if } Label(n) = \texttt{Max} & \text{then } \mathbb{T}(n, I) = Term_P \\
\text{if } Label(n) = \texttt{OR} & \text{then } \mathbb{T}(n, I) = \cup_{n' \in Desc(n)} \{t | (n', t) \in I\} \\
\text{if } Label(n) = f/k & \text{then } \mathbb{T}(n, I) = \{f(t_1, \ldots, t_k) | ArgPos(n, i) \\
 & \quad = n_i, (n_i, t_i) \in I\}.
\end{array}
$$

The set $2^{(Nodes \times Term_P)}$ forms a complete lattice with respect to $\subseteq$, $\cap$, and $\cup$. The bottom element is $\emptyset$, while $Nodes \times Term_P$ is the top element.

Definition 6.22 (*Adapted from Mulkers [1993]*). Let $\mathbb{T}_{Nodes}$ be the function $\mathbb{T}_{Nodes}:2^{(Nodes \times Term_P)} \rightarrow 2^{(Nodes \times Term_P)}$: $\mathbb{T}_{Nodes}(I) = I \cup \{(n, t) | n \in Nodes, t \in \mathbb{T}(n, I)\}$. Observe that $\mathbb{T}_{Nodes}$ is continuous.

Definition 6.23 (*Denotation of a Node in a Type Graph*). Let $T = (Nodes, ForArcs, BackArcs, Label, ArgPos)$ be a rigid type graph. The denotation of $n \in Nodes$ is defined as $\mathbb{D}(n) = \{t | (n, t) \in \mathbb{T}_{Nodes} \uparrow \omega\}$.

Definition 6.24 (*Denotation of a Rigid Type*). Let T be a rigid type with root n_{root}. Then $\mathbb{D}(T) = \mathbb{D}(n_{root})$.

Sometimes we use $\mathbb{D}(p(T_1, \cdots, T_n))$ as an abbreviation for the set $\{p(t_1, \cdots, t_n) | \forall 1 \leq i \leq n : t_i \in \mathbb{D}(T_i)\}$.

Example 6.25. Two examples of rigid types are presented in Figures 1 and 2. The denotation of T_{list} comprises the set of all nil-terminated lists of any terms. The type T_{lla} of Figure 2 comprises all nil-terminated lists of nil-terminated lists of any terms. The first type could be used, for example, in the permute program to specify the set of queries of interest: *permute*(T_{list}, *Max*). The pattern abstracts the set of all permute atoms having a list of any terms on the first argument position and any term on the second

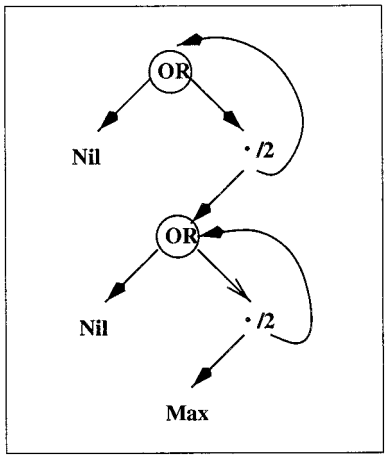

Fig. 2. Lists of lists of finite length.

one. The complete call set $Call(permute, \mathbb{D}(permute(T_{list}, Max)))$ is then safely approximated by the denotation of the patterns $permute(T_{list}, Max)$ and $delete(Max, T_{list}, Max)$.

In the remainder of this subsection, our interest is in deriving constraints when the call set is provided under the form of such type information. Decorte et al. [1993] proposed a method to automatically infer a norm from such type graphs. The main achievement there, which has practical value for the constraint generation that will come next, is that the derived norm $\|.\|_T$ is rigid on the set of terms $\mathbb{D}(T)$. We slightly reformulate the main result of the paper to the symbol-mapping context.

Definition 6.26 (*Critical Path*). Let $T = (Nodes, ForArcs, BackArcs, Label, ArgPos)$ be a rigid type. A *critical path* in T is a path from the root node to a node labeled *Max*.

PROPOSITION 6.27 (ADAPTED FROM DECORTE ET AL. [1993, PROP. 3.4]. *Let* $T = (Nodes, ForArcs, BackArcs, Label, ArgPos)$ *be a rigid type whose functors occur in* Fun_P, *and let s be a symbol mapping. If on each critical path P of T there exists an arc* (n_1, n_2), *with* $Label(n_1) = f/k$ *and* $ArgPos(n_1, i) = n_2$ *such that* $s(f_i) = 0$, *then* ${}^s\|.\|$ *is rigid with respect to* $\mathbb{D}(T)$.

Informally, the proposition states that on each path to a Max node, there should occur (at least) one functor node for which the norm does *not* measure arguments according to the next node on the path, i.e., the norm never manages to measure terms corresponding to Max arguments.

This proposition forms the basis for the following algorithm. It assumes the call set $Call(P, S)$ abstracted by one separate call-type pattern per predicate:

$$Call(P, S) = \cup_{p/n \in Pred_P} \mathbb{D}(p(T_1^p, \cdots, T_n^p)).$$

The algorithm $RigCon(S_{pat})$ takes any set of rigid type patterns and returns a set of rigidity constraints. In the algorithm, we denote the set of all critical paths in a type graph T by $Critic(T)$.

Algorithm 6.28 (*Generation of Rigidity Constraints from Types*).

$RigCon(S_{pat})$:

Initialization
$S_{con} \leftarrow \emptyset$

Constraint generation
while $S_{pat} \neq \emptyset$
do

—select a type pattern $p(T_1, \cdots, T_m) \in S_{pat}$
—$S_{pat} \leftarrow S_{pat} \backslash \{p(T_1, \cdots, T_m)\}$
—$S_{con} \leftarrow S_{con} \cup_{k=1}^{m} \{p_k = 0 | T_k = Max\}$
—$S_{con} \leftarrow S_{con} \cup_{k=1}^{n} \{\cup_{Path \in Critic(T_k)} \{p_k \prod_{f \in FA(Path)} f = 0\}\}$ where $FA(Path)$ stands for the set $\{f_i |$there is an arc (n_1, n_2) in $Path$, with $Label(n_1) = f/k$ and $ArgPos(n_1, i) = n_2\}$

end do
return S_{con}

Example 6.29 (*Delete*). Consider the following calls to delete: *delete*-(Max, T_{lla}, Max), where T_{lla} is the type of Figure 2. For the two types with label Max, we get $d_1 = d_3 = 0$. The type graph for the second argument contains one critical path $CP = (n_1, n_3, n_4, n_6, n_7)$, where we have numbered all nodes in a breadth-first, left-right way. Then, $FA(CP) = \{\bullet_1\}$ (both the nodes n_3 and n_6 are responsible for introducing $\bullet_1$ in $FA(CP)$). As a result, a constraint $d_2 \bullet_1 = 0$ is obtained for the second argument position.

PROPOSITION 6.30. *Let S_{pat} be any set of type patterns for predicates in a program P, and let s be a symbol mapping on P. If s is a solution of $RigCon(S_{pat})$, then s is rigid on $\mathbb{D}(S_{pat})$.*

PROOF. Take any $p(T_1, \cdots, T_m) \in S_{pat}$. Then ${}^s|.|$ has to be rigid on $\mathbb{D}(p(T_1, \cdots, T_m))$. Now, $\forall 1 \leq i \leq m$: if $p_i \neq 0$, then we must prove that ${}^s\|.\|$ is rigid with respect to $\mathbb{D}(T_i)$.

Consider now T_i. For any of its nodes n with $Label(n) = Max$ and for any critical path $Path$ from n_{root} to n, we know that $RigCon(S_{pat})$ contains a constraint $p_i f_1 \cdots f_k$, where each f_i corresponds directly to one arc $(n_1, n_2) \in Path$ with $Label(n_1) = f/k$ and $ArgPos(n_1, i) = n_2$.

Obviously, as $p_i \neq 0$, there is an i such that $s(f_i) = 0$ (s is a solution of $RigCon(S_{pat})$), or, in other words, on each forward path toward any Max node, we can find an arc (n_1, n_2) with $ArgPos(n_1, i) = n_2$ and $Label(n_1) = f/k$ such that $s(f_i) = 0$. Then we can use Proposition 6.27 to infer that ${}^s\|.\|$ is rigid on $\mathbb{D}(T_i)$. □

Example 6.31 (*Permute*). Let us reconsider the permute example. We are interested in the termination behavior of calls $permute(T_{list}, Max)$, where T_{list} denotes the type of lists of any terms. For ease of explanation, let us name all nodes in that type graph in a breadth-first, left-to-right fashion n_1 till n_4. The call set, specified by the patterns $permute(T_{list}, Max)$ and $delete(Max, T_{list}, Max)$, can then be fed into the algorithm.

For delete, we derive the following constraints, one for each argument position. The first and third argument positions may contain terms denoted by the *Max* type. Thus, the following constraints are immediate: $d_1 = 0$ and $d_3 = 0$. In the call type T_{list} for the second argument position, there is the following critical path from the root (n_1) to the *Max* node (n_4): $\{(n_1, n_3), (n_3, n_4)\}$. There is only one functor node (corresponding to the list constructor) on this path. The constraint generated is $d_2 \bullet_1 = 0$.

A similar analysis for permute results in constraints $p_1 \bullet_1 = 0$ and $p_2 = 0$.

7. EXAMPLES

7.1 Quicksort

$$
\begin{array}{lll}
(q_1) & \text{quicksort}(Nil, Nil) & \leftarrow \\
(q_2) & \text{quicksort}([h|t], s) & \leftarrow \text{partition}(h, t, t_1, t_2), \\
 & & \quad \text{quicksort}(t_1, s_1), \\
 & & \quad \text{quicksort}(t_2, s_2), \\
 & & \quad \text{append}(s_1, [h|s_2], s). \\
(p_1) & \text{partition}(el, Nil, Nil, Nil) & \leftarrow \\
(p_2) & \text{partition}(el, [v|t_1], [v|t_2], l) & \leftarrow el \leq v, \\
 & & \quad \text{partition}(el, t_1, t_2, l). \\
(p_3) & \text{partition}(el, [v|t_1], l, [v|t_2]) & \leftarrow el > v, \\
 & & \quad \text{partition}(el, t_1, l, t_2). \\
(a_1) & \text{append}(Nil, l, l) & \leftarrow \\
(a_2) & \text{append}([h|t_1], l, [h|t_2]) & \leftarrow \text{append}(t_1, l, t_2).
\end{array}
$$

Let us investigate whether quicksort is terminating whenever a query $quicksort(t_1, x)$, with t_1 a ground term and x a free variable, is made. Using mode information, we can represent this kind of query by $quicksort(g, f)$. A safe approximation of the call set is then given by the call modes $quicksort(g, f)$, $partition(g, g, f, f)$, and $append(g, g, f)$.

We can start the analysis by performing a rigidity analysis. We obtain the set of constraints

$$\{q_2 = 0, p_3 = 0, p_4 = 0, a_3 = 0\},$$

where symbols as $quicksort_i$ are abbreviated by q_i.

In a second step, we address the rigid acceptability condition. Let us for a moment assume that *partition* is known to be terminating with respect to *partition*(g, g, f, f), and assume that *append* terminates with respect to *append*(g, g, f) (which is not difficult to verify), so that we can concentrate on the termination of quicksort itself.

Given the above call set, the set $I_{partition}$ is fixed as {1, 2} and $J_{partition}$ as {3, 4}.

We can then apply the rigid acceptability condition with respect to the first recursive call to quicksort in (q_1), and after normalizing, we obtain

$$(Q_1): s(\forall[p_1^e\, h + p_2^e\, t - p_3^e\, t_1 - p_4^e\, t_2 - p_0^e \geq 0 \Rightarrow \\ q_1 \bullet_0 + q_1 \bullet_1 h + q_1 \bullet_2 t + q_2 s - q_1 t_1 - q_2 s_1 > 0]).$$

For the second recursive call, we obtain the following condition:

$$s(\forall[(p_1^e\, h + p_2^e\, t - p_3^e\, t_1 - p_4^e\, t_2 - p_0^e \geq 0) \wedge (q_1^e\, t_1 - q_2^e\, s_1 - q_0^e \geq 0) \Rightarrow \\ q_1 \bullet_0 + q_1 \bullet_1 h + q_1 \bullet_2 t + q_2 s - q_1 t_2 - q_2 s_2 > 0]).$$

To simplify the discussion, we remove the second condition on the left-hand side. Keeping this condition would additionally involve deriving an interargument relation for quicksort and one for append. The reader can verify that this will have no influence on the eventual solution. Our condition then becomes

$$(Q_2): s(\forall[(p_1^e\, h + p_2^e\, t - p_3^e\, t_1 - p_4^e\, t_2 - p_0^e \geq 0) \Rightarrow \\ q_1 \bullet_0 + q_1 \bullet_1 h + q_1 \bullet_2 t + q_2 s - q_1 t_2 - q_2 s_2 > 0])$$

Let us now consider that the validity constraints hold for the partition predicate. For each of the three clauses (p_1), (p_2), and (p_3), a corresponding validity condition must hold. After normalizing, they are

$$(P_1): \quad s(\forall[true \Rightarrow p_1^e\, el - p_0^e \geq 0])$$

$$(P_2): \quad s(\forall[p_1^e\, el + p_2^e\, t_1 - p_3^e\, t_2 - p_4^e\, l - p_0^e \geq 0 \Rightarrow \\ p_1^e\, el + p_2^e(\bullet_0 + \bullet_1 v + \bullet_2 t_1) - p_3^e(\bullet_0 + \bullet_1 v + \bullet_2 t_2) - p_4^e\, l - p_0^e \geq 0])$$

$$(P_3): \quad s(\forall[p_1^e\, el + p_2^e\, t_1 - p_3^e\, l - p_4^e\, t_2 - p_0^e \geq 0 \Rightarrow \\ p_1^e\, el + p_2^e(\bullet_0 + \bullet_1 v + \bullet_2 t_1) - p_3^e\, l - p_4^e(\bullet_0 + \bullet_1 v + \bullet_2 t_2) - p_0^e \geq 0]).$$

Any symbol mapping s satisfying these five conditions (Q_1), (Q_2), (P_1), (P_2), and (P_3) and which is rigid with respect to the proposed call set guarantees the termination of quicksort for that call set. To ease the verification of the existence of such s, we rewrite these conditions into a system of constraints.

As stated before, to enforce rigidity on the call set, we have the following system of constraints:

$$(C_0)\begin{cases} q_2 = 0 \\ p_3 = 0 \\ p_4 = 0 \\ a_3 = 0 \end{cases}$$

The next step is to rewrite the validity and rigid acceptability conditions. This is done using algorithm 6.15. Consider (Q_1), which after performing one evaluation step reduces the right-hand side to the following more suitable condition:

$$(q_1 \bullet_1 - p_1^e)h + (q_1 \bullet_2 - p_2^e)t + (p_3^e - q_1)t_1 + q_2 s + p_4^e\, t_2 - q_2 s_1 + q_1 \bullet_0 + p_0^e > 0$$

This condition translates into the following system of constraints:

$$(C_1)\begin{cases} q_1 \bullet_1 - p_1^e \geq 0 \\ q_1 \bullet_2 - p_2^e \geq 0 \\ p_3^e - q_1 \geq 0 \\ q_2 \geq 0 \\ p_4^e \geq 0 \\ -q_2 \geq 0 \\ q_1 \bullet_0 + p_0^e > 0 \end{cases}$$

Remark the expressivity of the last constraint. It says, that, for a valid termination proof, a reduction can only come from two sources: either data are consumed through the intermediate call to partition, or the reduction must happen through the first argument of quicksort in combination with a norm counting (at least) list constructors.

For (Q_2), we can rewrite the right-hand side of the implication in a similar way, leading to

$$(q_1^e \bullet_1 - p_1^e)h + (q_1 \bullet_2 - p_2^e)t + (p_4^e - q_1)t_2 + q_2 s + p_3^e\, t_1 - q_2 s_2 + q_1 \bullet_0 + p_0^e > 0$$

from which condition we obtain the following constraints:

$$(C_2)\begin{cases} q_1^e \bullet_1 - p_1^e \geq 0 \\ q_1 \bullet_2 - p_2^e \geq 0 \\ p_4^e - q_1 \geq 0 \\ q_2 \geq 0 \\ p_3^e \geq 0 \\ -q_2 \geq 0 \\ q_1 \bullet_0 + p_0^e > 0 \end{cases}$$

From validity condition (P_1), we immediately generate the following constraints:

$$(C_3)\begin{cases} p_1^e \geq 0 \\ -p_0^e \geq 0 \end{cases}$$

After we apply one application of the evaluation rule, the right-hand side of condition (P_2) becomes

$$(p_2^e - p_3^e)\bullet_1 v + (\bullet_2 - 1)p_2^e t_1 + (1 - \bullet_2)p_3^e t_2 + (p_2^e - p_3^e)\bullet_0 \geq 0.$$

The following system of constraints is derived:

$$(C_4)\begin{cases} (p_2^e - p_3^e)\bullet_1 \geq 0 \\ (\bullet_2 - 1)p_2^e \geq 0 \\ (1 - \bullet_2)p_3^e \geq 0 \\ (p_2^e - p_3^e)\bullet_0 \geq 0 \end{cases}$$

Finally, one evaluation step on (P_3) results in

$$(p_2^e - p_4^e)\bullet_1 v + (\bullet_2 - 1)p_2^e t_1 + (1 - \bullet_2)p_4^e t_2 + (p_2^e - p_4^e)\bullet_0 \geq 0$$

translating into the following four constraints:

$$(C_5)\begin{cases} (p_2^e - p_4^e)\bullet_1 \geq 0 \\ (\bullet_2 - 1)p_2^e \geq 0 \\ (1 - \bullet_2)p_4^e \geq 0 \\ (p_2^e - p_4^e)\bullet_0 \geq 0 \end{cases}$$

After removing all duplicate constraints and deleting redundant constraints like $q_2 \geq 0$, we obtain the following system:

$$\begin{cases} q_1\bullet_1 - p_1^e \geq 0 & (c_1) \\ q_1\bullet_2 - p_2^e \geq 0 & (c_2) \\ p_3^e - q_1 \geq 0 & (c_3) \\ q_1\bullet_0 + p_0^e > 0 & (c_4) \\ -q_2 \geq 0 & (c_5) \\ p_4^e - q_1 \geq 0 & (c_6) \\ -p_0^e \geq 0 & (c_7) \\ (p_2^e - p_3^e)\bullet_1 \geq 0 & (c_8) \\ (1 - \bullet_2)p_3^e \geq 0 & (c_9) \\ (p_2^e - p_3^e)\bullet_0 \geq 0 & (c_{10}) \\ (p_2^e - p_4^e)\bullet_1 \geq 0 & (c_{11}) \\ (\bullet_2 - 1)p_2^e \geq 0 & (c_{12}) \\ (1 - \bullet_2)p_4^e \geq 0 & (c_{13}) \\ (p_2^e - p_4^e)\bullet_0 \geq 0 & (c_{14}) \end{cases}$$

c_5 forces us to take $s(q_2) = 0$, and because of c_7 we fix $s(p_0^e) = 0$. If we restrict to boolean values, then c_4 imposes that $s(q_1) = 1$ and $s(\bullet_0) = 1$. Then, from c_6, $(p_4^e) = 1$, and from c_3, we have $s(p_3^e) = 1$. Next, c_{10} and c_{14} are immediate, and they result in $s(p_2^e) = 1$. From c_2 we derive that $s(\bullet_2) = 1$. Finally, c_8, c_{11}, c_9, c_{12}, and c_{13} are satisfied by this assignment. Only constraint c_1 remains: $\bullet_1 - p_1^e \geq 0$. Our conclusion is, that, if c_1 is solvable, then quicksort is terminating with respect to our proposed set of queries. Moreover, the level mapping has the following general form:

$$|quicksort(t_1, t_2)| = \|t_1\|$$

$$\|[t_1|t_2]\| = 1 + s(\bullet_1)\|t_1\| + \|t_2\|$$

$$R^{partition} = \{(x_1, x_2, x_3, x_4) | s(p_1^e)x_1 + x_2 \geq x_3 + x_4\}$$

Constraint c_1 has three different solutions, each of them corresponding to a different termination proof:

—$s(\bullet_1) = 1$: The norm is the term-size norm, and the interargument relation for partition can be either $\{(x_1, x_2, x_3, x_4) | x_1 + x_2 \geq x_3 + x_4\}$ or $\{(x_1, x_2, x_3, x_4) | x_2 \geq x_3 + x_4\}$.

—$s(\bullet_1) = 0$: Termination can be proved using the list-length norm and the interargument relation $\{(x_1, x_2, x_3, x_4) | x_2 \geq x_3 + x_4\}$.

7.2 Reverse with Accumulating Parameter

Reconsider the reverse program using an accumulating parameter:

$$\begin{array}{lcl} reverse(l, l_r) & \leftarrow & revacc(l, l_r, Nil) \\ revacc(Nil, l, l) & \leftarrow & \\ revacc([el|t], r, a) & \leftarrow & revacc(t, r, [el|a]) \end{array}$$

Let S be all queries to reverse with a nil-terminated list in their first argument: $reverse(T_{list}, Max)$. We can then abstract the corresponding call set by type patterns $reverse(T_{list}, Max)$ and $revacc(T_{list}, Max, T_{list})$.

Applying the rigidity conditions of Algorithm 6.28 on these patterns produces the following constraints:

$$\begin{cases} r_2 = 0 & (c_1) \\ ra_2 = 0 & (c_2) \\ r_1 \bullet_1 = 0 & (c_3) \\ ra_1 \bullet_1 = 0 & (c_4) \\ ra_3 \bullet_1 = 0 & (c_5) \end{cases}$$

where we abbreviate *reverse* by r and *revacc* by ra.

If we set up the rigid acceptability condition, we obtain the following (normalized) formula to be satisfied:

$$s(\forall[true \Rightarrow (ra_1 - ra_3)\bullet_1 el + (\bullet_2 - 1)ra_1 t + (1 - \bullet_2)ra_3 a + (ra_1 - ra_3)\bullet_0 > 0])$$

The following additional constraints are generated:

$$\begin{cases} (ra_1 - ra_3)\bullet_1 \geq 0 & (c_6) \\ (\bullet_2 - 1)ra_1 \geq 0 & (c_7) \\ (1 - \bullet_2)ra_3 \geq 0 & (c_8) \\ (ra_1 - ra_3)\bullet_0 > 0 & (c_9) \end{cases}$$

Let us again restrict to boolean values for all symbols. From constraint c_9, we are forced to take $s(\bullet_0) = 1$, and the remainder of the constraint $ra_1 - ra_3 > 0$ enforces that $s(ra_1) = 1$ and $s(ra_3) = 0$. Now, by c_3, we get $\bullet_1 = 0$. Then, from constraint c_7, we must fix $s(\bullet_2) = 1$. With this mapping, all constraints are satisfied.

Concluding, we say that termination can be proved using the list-length norm, and the level mapping is fixed to

$$|revacc(t_1, t_2, t_3)| = \|t_1\|_l.$$

Note if we had started the analysis from a mode pattern $reverse(g, a)$ instead, then the constraints c_3, c_4, and c_5 would not have been generated, and no value for s would have been fixed on $\bullet_1$. In that case, both the term-size and the list-length norms allow us to prove termination.

7.3 Reverse with Encapsulated Accumulating Parameter

$$\begin{aligned} reverse(l, l_r) &\leftarrow revacc([Nil|l], l_r) \\ revacc([l], l) &\leftarrow \\ revacc([a, el|t], r) &\leftarrow revacc([[el|a]|t], r) \end{aligned}$$

Let the set of queries to reverse be the same as in the previous example. Then the call set can be abstracted through $reverse(T_{list}, Max)$ and $revacc(T_{list}, Max)$.

From a rigidity analysis, Algorithm 6.28 gives the following constraints:

$$\begin{cases} r_2 = 0 & (c_1) \\ ra_2 = 0 & (c_2) \\ r_1\bullet_1 = 0 & (c_3) \\ ra_1\bullet_1 = 0 & (c_4) \end{cases}$$

The rigid acceptability condition on *revacc* now is the following (after normalizing and cleaning up):

$$(1 - \bullet_2)ra_1\bullet_1 a + (\bullet_2 - \bullet_1)ra_1\bullet_1 el + (\bullet_2 - 1)ra_1\bullet_2 t + (\bullet_2 - \bullet_1)ra_1\bullet_0 > 0$$

The following constraints are derived:

$$\begin{cases} (1 - \bullet_2) ra_1 \bullet_1 \geq 0 & (c_5) \\ (\bullet_2 - \bullet_1) ra_1 \bullet_1 \geq 0 & (c_6) \\ (\bullet_2 - 1) ra_1 \bullet_2 \geq 0 & (c_7) \\ (\bullet_2 - \bullet_1) ra_1 \bullet_0 > 0 & (c_8) \end{cases}$$

Due to constraint c_8, the symbol mapping must be fixed as $s(ra_1) = 1$ and $s(\bullet_0) = 1$, and from the remaining part $\bullet_2 - \bullet_1 > 0$, s must be defined as follows: $s(\bullet_2) = 1$, $s(\bullet_1) = 0$. Such a symbol mapping satisfies all constraints:

We can thus prove termination with

$$|revacc(t_1, t_2)| = \|t_1\|_l.$$

In this example, by moving again to the case of ground input, *reverse*(g, a), no additional proof (e.g., using term-size) is found.

7.4 Normalizing Expressions

Consider the following program *normal*/2, which rewrites expressions built up of some associative operator *op* into a normalized form:

$$\begin{array}{lll} (n_1)\ normal(f, f) & \leftarrow & \\ (n_2)\ normal(f, n) & \leftarrow & rewrite(f, f_1), ! \\ & & normal(f_1, n) \\ (r_1)\ rewrite(op(op(a, b), c), op(a, op(b, c))) & \leftarrow & ! \\ (r_1)\ rewrite(op(a, op(b, c)), op(a, l)) & \leftarrow & rewrite(op(b, c), l) \end{array}$$

Notice that the program is not a pure logic program, because of the cut operator !, whose operational behavior is to remove part of the SLD-tree. In our analysis, we will investigate finiteness of the complete SLD-tree, which, if satisfied, implies in a trivial way the same conclusion for any part of it.

Let us use the program to derive normal forms for expressions, thus querying it along the mode *normal*(g, f). This entails calls to *rewrite* from $\mathbb{D}$(*rewrite*(g, f)).

The usual first step in the analysis handles rigidity. The following constraints are obtained from the modes:

$$\begin{cases} n_2 = 0 \\ r_2 = 0 \end{cases}$$

Next, we set up the rigid acceptability conditions on *normal* and *rewrite*, which, after normalizing, boil down to

$$(N_1) : s(\forall [r_1^e f - r_2^e f_1 - r_0^e \geq 0 \Rightarrow n_1 f - n_1 f_1 > 0])$$

for the recursive *normal* clause, and for the *rewrite* clause, we obtain

$$(R_1^t): s(\forall[true \Rightarrow r_1(o_0 + o_1a + o_2o_0 + o_2o_1b + o_2o_2c) - r_1(o_0 + o_1b + o_2c) > 0)]).$$

As the first condition postulates an interargument relation for *rewrite*, we have to ensure its validity. Two conditions, corresponding to each defining clause, pop up:

$$(R_1): s(\forall[true \Rightarrow r_1^e(o_0 + o_0o_1 + o_1o_1a + o_1o_2b) + o_2c - r_2^e(o_0 + o_1a + o_2o_0 + o_2o_1bo_2o_2c) - r_0^e \geq 0])$$

$$(R_2): s(\forall[r_1^e(o_0 + o_1b + o_2c) - r_2^e\, l - r_0^e \geq 0 \Rightarrow r_1^e(o_0 + o_1a + o_2o_0 + o_2o_1b + o_2o_2c) - r_2^e(o_0 + o_1a + o_2l) - r_0^e \geq 0])$$

The next step is to rewrite the conditions into explicit constraints. (N_1) needs one evaluation step, and this results in the following right-hand-side formula:

$$(n_1 - r_1^e)\, f + (r_1^e - n_1)\, f_1 + r_0^e > 0$$

from which we obtain the following system of constraints:

$$\begin{cases} n_1 - r_1^e \geq 0 \\ r_1^e - n_1 \geq 0 \\ r_0^e > 0 \end{cases}$$

No evaluation nor substitution steps apply to R_1^t and R_1, and after rewriting them in a more suitable form, we obtain

$$s(\forall[true \Rightarrow r_1o_1a + (o_2 - 1)r_1o_1b + (o_2 - 1)r_1o_2c + r_1o_2o_0 > 0])$$

$$s(\forall[true \Rightarrow (r_1^e\, o_1 - r_2^e)o_1a + (r_1^e - r_2^e)o_1o_2b + (1 - r_2^e\, o_2)o_2c + (1 + o_1)r_1^e\, o_0 - (1 + o_2)r_2^e\, o_0 - r_o^e \geq 0])$$

These conditions are translated into the following constraints, for R_1^t:

$$\begin{cases} r_1o_1 \geq 0 \\ (o_2 - 1)r_1o_1 \geq 0 \\ (o_2 - 1)r_1o_2 \geq 0 \\ r_1o_2o_0 > 0 \end{cases}$$

and for R_1 into the system

$$\begin{cases} (r_1^e\ o_1 - r_2^e)o_1 \geq 0 \\ (r_1^e - r_2^e)o_1 o_2 \geq 0 \\ (1 - r_2^e\ o_2)o_2 c \geq 0 \\ (1 + o_1)r_1^e\ o_0 - (1 + o_2)r_2^e\ o_0 - r_o^e \geq 0. \end{cases}$$

Finally, we rewrite the right-hand side of R_2 using evaluation with its left-hand side into the following formula:

$$(r_2^e - r_1^e)o_1 a + (1 - o_2)r_1^e\ o_1 b + (1 - o_2)r_1^e\ o_2 c + (o_2 - 1)r_2^e\ l + (r_2^e - r_1^e\ 0_2)o_0 \geq 0$$

from which we derive following constraints:

$$\begin{cases} (r_2^e - r_1^e)o_1 \geq 0 \\ (1 - o_2)r_1^e\ o_1 \geq 0 \\ (1 - o_2)r_1^e\ o_2 \geq 0 \\ (o_2 - 1)r_2^e \geq 0 \\ (r_2^e - r_1^e\ 0_2)o_0 \geq 0 \end{cases}$$

Consider the strict inequality constraints $r_0^e > 0$ and $r_1 o_2 o_0 > 0$. They impose the following conditions on s: $s(r_0^e) = 1$, $s(r_1) = 1$, $s(o_2) = 1$, and $s(o_0) = 1$. After reflecting these fixes into the remaining constraints, and removing duplicate and redundant constraints, we are left with the following system:

$$\begin{cases} n_1 - r_1^e \geq 0 & (c_1) \\ r_1^e - n_1 \geq 0 & (c_2) \\ (r_1^e\ o_1 - r_2^e)o_1 \geq 0 & (c_3) \\ (r_1^e - r_2^e)o_1 \geq 0 & (c_4) \\ (1 + o_1)r_1^e - 2r_2^e - 1 \geq 0 & (c_5) \\ (r_2^e - r_1^e)o_1 \geq 0 & (c_6) \\ r_2^e - r_1^e \geq 0 & (c_7) \end{cases}$$

At this point, no further reductions can be made. Instead, we must iterate over the values of the remaining symbols. Let us select the symbol n_1 to iterate over. This is a good choice, as it is the only nonfixed symbol for the level coefficients for *normalize*, and its other coefficients are fixed to 0. The obvious assignment to make is $s(n_1) = 1$. (Starting with $s(n_1) = 0$ immediately leads to the unsolvable (over the booleans) system $-2r_2^e \geq 1$.)

(c_2) then implies $s(r_1^e) = 1$, and because of (c_4), we must take $s(r_2^e) = 1$. We end up with one more constraint, from (c_5): $o_1 - 2 \geq 0$.

Table I. Results on the Examples in De Schreye and Decorte [1994]

Program	Query	Us	LS	SSS	Suc	Term
permute	(g,f)	0.08	0.69	0.04	+++	+
duplicate	(g,f)	0.03	0.08	0.02	+++	+
sum	(g,g,f)	0.03	0.35	0.01	+++	+
merge	(g,g,f)	0.06	2.06	0.03	+++	+
reverse	(g,f,g)	0.04	0.17	0.04	+++	+
dis-con	(g)	0.18	0.34	0.02	+++	+

This constraint is *not* solvable over the boolean values, and the conclusion is that we are unable to prove termination when mapping all coefficients to boolean values.

The obvious solution is to map the functor coefficients to the natural numbers. In that case, the final constraint becomes solvable, and a possible mapping could be $s(o_1) = 2$.

As a conclusion, we can prove termination for *normal*/2 using the following level mapping, norm, and interargument relation:

$$|normal(t_1, t_2)| = \|t_1\|$$

$$|rewrite(t_1, t_2)| = \|t_1\|$$

$$\|op(t_1, t_2)\| = 1 + 2\|t_1\| + \|t_2\|$$

$$R^{rewrite} = \{(x_1, x_2) | x_1 \geq x_2 + 1\}$$

8. EXPERIMENTAL EVALUATION

We have implemented a prototype of the technique. Currently it consists of two different parts. The core of the system, which derives the constraint set, has been coded in *Prolog by BIM*, version 4.1.0. For the second part, which concentrates on the constraint solving, we have built on the finite-domain CLP-language ROPE [Vandecasteele and De Schreye 1994], which is written in Mercury.

As in Speirs et al. [1997], we have tested the performance of the system on three well-known test suites (collected by Lindenstrauss and Sagiv [1997]). Tables I–III show our results on each of the three test suites. The first two columns specify the program within the test suite and how it was queried. In the next column, we display the timings obtained from the mentioned prototype. The next two columns display the timings obtained from respectively the approaches in Lindenstrauss and Sagiv [1997] and Speirs et al. [1997]. The results of Speirs et al. [1997] were obtained after transforming the programs to Mercury. Both[2] sets of timings emerge from

[2]A third system, proposed in Codish and Taboch [1997], provides similar success and timing behavior as Lindenstraus and Sagiv [1997], on several example programs. As the authors did not yet provide success/timing results for the benchmarks, we have left the system out of the comparison.

Table II. Results on the Apt Test suite [Apt and Pedreschi 1994]

Program	Query	Us	LS	SSS	Suc	Term
append	(g,g,f)	0.03	0.17	0.03	+++	+
append	(f,f,g)	0.04	0.36	0.04	+++	+
list	(g)	0.02	0.06	0.03	+++	+
fold	(g,g,f)	0.04	0.29	0.03	+++	+
lte	none	0.03	0.14	0.04	+++	+
map	(g,f)	0.04	0.15	0.04	+++	+
member	(f,g)	0.02	0.09	0.01	+++	+
mergesort	(g,f)	0.27	36.0	0.04	−−−	+
mergesort_ap	(g,f,f)	0.34	27.96	0.07	+++	+
naive_rev	(g,f)	0.04	0.32	0.02	+++	+
ordered	(g)	0.03	0.21	0.02	+++	+
overlap	(g,g)	0.05	0.23	0.03	+++	+
permutation	(g,f)	0.13	1.39	0.07	+++	+
quicksort	(g,f)	0.28	129.48	0.05	+++	+
select	(f,g,f)	0.03	0.14	0.01	+++	+
subset	(g,g)	0.05	0.33	0.06	+++	+
subset	(f,g)	0.07	0.33	0.06	−−−	−
sum	(f,f,g)	0.02	0.12	0.02	+++	+

Table III. Results on the Plümer Test Suite [Plümer 1990]

Program	Query	Us	LS	SSS	Suc	Term
mergesort.t	mergesort(g,f)	0.43	0.43	0.06	−+−	+
pl1.1	append(g,g,f)	0.03	0.14	0.03	+++	+
pl1.1	append(f,f,g)	0.03	0.11	0.03	+++	+
pl1.2	perm(g,f)	0.16	1.05	0.08	+−+	+
pl2.3.1	p(g,f)	0.04	0.01	0.03	−−−	−
pl3.5.6	p(f)	0.06	0.01	0.04	−−−	−
pl3.5.6a	p(f)	0.05	0.06	0.05	++−	+*
pl4.01	append3(g,g,g,f)	0.05	0.26	0.07	+++	+
pl4.4.3	merge(g,g,f)	0.08	1.99	0.04	+++	+
pl4.4.6a	perm(g,f)	0.09	0.31	0.06	+++	+
pl4.5.2	s(g,f)	0.03	0.03	0.03	−−−	−
pl4.5.3a	p(g)	0.03	0.00	0.04	−−−	−*
pl5.2.2	turing(g,g,g,f)	0.09	4.26	0.07	−−−	−
pl6.1.1	qsort(g,f)	0.13	131.18	0.07	+++	+
pl7.2.9	mult(g,g,f)	0.08	0.43	0.03	+++	+
pl7.6.2a	reach(g,g,g)	0.10	0.05	0.03	−−−	−
pl7.6.2b	reach(g,g,g,g)	0.14	0.25	0.18	−−−	−
pl7.6.2c	reach(g,g,g,g)	0.11	28.23	0.03	+++	+
pl8.2.1	mergesort(g,f)	0.29	35.82	0.06	−−−	+
pl8.2.1a	mergesort(g,f)	0.31	26.52	0.06	+++	+
pl8.3.1	minsort(g,f)	0.24	2.90	0.04	−−−	+
pl8.3.1a	minsort(g,f)	0.20	5.72	0.06	+++	+
pl8.4.1	even(g)/odd(g)	0.02	0.22	0.03	+++	+
pl8.4.2	e(g,f)	0.44	4.64	0.18	+++	+

the respective papers. The next column displays success or failure for each respective system: it contains a plus symbol if the system succeeds in proving termination and a minus symbol if it failed to do so. In the final

column, we specified whether the program is indeed terminating for the considered queries. Again, plus means terminating, and minus means that there exist queries among the ones considered for which the program gets into an infinite computation. Note, that, in some cases, the transformation to Mercury changes the termination properties of the program. Our conventions in such cases for expressing this in the tables is that we place either $+*$ or $-*$ in the final column. Here, $+*$ (resp., $-*$) should be read as $+$ (resp., $-$) for the columns for our own prototype and the one of Lindenstrauss and Sagiv [1997], but read as $-$ (resp., $+$) for the column of Speirs et al. [1997].

Let us first discuss the precision of the approach. Of all terminating test suites, there were only four programs for which the system failed to prove termination with respect to the considered queries: mergesort of Table II and minsort and two versions of mergesort of Table III. In each case we fail because the interargument relation derived is too general for the specific case. Consider for example the mergesort program with the call pattern $mergesort(g, f)$:

$$
\begin{aligned}
mergesort(Nil, Nil) &\leftarrow \\
mergesort([e], [e]) &\leftarrow \\
mergesort([e, f|u], v) &\leftarrow split([e, f|u], w, y), \\
&\quad\ mergesort(w, x), mergesort(y, z), \\
&\quad\ merge(x, z, v). \\
merge(x, Nil, x) &\leftarrow \\
merge(Nil, x, x) &\leftarrow \\
merge([a|x], [b|y], [a|z]) &\leftarrow a \leq b, \\
&\quad\ merge(x, [b|y], z). \\
merge([a|x], [b|y], [b|z]) &\leftarrow a > b, \\
&\quad\ merge([a|x], y, z). \\
split(Nil, Nil, Nil) &\leftarrow \\
split([e|u], [e|v], w) &\leftarrow split(u, w, v).
\end{aligned}
$$

Whereas the split predicate is never called with its arguments being empty lists, this can not be detected by the system, which derives the following relation:

$$\{(x_1, x_2, x_3) | x_1 \geq x_2 + x_3\}$$

To prove termination, the relation should express (for the first recursive call) that $x_1 > x_2$, as the partial data in the atom $split([e, f|u], w, y)$ prevents it from matching with $split(Nil, Nil, Nil)$. The other systems cannot deal with these examples either, with the exception of Lindenstrauss and Sagiv [1997] on mergesort.t of Table III. Most likely this is achieved through a better propagation of call information into the interargument derivation. On the other hand, Lindenstrauss and Sagiv [1997] fail

on the *perm* example of Table III, for which the two other systems are successful. On the whole, for these benchmarks, the precision of the three systems seems very comparable.

Note though that some examples of De Schreye and Decorte [1994] were not included in the benchmark of Lindenstrauss and Sagiv [1997]. Consider the *confused delete* program:

$$conf(x) \leftarrow delete_2(x, z), delete(u, y, z), conf(y)$$
$$delete_2(x, y) \leftarrow delete(u, x, z), delete(v, z, y)$$

Dealing with this program requires linear interargument relations of the type $x = y + 2$ (for $delete_2$) and $y = z + 1$ (for *delete*). Only our system is precise enough to generate them.

Other issues in enhanced precision relate to our ability to (automatically) consider any norm or level mapping of the given forms. In Lindenstrauss and Sagiv [1997] and Speirs et al. [1997], the norm is given as input to the systems. Below we discuss cases in which more refined norms are useful. To use them in the systems of Lindenstrauss and Sagiv [1997] or Speirs et al. [1997], one would either need to backtrack over different proof attempts (which would be extremely inefficient) or be very demanding on the creativity of the user. In our system, as opposed to Lindenstrauss and Sagiv [1997] and Speirs et al. [1997], the only input consists of the query pattern. We return to this discussion in Section 9.

Next, we evaluate the efficiency. All tests have been performed on similar machines. In the case of Lindenstrauss and Sagiv [1997], the machine was a SPARCstation 10 model 51 clone. Both the results from Speirs et al. [1997] and ours were obtained on a SPARCserver 1000. We deliberately did our experiments on this fairly old machine in order to make the comparison with the two other systems feasible. The SPECint92 ratings for these two machines are respectively 65.2 and 60.3, which are quite comparable.

Overall, Speirs et al. [1997] tends to be the fastest system, ours being somewhat slower and the one of Lindenstrauss and Sagiv [1997] the slowest. On most of the very small examples in the benchmark (those in Table I and most of those in Table II) our timings range between being equal to, to being at most double of the ones of Speirs et al. [1997]. On the somewhat larger examples, including most of the sorting examples, Speirs et al. [1997] tends to be up to five times faster than we. In these cases, note that Lindenstrauss and Sagiv [1997] can sometimes be 500 or more times slower than Speirs et al. [1997].

In evaluation of these differences, it should be noted that the work in Speirs et al. [1997] was not aimed at developing a novel approach to termination analysis, but at providing a highly optimized implementation of an existing termination analysis technique—optimizing some aspects of that technique in the process—using the very efficient programming language Mercury. Our own Prolog implementation is only prototypal. Providing an optimized implementation, in the style of Speirs et al. [1997], would be a different research issue. Moreover, our system was designed to be able

to cope with partially instantiated input, while Speirs et al. [1997] is for well-moded programs only. As a result, our analysis includes a richer component for combined mode-and-type analysis, which is likely to be (partly) responsible for the increased performance gap between our results and those of Speirs et al. [1997] for larger programs.

Note that our technique has been developed for directly recursive programs only. In order to deal with the mutually recursive programs in the benchmarks (dis-con in Table I, mergesort.t, pl8.4.1, and pl8.4.2 in Table III) we extended the system with a straightforward mutual-to-direct recursion transformation. In the case of predicates with only one argument position (dis-con and the even and odd of pl8.4.1), this transformation just adds a predicate symbol on top of the original predicates. As an example,

$$\begin{aligned} even(0) &\leftarrow \\ even(s(x)) &\leftarrow odd(x) \\ odd(s(x)) &\leftarrow even(x) \end{aligned}$$

becomes

$$\begin{aligned} m(even(0)) &\leftarrow \\ m(even(s(x))) &\leftarrow m(odd(x)) \\ m(odd(s(x))) &\leftarrow m(even(x)). \end{aligned}$$

For predicates with multiple arguments, this transformation is unreasonably weak, because the $m/1$ predicate has only one argument, and therefore the mode information for the original predicates gets lost (the single argument of $m/1$ will typically get mode "any" for all calls). Also, interargument relations ranging over the arguments of the original predicates can no longer be expressed in terms of the one argument of $m/1$.

In these cases, we used a transformation similar to the one above, but now we maintain the old arguments of mutually recursive predicates as different arguments of the m predicate and add one extra argument to represent the old predicate symbol (as a functor of arity one). Going back to the even/odd example, the transformation would then yield

$$\begin{aligned} m(even(\_), 0) &\leftarrow \\ m(even(\_), s(x)) &\leftarrow m(odd(\_), x) \\ m(odd(\_), s(x)) &\leftarrow m(even(\_), x). \end{aligned}$$

Note that this is a very weak approach to dealing with mutual recursion and that it puts extra precision requirements on the termination analysis. Specifically in clauses of the type

$$e(x, y) \leftarrow t(x, y)$$

taken from pl8.4.2, which become transformed to

$$m(e(\_), x, y) \leftarrow m(t(\_), x, y),$$

it is now crucial to allow norms that measure different functor symbols in different ways. In particular, the term-size norm would be insufficient.

In the tables, this weak treatment has its effect on the relatively slow treatment of dis-con, mergesort.t, and pl8.4.2. In future work, we aim to reformulate the entire technique in terms of mutual recursion, using the minimal cyclic collections of De Schreye et al. [1992].

Another observation that can be made from the tables is that we have a relatively slow performance on those examples which do not terminate, or for which we fail to find a proof. The reason for this is that we consider more potential termination proofs than the other approaches. Because we do not commit to one specific norm, when our analysis fails to prove termination, it has actually shown that the termination condition fails for all possible norms of the given type. In a successful proof, this has little effect on the efficiency of the analysis, since the system will terminate as soon as it finds one successful proof. In a failing proof, the constraint solver will take more time in establishing that no solution can be found. This inefficiency might be reduced by taking a satisfiability-checking approach to the constraint solving, instead of the finite-domain-, enumeration-, and pruning-based approach that we currently use.

Given the above observations, we believe that the experiments show the feasibility of the approach. For future work, it would be interesting to develop a much more optimized version of our prototype, preferably written in Mercury or C, and focused on well-moded programs only, so that a more accurate comparison with Speirs et al. [1997] would be possible.

9. RELATED WORK

Our technique is an instance of the "norm-based" approach to logic program termination analysis. There are many other techniques, both within the norm-based approach and outside it.

In De Schreye and Decorte [1994], we present an extensive survey of the different techniques proposed up till 1994. This includes discussions and comparisons between the different lines of work. In particular, it includes discussions on relations between different norm-based approaches, comparisons with transformational approaches (e.g., Rao et al. [1992]), with theorem-proving approaches (e.g., Baudinet [1992]), and others (e.g., Wang and Shyamasundar [1994], Brodsky and Sagiv [1989], and Franchez et al. [1985]). In Rao et al. [1998], this discussion is nicely summarized, extended with valuable additional issues, and brought up to date, especially on the level of the transformational techniques (including a.o. Aguzzi and Modigliani [1994], Arts and Zantema [1993], Ganzinger and Waldmann [1993], and Marchiori [1994]). We do not repeat these general comparative arguments here, but refer to De Schreye and Decorte [1994] and Rao et al. [1998] instead.

In this section, we discuss the relation between our technique and some other more recent contributions to the norm-based approach. We also argue

how our technique reduces the differences with the transformational approach to some extent.

First, with respect to other norm-based approaches, apart from our own previous work in Decorte et al. [1993] and Decorte et al. [1997], we are not aware of any other techniques in which suitable norms are automatically generated. This includes the techniques in Lindenstrauss and Sagiv [1997] and Speirs et al. [1997] mentioned in the previous section, but also the constraint-based approach in Mesnard [1993; 1996] and Mesnard and Maillard [1998]. In all these works, the norm is given as input to the analysis.

For the programs in the benchmarks of Section 8 this has no effect. For all terminating programs in these benchmarks the term-size norm is sufficient. In Section 8, we already provided some examples that require other norms. Note, though, that even for the programs in the benchmarks, term-size is insufficient to prove termination for non-well-moded queries. As soon as queries with partial input are considered, more-refined norms are needed to prove termination. The point is already illustrated with the *permute* example in Section 3. We feel this is not a minor point. The ability to compute with partially instantiated input is one of the most distinguishing features of logic programming. Thus, the termination analysis should be able to cope with it. In Section 3, we also discussed the *combine* example, in which term-size is unable to provide an appropriate linear interargument relation.

Probably the most relevant example, illustrating the need for generating appropriate norms, is given in Rao et al. [1993], which discusses the use of a transformational approach to logic program termination analysis in the context of verifying the ProCos compiler. The technique was successful in proving its termination. Rao et al. [1993] observed that norm-based techniques were unable to prove termination in this case. The heart of the problem is that the ProCos compiler contains clauses of the form

$$p(f(x), \ldots) \leftarrow p(g(x), \ldots)$$

$$p(g(x), \ldots) \leftarrow p(x, \ldots).$$

Similar to the even/odd example mentioned above, norm-based approaches have problems in finding a norm that decreases on the atoms in the first clause. As no real data consumption results from using the first clause, proving termination of the program requires an order relation on function symbols. Especially if there are a number of clauses of this type in the program, the way in which the norm measures the different functions (e.g., $f/1$ and $g/1$) needs to be tuned very carefully. Alternatively, a partial evaluation step would be needed to eliminate the problematic clause.

Transformational approaches to termination analysis do not have problems with such cases. Appropriate term orderings, in particular the simplification orderings [Dershowitz 1987], can automatically be generated by rewrite theorem provers.

Thanks to our symbolic norms and to the constraint-based termination analysis, our approach also automatically tunes the relative weights that the norm should assign to these different functors. In this respect, the technique brings the norm-based and transformational approaches closer together.

The concepts of symbolic level mappings and norms were already proposed in Verschaetse [1992] and Plümer [1990], but no elegant and practical use of them has been developed before, as far as we know. Rather, the only solution method which was advocated in these works was to perform an exhaustive search over all possible alternatives, i.e., considering all possible subsets of input positions for each predicate, and performing one separate analysis for each alternative. This of course strongly increases the complexity of the analysis.

To the best of our knowledge, the only attempt we are aware of for solving this problem in an elegant way is that of Sohn and Van Gelder [1991], who propose a solution based on linear programming techniques. However, we immediately stress that also their approach starts off from a fixed norm, namely list-length. Only the level mapping and the interargument relations are generated from symbolic versions. We refer to De Schreye and Decorte [1994] for a concise description of the approach, formulated in terms of the concepts introduced above.

In theory, this approach could be generalized to start off from a symbolic norm. However, by doing this, the linearity property, which is fundamental for applying the linear programming techniques, gets lost. Furthermore, inferring the interargument relations, using Van Gelder [1991] for example, on the basis of a symbolic norm seems difficult, since it involves global analysis. Introducing a further level of abstraction in the norm would make the analysis very complex.

Mesnard [1993; 1996] and Mesnard and Maillard [1998] use the technique of Sohn and Van Gelder [1991] in a constraint-based approach to prove termination of constraint logic programs. Technically, the main novelty in the approach is the introduction of several abstract versions of the analyzed program. Such programs are referred to as *approximations*. The final approximation is a constraint logic program over the domain of the booleans. From this program, using the techniques of Sohn and Van Gelder [1991], sufficient boundedness conditions on the arguments of the predicates can easily be inferred to ensure termination. As such, the approach has the advantage that the set of queries of interest does not need to be specified. Maximal sets of terminating queries are automatically inferred. Note, though, that the technique is again restricted to use of a specific norm.

In Decorte and Schreye [1998], we adapt our approach to also support this extended functionality. There, we automatically generate a finite set of norms, such that the set of queries that can be proved to terminate using these norms (and our termination condition) is maximized. However, in the process, the termination condition is slightly weakened. As such, this

extension is not strictly stronger than the technique presented here. Also, the extension is considerably less efficient.

Finally, our technique is very strongly related to our previous work in Decorte et al. [1993] and Decorte et al. [1997]. In Decorte et al. [1993], we propose two different techniques for inferring norms from type information. The first of these forms the basis of our rigidity analysis, presented in Section 6. The second technique introduces a stronger class of norms, called *typed norms*. Typed norms are strictly more expressive than the norms defined in this article. However, the use of typed norms requires much more complex methods for inferring interargument relations, as shown in Decorte et al. [1997]. Our final evaluation of this work was that the gained precision was not in proportion to the added complexity of the analysis.

10. CONCLUSION

We have proposed a termination condition which is parametrized by a set of symbolic constants which identify generic level mappings, norms, and interargument relations. We have shown how to explicitize systems of constraints automatically from that condition, and we have established the relationship between the solutions of the system of constraints and a correct termination proof. It is exactly the formulation of each phase in a constraint-based way that enables viable information to be shared. Because of the way the constraints express which conditions must minimally be satisfied for which symbols, there is only a minimal negative constraining from one phase on another.

Like the condition in Definition 2.7 of De Schreye et al. [1992], the new termination condition can, in principle, be used on any set of calls of interest, and it focuses very naturally on the recursive structure of the clauses, resulting in natural norms and level mappings in practice. Unlike in Definition 2.7, the condition is expressed in terms of the syntactic structure of the clauses themselves, and not in terms of $Call(P, S)$. Treating $Call(P, S)$ is now considered as a separate condition on the level mapping: it should be rigid on these atoms. As such, this condition comes closer to the standard notions of acceptability found in the literature.

The approach is able to deal in an easy and natural way with examples, such as the verification of the ProCos compiler [Rao et al. 1993], which could not easily be handled by norm-based approaches before. It has also been formulated in the context of nonground inputs, which is not supported by many other systems.

A next feature is the ability to synthesize linear inequality interargument relations, which, as discussed, are more expressive than equality relations. However, currently the approach is limited to the derivation of only one inequality for each predicate. This could prove too restricting for cases where independent relations between two sets of input and output argument positions exist.

A final observation concerns the modularity of our condition which is expressed in the computation of each phase as a separate system of

constraints. The solvability of the unified system provides different instances of termination proofs. However, the analysis can be started equally well from incomplete data. Solving a system corresponding to the results of only two phases expresses those conditions under which the third phase is solvable. An example of an application here is the inference of sets of terminating queries for the specific program under consideration. They are obtained by considering as input only those systems corresponding to the rigid acceptability and the validity condition. Decorte and Schreye [1998] shows how a minor extension of the technique provides this functionality.

The approach is applicable with no further difficulties in modular program development systems, mainly because the rigid acceptability condition is expressed completely at the clause level. Here, the information obtained for one module (in the form of constraints) can be preserved for later use. When later a new module is added (potentially supplying new definitions for existing predicates), acceptability of the extended system coincides with the solvability of the merged systems of constraints obtained from each separate module. It is exactly the use of constraints that allows us to express conditions in terms of predicates which do not have to be known at that moment. For these predicates a number of minimally required conditions are proposed. Moreover, the new module can be analyzed without having to restart the analysis for the modules on which it builds, by simply importing the previously generated constraints for that module.

A prototype implementation exists. The experiments performed with the system are satisfactory. A more extensive implementation effort will be required to establish whether the most efficient current system can be surpassed.

ACKNOWLEDGMENTS

We thank Zoltan Somogyi and Naomi Lindenstrauss for valuable information concerning their own tests. We also thank the anonymous referees for their very relevant questions and remarks that helped us improve the article.

REFERENCES

AGUZZI, G. AND MODIGLIANI, U. 1994. Proving termination of logic programs by transforming them into equivalent term rewriting systems. In *Proceedings of FST and TCS*. LNCS, vol. 761. Springer-Verlag, 114–124.

APT, K. 1990. Logic programming. In *Handbook of Theoretical Computer Science*. Vol. B, J. van Leeuwen, Ed. Elsevier Science Publishers.

APT, K. AND BEZEM, M. 1991. Acyclic programs. *New Gen. Comput*. 9, 335–363.

APT, K. AND PEDRESCHI, D. 1990. Studies in pure Prolog: Termination. In *Proceedings Esprit Symposium on Computational Logic*. Springer-Verlag, 150–176.

APT, K. AND PEDRESCHI, D. 1991. Proving termination of general Prolog programs. In *Proc. International Conference on Theoretical Aspects of Computer Science*.

APT, K. AND PEDRESCHI, D. 1994. Modular termination proofs for logic and pure Prolog programs. In *Advances in Logic Programming Theory*. Oxford University Press, 183–229.

ARTS, T. AND ZANTEMA, H. 1993. Termination of logic programs via labelled term rewrite systems. In *Proceedings of CSN'95*. 22–34.

BAUDINET, M. 1992. Proving termination properties of Prolog programs: A semantic approach. *J. Logic Program. 14*, 1–29.

BEZEM, M. 1992. Characterizing termination of logic programs with level mappings. *J. Logic Program. 15*, 1 & 2, 79–98.

BOSSI, A., COCCO, N., AND FABRIS, M. 1991. Proving termination of logic programs by exploiting term properties. In *Proc. CCPSD-TAPSOFT'91*. Springer-Verlag, LNCS 494, 153–180.

BOSSI, A., COCCO, N., AND FABRIS, M. 1992. Typed norms. In *Proc. ESOP'92*, B. Krieg-Brueckner, Ed. Springer-Verlag, LNCS 582, 73–92.

BRODSKY, A. AND SAGIV, Y. 1989. On termination of Datalog programs. In *First International Conference on Deductive and Object Oriented Databases*. 95–112.

BRODSKY, A. AND SAGIV, Y. 1991. Inference of inequality constraints in logic programs. In *Proceedings of the 10th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*. 227–240.

BRONSARD, F., LAKSHMAN, T., AND REDDY, U. 1992. A framework of directionality for proving termination of logic programs. In *Proc. JICSLP '92*, K. Apt, Ed. MIT Press, 321–335.

BRUYNOOGHE, M. 1991. A practical framework for the abstract interpretation of logic programs. *J. Logic Program. 10*, 2, 91–124.

BRUYNOOGHE, M. AND BOULANGER, D. 1994. Abstract interpretation for (constraint) logic programming. In *Constraint Programming*, J. P. B. Mayoh, E. Tyugu, Ed. NATO ASI Series, vol. F/131. Springer-Verlag, 228–258.

CODISH, M. AND TABOCH, C. 1997. A semantic basis for termination analysis of logic programs and its realization using symbolic norm constraints. In *Proceedings of the Sixth International Conference on Algebraic and Logic Programming*. To appear.

COUSOT, P. AND HALBWACHS, N. 1978. Automatic discovery of linear restraints among variables of a program. In *Proceedings 5th ACM Symposium on Principles of Programming Languages*. 84–96.

DE SCHREYE, D. AND DECORTE, S. 1994. Termination of logic programs: the never-ending story. *J. Logic Program. 19* & *20*, 199–260.

DE SCHREYE, D. AND VERSCHAETSE, K. 1992. Termination analysis of definite logic programs with respect to call patterns. Tech. Rep. CW 138, Department Computer Science, K.U.Leuven.

DE SCHREYE, D. AND VERSCHAETSE, K. 1995. Deriving linear size relations for logic programs by abstract interpretation. *New Gen. Comput. 13*, 2, 117–154.

DE SCHREYE, D., VERSCHAETSE, K., AND BRUYNOOGHE, M. 1992. A framework for analysing the termination of definite logic programs with respect to call patterns. In *Proc. FGCS'92*. ICOT, Tokyo, 481–488.

DEBRAY, S. 1989. Static inference of modes and data dependencies in logic programs. *ACM Trans. Program. Lang. Syst. 11*, 3 (July), 418–450.

DEBRAY, S. K. AND WARREN, D. S. 1986. Automatic mode inference for Prolog programs. In *Proceedings 1986 Symposium on Logic Programming*. 78–88.

DECORTE, S. AND DE SCHREYE, D. 1998. Termination analysis: Some practical properties of the norm and level mapping space. In *Proceedings IJCSLP98, the International Joint Conference and Symposium on Logic Programming*, J. Jafar, Ed. MIT Press, 235–249.

DECORTE, S., DE SCHREYE, D., AND FABRIS, M. 1993. Automatic inference of norms: A missing link in automatic termination analysis. In *Proceedings ILPS'93*, D. Miller, Ed. 420–436.

DECORTE, S., DE SCHREYE, D., AND FABRIS, M. 1997. Exploiting the power of typed norms in automatic inference of interargument relations. Tech. Rep. 246, Department of Computer Science, K.U.Leuven, Belgium.

DERSHOWITZ, N. 1987. Termination of rewriting. *J. Symbol. Comput. 3*, 1 & 2, 69–116.

FALASCHI, M., LEVI, G., MARTELLI, M., AND PALAMIDESSI, C. 1989. Declarative modelling of the operational behaviour of logic languages. *Theoret. Comput. Sci. 69*, 3, 289–318.

FRANCHEZ, N., GRUMBERG, O., KATZ, S., AND PNUELI, A. 1985. Proving termination of Prolog programs. In *Logics of Programs*, R. Parikh, Ed. Springer-Verlag, 89–105.

GANZINGER, H. AND WALDMANN, U. 1993. Termination proofs of well-moded logic programs via conditional rewrite systems. In *Proceedings of CTRS'92*. LNCS 656. Springer-Verlag, 216–222.

JANSSENS, G. AND BRUYNOOGHE, M. 1992. Deriving descriptions of possible values of program variables by means of abstract interpretation. *J. Logic Program. 13*, 2 & 3, 205–258.

LINDENSTRAUSS, N. AND SAGIV, Y. 1997. Automatic termination analysis of logic programs. In *Proc. 14th International Conference on Logic Programming*, L. Naish, Ed. 63–77.

LLOYD, J. 1987. *Foundations of Logic Programming*. Springer-Verlag.

MARCHIORI, M. 1994. Logic programs as term rewriting systems. In *Proceedings of ALP'94*. LNCS 850. Springer-Verlag, 223–241.

MARTELLI, A. AND MONTANARI, U. 1982. An efficient unification algorithm. *ACM Trans. Program. Lang. Syst. 4*, 2, 258–282.

MARTIN, J. C., KING, A., AND SOPER, P. 1996. Typed norms for typed logic programs. In *Proceedings of LOPSTR'96: Logic Program Synthesis and Transformation*, J. Gallagher, Ed. Number 1207 in LNCS. Springer-Verlag, 143–153.

MELLISH, C. 1985. Some global optimizations for a Prolog compiler. *J. Logic Program. 2*, 1 (April).

MESNARD, F. 1993. Etude de la terminaison des programmes logiques avec contraintes au moyen d'approximations. Ph.D. thesis, Paris VI.

MESNARD, F. 1996. Inferring left-terminating classes of queries for constraint logic programs. In *Proc. IJCSLP96*. MIT-Press, Bonn, Germany, 7–21.

MESNARD, F. AND MAILLARD, A. 1998. Clp(x) for automatically proving program properties. *J. Logic Program. 37* (1–3), 77–94.

MULKERS, A. 1993. *Deriving Live Data Structures in Logic Programs by Means of Abstract Interpretation*. Number 675 in LNCS. Springer-Verlag.

PFENNING, F., ED. 1992. *Types in Logic Programming*. MIT Press.

PLÜMER, L. 1990. *Termination Proofs for Logic Programs*. Number 446 in LNAI. Springer-Verlag.

RAO, M. K., KAPUR, D., AND SHYAMASUNDAR, R. 1998. Transformational methodology for proving termination of logic programs. *J. Logic Program. 34*, 1, 1–42.

RAO, M. K., KAPUR, D., AND SHYAMASUNDAR, R. K. 1992. A transformational methodology for proving termination of logic programs. In *Proc. CSL'92*. LNCS 626. Springer.

RAO, M. K., PANDYA, P., AND SHYAMASUNDAR, R. K. 1993. Verification tools in the development of provably correct compilers. In *Proceedings 5th Symposium on Formal Methods Europe FME'93*.

SOHN, K. AND VAN GELDER, A. 1991. Termination detection in logic programs using argument sizes. In *Proceedings 10th Symposium on Principles of Database Systems*. ACM Press, 216–226.

SPEIRS, C., SOMOGYI, Z., AND SONDERGAARD, H. 1997. Termination Analysis for Mercury. In *Proceedings of the Fourth International Symposium on Static Analysis*, P. Van Hentenryck, Ed. Number 1302 in LNCS. Springer, 157–171.

ULLMAN, J. AND VAN GELDER, A. 1988. Efficient tests for top-down termination of logical rules. *J. ACM 35*, 2 (April), 345–373.

VAN GELDER, A. 1991. Deriving constraints among argument sizes in logic programs. *Ann. Math. Artif. Intell. 3*.

VANDECASTEELE, H. AND DE SCHREYE, D. 1994. Implementing a finite-domain CLP-language on top of Prolog: A transformational approach. In *Proceedings of Logic Programming and Automated Reasoning*, F. Pfenning, Ed. Lecture Notes in Artificial Intelligence 822. Springer-Verlag, 84–98.

VERSCHAETSE, K. 1992. Static termination analysis for definite Horn clause programs. Ph.D. thesis, Dept. Computer Science, K.U.Leuven. Accessible via http://www.cs.kuleuven.ac.be/~lpai.

WANG, B. AND SHYAMASUNDAR, R. 1994. A methodology for proving termination of logic programs. *J. Logic Program. 21*, 1, 1–30.

Received January 1998; revised October 1998; accepted March 1999