

Proving Termination for Logic Programs by the Query-Mapping Pairs Approach

Naomi Lindenstrauss¹, Yehoshua Sagiv¹, and Alexander Serebrenik²

¹ School of Computer Science and Engineering, The Hebrew University,
Jerusalem 91904, Israel

{naomil,sagiv}@cs.huji.ac.il

² École Polytechnique (STIX), 91128 Palaiseau Cedex, France
Alexander.Serebrenik@stix.polytechnique.fr

©Springer-Verlag (www.springer.de/comp/lncs/index.html)

Abstract. This paper describes a method for proving termination of queries to logic programs based on abstract interpretation. The method uses query-mapping pairs to abstract the relation between calls in the LD-tree associated with the program and query. Any well founded partial order for terms can be used to prove the termination. The ideas of the query-mapping pairs approach have been implemented in SICStus Prolog in a system called *TermiLog*, which is available on the web. Given a program and query pattern the system either answers that the query terminates or that there *may* be non-termination. The advantages of the method are its conceptual simplicity and the fact that it does not impose any restrictions on the programs.

1 Introduction

In this paper we describe a method for proving termination of queries to logic programs based on abstract interpretation. The results of applying the ideas of abstract interpretation to logic programs (cf. [24]) seem to be especially elegant and useful, because we are dealing in this case with a very simple language which has only one basic construct—the clause. Termination of programs is known to be undecidable, but in the case of logic programs, which have a clearly defined formal semantics and in which the only possible cause for non-termination is infinite recursion, it is possible to prove termination automatically for a large class of programs.

Given a logic program, that is a finite set of clauses of the form

$$A : - B_1, B_2, \dots, B_n$$

where A, B_1, B_2, \dots, B_n are atoms, $n \geq 0$, and a query, which is an atom, we want to find, if possible, substitutions for the variables of the query which make it a logical consequence of the program (if there are no variables in the query we just want to show that it is a logical consequence of the program). To do so we usually use SLD-resolution to compute answers to the query (for all these notions see [1, 48]).

A crucial question in this case is whether the computation terminates. The way the computation proceeds depends on the choice of the atom in the goal on which resolution is performed at each step, and on the choice of the clause with which it is resolved. Termination irrespective of which atom and clause are chosen is called *strong termination* (cf. [12]) and means that all SLD-trees constructed for the program and the query are finite. One can consider a weaker notion of termination, where one can choose any clause, but the choice of atom is determined by Prolog's computation rule: always choose the leftmost atom in a goal to resolve upon. This notion amounts to finiteness of the SLD-tree constructed according to Prolog's computation rule, which is called the LD-tree (cf. [1]). A still weaker notion is \exists -termination, where one assumes that there is at least one way to choose atoms so that for any way of choosing clauses there is termination (cf. [65, 57]). All notions of termination where one can choose any clause fall in the category of *universal termination*, because one computes all answers. The weakest notion of termination is *existential termination*, which means that there either is finite failure or at least one way to choose atoms and clauses, so that there is a successful derivation (cf. [80, 8, 49]).

All these kinds of termination are undecidable (this follows from the fact that the operation of a Turing machine can be described by a logic program, and the halting problem for Turing machines is undecidable). Nevertheless it is important to find sufficient conditions for termination that can be verified automatically.

We consider the second notion of termination mentioned above, namely termination of computing all answers using the leftmost computation rule of Prolog. This notion of termination is also known as LD-termination (cf. [32]). Observe, that finding all answers in finite time is essential, even if one seems to be interested in finding a single answer only. The query we solved may be backtracked into and it is crucial that there will be termination also in that case (cf. the section on 'Backwards Correctness' in [56]). Moreover, Prolog's built-in predicates *findall*, *setof*, *bagof* depend on computing all answers to the query they include.

One of the difficulties when dealing with the LD-tree for a query, given a logic program, is that infinitely many different atoms may appear as subgoals. The basic idea is to abstract this possibly infinite structure to a finite one. The query-mapping pairs method (cf. [66, 44]) uses a certain kind of graphs to abstract the relation between arguments of calls in the LD-tree associated with the program and query. The method has been implemented in SICStus Prolog ([68]) in a termination analyzer called *TermiLog* (cf. [46]), which is available on the web ([74]). Given a program and query pattern the analyzer either answers that the query terminates or that there *may* be non-termination.

TermiLog was, as far as we know, the first publicly available automatic tool for proving termination of logic programs. It is based on one clear and simple idea — the notion of query-mapping pair — and the use of Ramsey's theorem. This paper presents both the theoretical framework, which can be used for any well-founded order, and its application for linear norms, as implemented in *TermiLog*. The paper is completely self-contained. For instance the instantiation

analysis, which is an extension of groundness analysis, and the constraint inference are developed from the basics. The whole development of *TermiLog* was done within the framework of standard Prolog. This and the inherent simplicity of the approach make the system very flexible and various optimizations can be easily added.

The remainder of the paper is organized as follows. In Section 2 the key notion of the approach, *query-mapping pairs*, is introduced. Query-mapping pairs are defined relative to a partial mapping ϕ from terms to a strictly ordered well-founded set. Any partial mapping ϕ from terms to a strictly ordered well-founded set can be taken. In Section 3 the Basic Theorem is proved with the help of Ramsey's Theorem. From the Basic Theorem a sufficient condition for termination, formulated in terms of query-mapping pairs, is derived. An example of using the condition is given for a suitable mapping ϕ . However, in order to automate the process of proving termination, we cannot let the choice of ϕ be determined by ingenuity, but need a uniform way of constructing it. This is done by means of linear norms. Section 4, which comprises the main part of the paper, explains the theoretical foundation for the use of query-mapping pairs based on linear norms, as it is implemented in the *TermiLog* system ([74, 46]). First linear norms and the more general symbolic linear norms are defined. Then the weighted rule graph that corresponds to a rule³ of the program is defined. This graph is a main tool in the construction of query-mapping pairs and in the constraint inference (Subsection 4.7). Then it is explained how to obtain all the query-mapping pairs relevant to a program and a query by the processes of generation and composition. Next, it is explained how instantiation analysis and constraint inference, which are used in the construction of the query-mapping pairs, are performed, by a process of bottom-up abstract interpretation. We then give some information about the implementation and the experimental evaluation of the system. The paper ends with some information about related work and the development of the query-mapping pairs approach.

2 LD-Trees and Query-Mapping Pairs

To prove termination we use partial mappings from terms to strictly ordered well-founded sets. A set \mathcal{S} is called *strictly ordered* if there is a binary relation $>$ defined on it that is transitive (i.e. if x, y, z are in \mathcal{S} then $x > y, y > z$ implies $x > z$) and asymmetric (i.e. if x, y are in \mathcal{S} then $x > y$ implies that $y > x$ cannot hold). Note that this also means that it cannot be reflexive. A strictly ordered set $(\mathcal{S}, >)$ is called *well-founded* if there are no infinite sequences $x_1 > x_2 > x_3 > \dots$ of elements of \mathcal{S} . The way to prove termination will be to show that if there was non-termination one could produce an infinite descending sequence of elements.

We will consider partial mappings ϕ from terms to a strictly ordered well-founded set $(\mathcal{S}, >)$. Most often this will be the set of non-negative integers with the usual order, but any other strictly ordered well-founded set can be used. We

³ A rule is a clause with non-empty body.

will require these mappings to be compatible with substitutions in the sense of the following definition.

Definition 2.1 (Substitution-Compatible Mapping). *A partial mapping ϕ defined for (not necessarily all) terms is called substitution-compatible if whenever $\phi(T)$ is defined for a term T and θ is a substitution, then $\phi(T\theta)$ is defined too and is equal to $\phi(T)$.*⁴

We now proceed to define a relation between nodes in the LD-tree that 'call' each other.

Definition 2.2. *Let P be a program and Q be a query. Let $\leftarrow P_1, \dots, P_m$ and $\leftarrow Q_1, \dots, Q_n$ be nodes in the LD-tree of P and Q . We say that node $\leftarrow Q_1, \dots, Q_n$ is a direct offspring of node $\leftarrow P_1, \dots, P_m$ if P_1 has been resolved with a clause c in P and Q_1 is, up to a substitution, a body subgoal of c .*

Suppose we assign to each node in the LD-tree a unique natural number (it does not matter how, as long as there is a one-to-one correspondence between nodes and numbers), and suppose that if node (k) is resolved with the clause instance $A \leftarrow B_1, \dots, B_n$ we add a subscript (k) to all the predicates of the atoms $B_j, 1 \leq j \leq n$. Then, if the subscript of the predicate of the first atom of node (l) is (m) this means that node (l) is a direct offspring of node (m)

Definition 2.3 (Offspring Relation and Call Branch). *We define the offspring relation as the non-reflexive closure of the direct offspring relation. We call a path between two nodes in the tree such that one is the offspring of the other a call branch.*⁵

Take for example the program for computing Ackermann's function with the goal $\text{ack}(s(0), s(s(0)), A)$:

Example 2.1.

- (i) $\text{ack}(0, N, s(N))$.
- (ii) $\text{ack}(s(M), 0, A) :- \text{ack}(M, s(0), A)$.
- (iii) $\text{ack}(s(M), s(N), A) :- \text{ack}(s(M), N, A1), \text{ack}(M, A1, A)$.

□

The LD-tree is given in Figure 1. Note that the predicate of each atom in the LD-tree has a subscript that reports who is its 'parent', that is the node in the LD-tree that caused this atom to be called as the result of resolution. Node

⁴ The reader may notice a similarity between this notion and the notion of a rigid norm (cf. for example [25]). The difference is that a norm is defined for all terms. Here the mapping is partial and a crucial part of the requirements for it to be substitution-compatible is, that it be defined for $T\theta$, for any substitution θ , if it is defined for T .

⁵ There is affinity between our offspring relation and the descendant relation in [69], however the relation here is defined for nodes in the LD-tree, while the relation there is defined for atoms.

(2) and node (6) are, for instance, direct offspring of node (1), because the first atoms in their respective goals come from the body of clause (iii), with which the goal of node (1) was resolved. Node (3), for instance, is an offspring of node (1), but not a direct offspring.

(1) $\leftarrow \text{ack}(s(0), s(s(0)), A)$
(2) $\leftarrow \text{ack}_{(1)}(s(0), s(0), A1), \text{ack}_{(1)}(0, A1, A)$
(3) $\leftarrow \text{ack}_{(2)}(s(0), 0, A2), \text{ack}_{(2)}(0, A2, A1), \text{ack}_{(1)}(0, A1, A)$
(4) $\leftarrow \text{ack}_{(3)}(0, s(0), A2), \text{ack}_{(2)}(0, A2, A1), \text{ack}_{(1)}(0, A1, A)$
 $\quad \{A2 \mapsto s(s(0))\}$
(5) $\leftarrow \text{ack}_{(2)}(0, s(s(0)), A1), \text{ack}_{(1)}(0, A1, A)$
 $\quad \{A1 \mapsto s(s(s(0)))\}$
(6) $\leftarrow \text{ack}_{(1)}(0, s(s(s(0))), A)$
 $\quad \{A \mapsto s(s(s(s(0))))\}$
(7) \leftarrow

Fig. 1. LD-tree of *ack*

A graphical representation of the direct offspring relation is given in Figure 2.

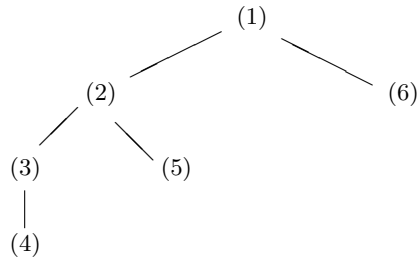


Fig. 2. The offspring relation of *ack*

In this example, to which we will return later, there is only one branch in the LD-tree. To give an example where the tree consists of more than one branch, take the program *pqrst*:

Example 2.2.

$p(X) \text{ :- } q(X), r(X).$

$p(X) \text{ :- } s(X).$

$s(X) \text{ :- } t(X).$

$q(a). \quad r(a). \quad t(b).$

□

In this case the LD-tree is given in Figure 3 and a representation of the offspring relation is given in Figure 4.



Fig. 3. LD-tree of $pqrst$

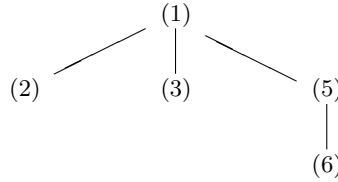


Fig. 4. The offspring relation of $pqrst$

In order to give information about call branches in the LD-tree we will use *argument mappings*, or *mappings* in short.

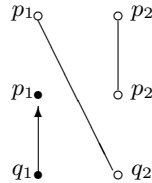
Definition 2.4 (Argument Mapping). *An argument mapping is a mixed graph, that is a graph with both arcs and edges, whose nodes correspond to argument positions of some atoms (possibly just one atom). A node corresponding to an argument position of an atom is labeled by the predicate of the atom and by the argument number. Nodes are either black or white. Nodes connected by an edge must be of the same color. Nodes connected by an arc must be black.*

The intuition behind this definition is the following: as we said in the beginning of this section we will use partial mappings ϕ from terms to a strictly ordered well-founded set. Nodes of the argument mapping correspond to arguments of atoms. The color of a node, black or white, will be used to depict whether ϕ is defined or, respectively, not known to be defined for the argument. An arc going from one node to another depicts the fact that the value of ϕ for the argument corresponding to the first node is greater than the value for the argument corresponding to the second. Nodes connected by an arc must therefore be black, because ϕ is defined for the arguments they represent. An edge can either connect two black nodes for which ϕ is defined and the values are equal, or two nodes for which the corresponding arguments are identical (and then ϕ is defined or undefined for both). Suppose we have the rule

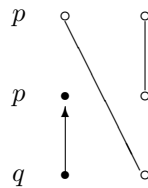
$p(X,Y) :- p(a,Y), q(b,X).$

and $\phi(a) = 1, \phi(b) = 2.$

Then we can depict the relations between the arguments of the atoms appearing in the rule by



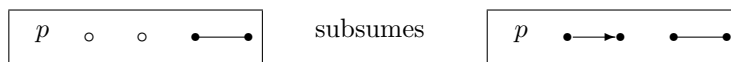
In order not to clutter the picture we will usually omit some of the labels and imply them by the layout of the graph:



One mapping may be more 'general' than another. This brings us to the definition of subsumption.

Definition 2.5 (subsumption). *Given two mappings G_1 and G_2 , we say that G_1 subsumes G_2 if they have the same nodes up to color, every node that is black in G_1 is also black in G_2 , and every edge or arc between nodes in G_1 also appears for the respective nodes in G_2 .*

The intuition behind this definition is that we will assume that ϕ is substitution-compatible, so when we apply a substitution to an atom, the arguments for which ϕ was defined will continue to be defined and have the same value, while it may happen that ϕ will be defined for more arguments. If we have a mapping, and apply a substitution to the arguments it represents, the resulting mapping will be subsumed by the original mapping. To give an example:



We now define *basic query-mapping pairs*. Basic query-mapping pairs give information about the size relations between the arguments of the first atoms of two nodes such that one is the direct offspring of the other.

Definition 2.6 (Basic Query-Mapping Pairs). *Let ϕ be a partial mapping from terms to a strictly ordered well-founded set $(S, >)$ that is substitution-compatible. Let a logic program and a query be given. Suppose $\leftarrow P_1, \dots, P_m$ and $\leftarrow Q_1, \dots, Q_n$ are two nodes in the LD-tree such that the second is a direct offspring of the first. Let θ be the composition of the substitutions along the path between these two nodes.*

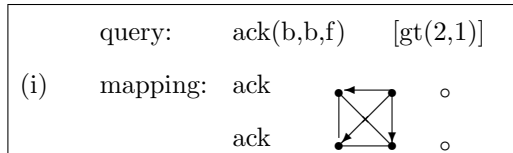
A basic query-mapping pair corresponding to these two nodes consists of two parts:

- The query pattern, that is a mapping whose nodes correspond to the argument positions of $P_1 = p(t_1, \dots, t_k)$ and are either black, if we know that ϕ is defined for the corresponding argument, or white, if we don't know whether ϕ is defined for the corresponding argument. If we know that the value of ϕ is equal for the i 'th and j 'th arguments or that the arguments are identical, the graph will include an edge from the i 'th to the j 'th position. If we know that $\phi(t_i) > \phi(t_j)$, the graph will include an arc from the i 'th to the j 'th position.
- A mapping, whose nodes correspond to the argument positions of $P_1\theta$ and the argument positions of Q_1 . Again nodes can be black or white and there can be edges and arcs between them, with the meaning as above. We call the nodes corresponding to the argument positions of $P_1\theta$ with the edges and arcs between them the domain of the mapping, and the nodes corresponding to the argument positions of Q_1 with the edges and arcs between them the range of the mapping.

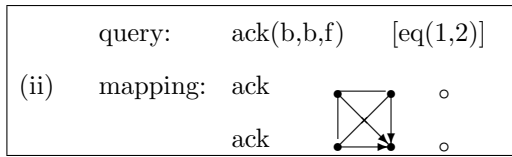
Since ϕ is assumed to be substitution-compatible, the query pattern of a query-mapping pair subsumes the domain of its mapping. Note that the information given by the query-mapping pair does not have to be complete—the important thing is that it be sound and hopefully sufficient for the proof of termination. The query describes abstractly the atom P_1 . The mapping describes the relation between the atoms $P_1\theta$ and Q_1 . By the time we arrive at Q_1 certain substitutions will have been applied to the variables of P_1 and we want to take them into account in order to have ϕ , which we will use to prove the termination, be defined on as many nodes as possible.

We use the following notation for the query-mapping pairs: for the query pattern we first give an atom with the predicate of P_1 and arguments b or f , depending on whether the corresponding nodes are black or white, and then a list of the edges and arcs in the form $eq(i, j)$ for an edge between the i 'th and j 'th argument position and $gt(i, j)$ for an arc from the i 'th to the j 'th argument position; for the part of the mapping we use a pictorial representation.

If we take node (1) and node (2) of the LD-tree for Ackermann's function and assume that ϕ gives for each number written in successor notation its corresponding (nonnegative) value, we get the basic query-mapping pair



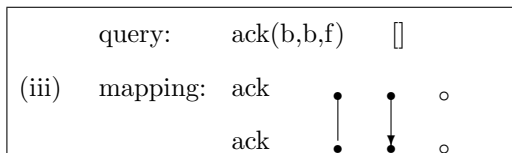
If we take node (2) and its direct offspring node (3), we get the following basic query-mapping pair



Looking at each pair of nodes in the LD-tree that are direct offspring of each other is not practical, because the tree may be infinite. In our case both nodes (2) and node (3) originate from resolution with clause (iii) and their first atoms originate in the first atom of the body of the clause. So we may just look at the relation between the head and first body atom in

$\text{ack}(s(M), s(N), A) \text{ :- ack}(s(M), N, A1), \text{ack}(M, A1, A).$

Then we get the basic query-mapping pair



Note that this pair subsumes the previous two.

Basic query-mapping pairs express relations between first atoms of nodes that are direct offspring of each other. To express relations between first atoms of nodes that are offspring but not direct offspring of each other we compose query-mapping pairs.

We have seen that for some mappings we have defined domain and range. For such mappings we define composition.

Definition 2.7 (Composition of Mappings). *Let μ and ν be mappings with domain and range. If the range of μ and the domain of ν are labeled by the same predicate, then the composition of the mappings μ and ν , denoted $\mu \circ \nu$, is obtained by unifying each node in the range of μ with the corresponding node in the domain of ν (two nodes are corresponding if they correspond to the same argument position). When unifying two nodes, the result is a black node if at least one of the nodes is black, otherwise it is a white node. If a node becomes black, so do all nodes connected to it with an edge. The nodes of the domain of $\mu \circ \nu$ are the nodes of the domain of μ , and the nodes of the range of $\mu \circ \nu$ are the nodes of the range of ν . The edges and arcs of $\mu \circ \nu$ consist of the transitive closure of the union of the edges and arcs of μ and ν .*

The following figure shows two mappings (left) and their composition (right).



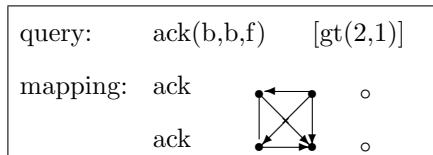
Definition 2.8 (Consistency). A mapping is consistent if it has no positive cycle (a positive cycle is a cycle consisting of edges and at least one arc where all arcs are in the same direction).

Note that a mapping that expresses relations in an LD-tree must be consistent. Indeed, assume that this is not the case, that is, there exists an inconsistent mapping expressing relations in the LD-tree. A positive cycle may only have black nodes. Because of transitivity we get that for each argument t corresponding to a node on the positive cycle we must have $\phi(t) < \phi(t)$, in contradiction to the irreflexivity of the order $<$.

Definition 2.9 (Summary). If μ is a consistent mapping with domain and range, then the summary of μ consists of the nodes in the domain and range of μ and the edges and arcs between these nodes. The summary is undefined if μ is inconsistent.

Definition 2.10 (Composition of Query-Mapping Pairs). Let (π_1, μ_1) and (π_2, μ_2) be query-mapping pairs, such that the range of μ_1 is identical to π_2 . The composition of (π_1, μ_1) and (π_2, μ_2) is (π_1, μ) , where μ is the summary of $\mu_1 \circ \mu_2$ (and, hence, the composition is undefined if $\mu_1 \circ \mu_2$ is inconsistent).

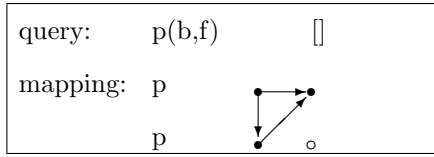
By composing the query-mapping pairs (i) on page 8 and (ii) on page 9 we get the query-mapping pair



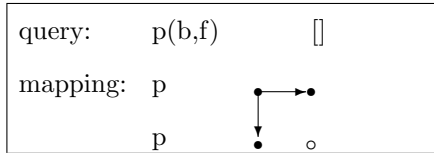
By composing the pair (iii) on page 9 with itself we get again the pair (iii), that is, it is idempotent.

Definition 2.11. A query-mapping pair that can be composed with itself, that is a pair in which the query is identical to the range of the mapping, is called circular.

The pair (iii) is circular and idempotent. The following pair is circular but not idempotent:



The result of composing it with itself is



3 The Basic Theorems

The following theorem holds:

Theorem 3.1. *If there is an infinite branch in the LD-tree corresponding to a program and query then there is an infinite sequence of nodes N_1, N_2, \dots such that for each i , N_{i+1} is an offspring of N_i .*

Proof. Straightforward (cf. [81]). □

Suppose we have some way to associate basic query-mapping pairs with call branches between nodes that are direct offspring of each other. Suppose that this is done in such a way that if N_2 is a direct offspring of N_1 and N_3 is a direct offspring of N_2 then the range of the mapping of the pair for N_1, N_2 is equal to the query of the pair for N_2, N_3 . Then we can use composition to associate a query-mapping pair with each call branch. Note that composition of query-mapping pairs is associative. Note also that given a program there can only be a finite number of query-mapping pairs associated with it.

The finiteness of the set of query-mapping pairs allows us to use the following version of Ramsey's theorem to prove our basic theorem (cf. [29]). We will use the following notation: if M is a subset of the natural numbers \mathcal{N} , we will denote by $M^{[n]}$ the set of all subsets of M of cardinality n .

Theorem 3.2 (Ramsey). *Let α be a mapping from $\mathcal{N}^{[n]}$ to some finite set A . Then there is an infinite subset M of \mathcal{N} such that α is constant on $M^{[n]}$.*

(Cf. [62, 39]. For a short self contained proof of this version of the theorem see [10], p. 290.)

Theorem 3.3 (Basic Theorem). *Suppose the LD-tree for a program and query has an infinite branch. Suppose a substitution-compatible partial mapping ϕ from terms to a strictly ordered well-founded set is given. Suppose that basic query-mapping pairs are assigned to nodes that are direct offspring of each other in such a way that if N_2 is a direct offspring of N_1 and N_3 is a direct offspring of N_2 then the range of the mapping of the pair for N_1, N_2 is equal to the query of*

the pair for N_2, N_3 . In this case a query-mapping pair can be assigned to each call path by composing the basic pairs along the path. Then there is a sequence of nodes M_1, M_2, \dots and a query-mapping pair (π, μ) so that for each i , M_{i+1} is an offspring of M_i , and for each j, k the query-mapping pair corresponding to the call branch from M_j to M_k is (π, μ) . The pair (π, μ) can be composed with itself and is idempotent.

Proof. By Theorem 3.1 there is an infinite sequence of nodes N_1, N_2, \dots such that for each i , N_{i+1} is an offspring of N_i . The set of query-mapping pairs that can be constructed with the predicate symbols of a program is finite. For each $i < j$ we get one query-mapping pair in this set. By using Ramsey's theorem for $n = 2$ we get that there is an infinite subsequence N_{i_1}, N_{i_2}, \dots and a unique query-mapping pair (π, μ) so that for each $k < l$ the pair assigned to the call branch from N_{i_k} to N_{i_l} is (π, μ) . Clearly this pair can be composed with itself and is idempotent. To see that it is idempotent, take for example the three nodes $N_{i_1}, N_{i_2}, N_{i_3}$. The pair (π, μ) corresponds to each of the three call paths from N_{i_1} to N_{i_2} , from N_{i_2} to N_{i_3} and from N_{i_1} to N_{i_3} . Since the pair corresponding to the path from N_{i_1} to N_{i_3} is the composition of the pairs corresponding to the paths from N_{i_1} to N_{i_2} and from N_{i_2} to N_{i_3} we get the idempotence. \square

From this theorem we get the following sufficient condition for termination which is formulated in terms of query-mapping pairs.

Theorem 3.4 (Sufficient Condition for Termination). *Suppose the LD-tree for a program and query and a substitution-compatible partial mapping ϕ from terms to a strictly ordered well-founded set \mathcal{S} are given. Consider a complete set of query-mapping pairs associated with the tree and the mapping ϕ (that is, pairs for all nodes that are direct offspring of each other and their compositions). If for every circular idempotent pair there is an arc from an argument in the domain to the corresponding argument in the range then the tree must be finite, i.e. there is termination for the query with Prolog's computation rule.*

Proof. From the previous theorem we get that if there is an infinite branch there must be an infinite sequence of nodes N_1, N_2, \dots such that the same circular idempotent pair corresponds to each pair of nodes N_i, N_{i+1} . Since every circular idempotent pair contains an arc from an argument in the domain to the corresponding argument in the range, this would imply the existence of an infinite descending sequence of ϕ values in contradiction to the assumption that \mathcal{S} is well-founded. \square

It turns out that we do not have to construct all query-mapping pairs. Obviously it is enough to consider only pairs that may participate in the creation of a circular pair.

Definition 3.1 (Predicate Dependency Graph). *The predicate dependency graph of a program is a graph whose nodes are the predicates of the program and which has, for every rule $A : - B_1, \dots, B_n$ in the program (remember that a rule is a clause with non-empty body) and every i , $1 \leq i \leq n$, an arc from the predicate of A to the predicate of B_i (cf. [58]).*

We can consider the strongly connected components of this graph. We call a strongly connected component **trivial** if it consists of a single node that has no arc going from itself to itself. It is easy to see that if the predicate dependency graph has no non-trivial strongly connected component, there can be no recursion in the program. Also the only pairs that can participate in the creation of a circular pair are those for which the predicate of the domain and the predicate of the range are in the same non-trivial strongly connected component.

Definition 3.2 (Recursive Query-Mapping Pair). *A query-mapping pair is called recursive if the predicates of the domain and range belong to the same strongly connected component of the predicate dependency graph.*

Theorem 3.5 (Optimization of Sufficient Condition for Termination). *Theorem 3.4 remains true if we only consider the recursive query-mapping pairs.*

If we use Theorem 3.5 instead of Theorem 3.4 there will in general be fewer pairs to consider so we will get an answer more quickly.

Example 3.1. Suppose a finite directed acyclic graph is given by a set of facts of the form $arc(X, Y)$ denoting an arc going from X to Y . We can use this graph to define a relation $X > Y$ if there is a path of arcs from X to Y . This relation is clearly transitive. Because of the acyclicity of the graph, it is also asymmetric. Hence, the relation we have defined is an order. Moreover it is well-founded because of the finiteness and acyclicity of the graph. Consider the program consisting of all the facts of the form $arc(-, -)$ and the clauses

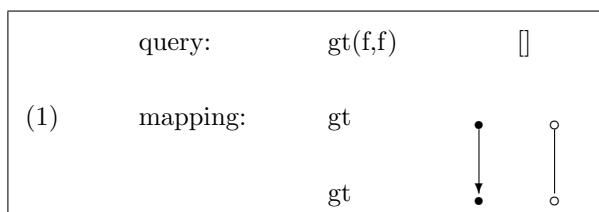
$gt(X, Y) :- arc(X, Y).$
 $gt(X, Y) :- arc(X, Z), gt(Z, Y).$

and the query pattern $gt(f, f)$.

If we resolve the goal $\leftarrow gt(X, Y)$ with the second rule we get

$$\leftarrow arc(X, Z), gt(Z, Y).$$

After one more step of LD-resolution we get $\leftarrow gt(z_0, Y)$, where $\{X \rightarrow x_0, Z \rightarrow z_0\}$ is a substitution that unifies $arc(X, Z)$ with one of the facts of the program. To construct query-mapping pairs we can take as ϕ the identity mapping on the nodes of the acyclic graph with the order defined above. We get the query-mapping pair



(Note, by the way, that in this case the query pattern and the domain are different.) Now we have a new query pattern, $gt(b, f)$, for which we get the pair

	query:	gt(b,f)	□
(2)	mapping:	gt	•
		gt	○
			↓
			○

These are the only recursive query-mapping pairs in this case (if we were considering all query-mapping pairs, not only the recursive ones, we would also have pairs with predicate gt in the domain and predicate arc in the range). The only circular pair is the second one, and it has an arc from the first argument in the domain to the first argument in the range, so we get that there is termination for the query pattern $gt(f, f)$. Note that this result is not obvious, since for graphs that are not acyclic we could get non-termination of queries matching the query pattern. \square

4 Query-Mapping Pairs Based on Linear Norms

This section explains the theoretical foundations for the implementation of the *TermiLog* system [74, 46]. Since our aim is automatic termination analysis, we need a uniform method to create query-mapping pairs associated with a query and program in order to test the sufficient condition of Theorems 3.4 and 3.5. What we need is a uniform way of ordering terms. In our case, the order on terms is defined by means of linear norms.

4.1 Linear Norms and Symbolic Linear Norms

For each ground term we define a *norm*, which is a non-negative integer; note that different terms may have the same norm.

Definition 4.1 (Linear Norms). *A linear norm of a ground term $f(T_1, \dots, T_n)$ is defined recursively as follows*

$$\|f(T_1, \dots, T_n)\| = c + \sum_{i=1}^n a_i \|T_i\|$$

where c and a_1, \dots, a_n are non-negative integers that depend only on f/n . Note that the definition also applies, as a special case, to constants (which are zero-arity function symbols).

Linear norms generalize earlier norms used in automatic termination analysis. In particular, the list size of [76] and the term size of [78] are special cases of linear norms. Plümer used in his work on termination two restricted cases of linear norms. One corresponds to the case where all the a_i are equal to 1 (in [58], it is called a “linear norm”) and the other corresponds to the case where each a_i is chosen to be either 0 or 1 (in [59], it is called a “semi-linear norm”).

Since the terms that appear in logic programs are very often nonground, we extend the definition of a linear norm to nonground terms by denoting the norm of a variable X by X itself. Thus, a nonground term has a **Symbolic Linear Norm**, which is a linear expression with non-negative coefficients. For example, if for a function symbol f of arity 3, the a_i and c are all equal to 1, then the symbolic norm of the term $f(X, Y, X)$ is $2X + Y + 1$. We will say that a linear expression is in *normalized form* if each variable appears once in it and also the free coefficient appears at most once. So $2X + 5Y + 2$ is in normalized form, while $X + X + 3Y + 2Y + 1 + 1$ is not.

The idea of associating an integer variable with a logic variable goes back to [78]. In [71] what we call *symbolic norm* for the case of term-size is called *structural term size*. Some authors define the norm of a variable to be 0 and then use the norm only for terms that are *rigid* with respect to it (cf. [59],[25]). In our context it is more convenient to use the symbolic norm. If the symbolic norm of a term is an integer then we know that the term is rigid—its norm is a constant and cannot change by different instantiations of its variables.

Definition 4.2 (Instantiated Enough). A (possibly nonground) term is *instantiated enough* with respect to some linear norm if its symbolic norm is an integer.

Instantiated-enough terms are essentially *rigid* terms in the terminology of [59,25]; that is, terms that cannot change their sizes due to further unifications.

For terms t that are instantiated enough we will take $\phi(t) = \|t\|$. Thus terms that are instantiated enough will be mapped into the well-founded set of the non-negative integers. In this context black nodes will correspond to arguments that are instantiated enough.

As an example for a linear norm, consider the **list-size norm** defined for list terms as

$$\|[H|T]\| = 1 + \|T\|$$

that is, $c = 1$, $a_1 = 0$ and $a_2 = 1$, while for all other functors the norm is 0. In this case, the norm is a positive integer exactly for lists that have a finite positive length, regardless of whether the elements of those lists are ground or not. Thus, all finite lists are instantiated enough with respect to the list-size norm.

Another example is the **term-size norm**. It is defined for a functor f of arity n by setting each a_i to 1 and c to n . According to the term-size norm, a term is instantiated enough only if it is ground.

In our experimentation we found that in most cases the term-size norm is sufficient for showing termination. In some cases the list-size norm is needed. There were only few cases in which a general linear norm was needed. In the version of *TermiLog* on the web [74] it is only possible to use the term-size or list-size norms, while in the full version there also is a possibility for the user to define an appropriate general norm. Note also that the set of queries described by a query pattern depends on the norm, so if we prove termination of $\text{append}(b, b, f)$ with the term-size norm this means that there is termination

for queries in which the first two arguments are ground, while termination with the list-size norm means that there is termination for queries whose first two arguments are lists of finite length that may contain variables.

4.2 The Weighted Rule Graph

Our first step is to construct from each rule of the program a graph, which extracts all the information about argument norms that is in the rule. This graph will be used in the construction of the query-mapping pairs. The graph will have nodes labeled by the terms that are the arguments of the atoms of the rule. For each term we will compute its symbolic linear norm, which is a linear expression in the variables of the term. As mentioned earlier we use the name of a logic variable to denote its norm. If nodes $N1$ and $N2$ are labeled by terms $T1$ and $T2$ and $\|T1\| = \|T2\|$ we will put in the graph an edge between the nodes. Otherwise we will compute the difference $\|T1\| - \|T2\|$. This difference is a linear expression. If this expression, when put into normalized form, has non-negative coefficients and a positive free coefficient (by "free coefficient" we mean the coefficient that does not precede a variable, say a_0 in $a_0 + a_1X + a_2Y$) this means that whenever the numeric variables will get non-negative norm values (when the respective logic variables become instantiated enough through appropriate substitutions) the expression will be a positive number. In this case we will put in the graph a *potential arc*, labeled by the normalized norm difference, from $N1$ to $N2$. We will draw potential arcs as dashed arcs.

It should be explained what potential arcs are. In the termination proof we use the fact that the order induced by the norm on terms that are instantiated enough is well-founded (recall that for such terms the norm is a non-negative integer). Once we know that the nodes connected by a potential arc are instantiated enough, we connect them with an arc. However, we will not do this when we do not know that the arguments are instantiated enough, because we want to be sure that there cannot be an infinite path consisting of arcs. Consider for example the program

```
int(0).
int(s(X)) :- int(X).
```

with the query $int(Y)$ and the term-size norm. From the rule we get the weighted rule graph

$$\begin{array}{ccc} \text{int} & & \text{s}(X) \\ & & \downarrow 1 \\ \text{int} & & X \end{array}$$

However, there is an infinite derivation

$$\begin{array}{l}
\leftarrow \text{int}(Y) \\
\quad \{Y \mapsto s(Y1)\} \\
\leftarrow \text{int}(Y1) \\
\quad \{Y1 \mapsto s(Y2)\} \\
\leftarrow \text{int}(Y2) \\
\quad \vdots
\end{array}$$

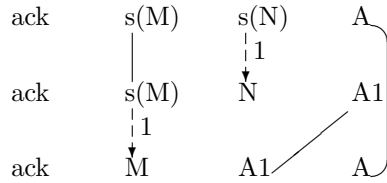
We now come to the formal definition:

Definition 4.3. *The weighted rule graph associated with a rule has as nodes all the argument positions of the atoms in the rule, each labeled by the term filling that argument position. Let $N1$ and $N2$ be any nodes in the graph, labeled by the terms $T1$ and $T2$, respectively. If $\|T1\| = \|T2\|$ then the graph will contain an edge between $N1$ and $N2$. If the normalized form of $\|T1\| - \|T2\|$ has non-negative coefficients and a positive free coefficient then the graph will contain a potential arc from $N1$ to $N2$ labeled by $\|T1\| - \|T2\|$.*

For example, using the term-size norm, we get for the rule

$$\text{ack}(s(M), s(N), A) \text{ :- } \text{ack}(s(M), N, A1), \text{ack}(M, A1, A).$$

the weighted rule graph that is shown in the figure. (Note that in order not to clutter the figure, we do not usually show edges and arcs that could be deduced from other edges and arcs.)



4.3 Generation of Basic Query-Mapping Pairs

To generate basic query-mapping pairs we'll use

- Results of the instantiation analysis (see Subsection 4.6).
- Results of the constraint inference (see Subsection 4.7).
- Weighted rule graphs for the rules of the program.

The instantiation analysis and constraint inference use abstract interpretation to give information about atoms that follow from the program.

Instantiation analysis tells us which instantiations we can expect in atoms that are logical consequences of a program. For instance for the Ackermann program with the term-size norm we get the instantiation patterns

$$\text{ack}(ie, ie, ie) \quad \text{ack}(ie, nie, nie)$$

where *ie* denotes an argument that is instantiated enough with respect to the norm and *nie* denotes an argument that is *not* instantiated enough. The result of the instantiation analysis is a set of instantiation patterns — atoms with a predicate that is a predicate of the program and arguments that are either *ie* or *nie*. Each atom that follows logically from the program is described by one of these instantiation patterns.

Constraint inference tells us which constraints we can expect in atoms that are logical consequences of a program. For instance for the Ackermann function we get the constraints

$$\begin{aligned} & \text{constraint}(\text{ack}/3, [\text{gt}(3, 1), \text{gt}(3, 2)]) \quad \text{constraint}(\text{ack}/3, [\text{gt}(1, 2), \text{gt}(3, 2)]) \\ & \qquad \qquad \qquad \text{constraint}(\text{ack}/3, [\text{gt}(3, 2)]) \end{aligned}$$

(The notation $\text{gt}(i, j)$ means that the i 'th argument is greater than the j 'th argument.) The constraint inference gives us for each predicate of the program such a disjunction of conjunctive constraints. A conjunctive constraint has the form

$$\text{constraint}(\text{pred}/ar, \text{list_of_argument_constraints})$$

where *pred* is a predicate of the program, *ar* is its arity, and the List Of Argument Constraints contains basic constraints of the form $\text{gt}(i, j)$ or $\text{eq}(i, j)$. A disjunctive constraint for a predicate is given by a list of conjunctive constraints for the predicate. Each atom that follows logically from the program satisfies the basic constraints of one of the conjuncts of the disjunctive constraint describing its predicate. Since any atom satisfies the empty list of basic constraints, performing the constraint inference is optional — if we can prove termination without it, it usually is faster. However, there are cases in which it is impossible to prove termination without it. For each example in the tables at the end of the paper one can see whether constraint inference was used in the termination proof by observing if there is a time given in the "Constr" column. One can see that there are more examples for which it was not used. However in a case like **quicksort** we need constraint inference for *partition* to deduce that in the clause

```
quicksort([X|Xs], Ys) :- partition(Xs, X, Littles, Bigs),
                        quicksort(Littles, Ls),
                        quicksort(Bigs, Bs),
                        append(Ls, [X|Bs], Ys).
```

the norms of the local *Littles* and *Bigs* are smaller than the norm of $[X|Xs]$.

In argument mappings we only have edges and arcs and black and white nodes. In the weighted rule graph there is more information — there are weighted arcs and there are labels for the nodes. We want to deduce argument mappings from weighted graphs augmented with information about instantiations and constraints. To do so we need the following.

Definition 4.4 (Zero-Weight and Positive-Weight Paths). *When traversing a path, an edge can be traversed in both directions and its weight is zero. An*

arc can be traversed only in the direction of the arrow. A weighted arc with a label w can be traversed in both directions; in the direction of the arrow its weight is w and in the opposite direction its weight is $-w$. A path has a positive weight if either

- there is at least one arc between adjacent nodes and the normalized expression for the sum of the weights has non-negative coefficients
- or
- there is no arc between adjacent nodes but the normalized expression for the sum of the weights has non-negative coefficients and a positive free coefficient.

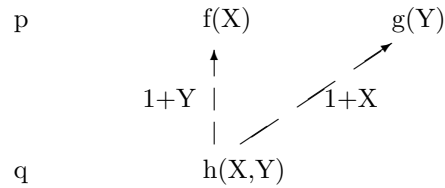
A path has zero weight if it only has edges and weighted arcs (but no arcs) and the sum of the weights along the path is zero.

Definition 4.5 (Inferred Edges and Arcs). Let G be an augmented weighted rule graph. We infer a new edge between nodes u and v if there is a zero-weight path between these nodes. We infer a new potential arc from node u to node v if there is a positive-weight path from u to v . This arc is a real arc if the terms labeling u and v are instantiated enough.

If we know from the query pattern or the instantiation analysis that a certain node of the weighted rule graph is black, i.e. its label is instantiated enough, we can compute the linear norm of the label and deduce that all variables appearing in the expression for the norm are ground. By propagating this information to other labels it may be possible to deduce that they are also instantiated enough. We call this process **Inference of Black Nodes**. For example, suppose we have the rule

$$p(f(X), g(Y)) \text{ :- } q(h(X, Y)).$$

and suppose we use the term-size norm. In this case the norms of the terms $f(X), g(Y), h(X, Y)$ are, respectively, $1 + X, 1 + Y, 2 + X + Y$. The weighted rule graph is:



Suppose a query $p(b, b)$ is given. Then we know that X and Y must be ground and can deduce that the argument of q must be ground too, i.e. the corresponding node is black.

Suppose a query pattern Q is given. Take a rule $r : H \text{ :- } S_1, \dots, S_n$ such that the predicate of H is identical to the predicate of Q . We will describe how to generate a query pattern corresponding to S_i and, only if the predicates of H and S_i are in the same strongly connected component of the predicate dependency

graph, a query-mapping pair corresponding to H and S_i ($1 \leq i \leq n$). The rule r means: to prove H (that is, find substitutions for its variables so that it follows logically from the program) you have to prove S_1, \dots, S_n . Since we use Prolog's computation rule we proceed from left to right. If we arrive at S_i this means that we have already proved S_1, \dots, S_{i-1} , so we can use for them the results of the instantiation analysis and constraint inference. Sometimes several choices for instantiations or constraints will be possible. We will pursue all of them. So one query may generate several queries and pairs depending on the choices for rule, subgoal and the instantiations and constraints for the subgoals preceding the chosen subgoal. For each new query generated we repeat the process applied to Q .

We now outline in detail the algorithm which creates a complete set of basic query-mapping pairs, so that for each pair of nodes that are direct offspring of each other in an LD-tree for the the given program and a query that matches the query pattern Q , there is a basic query-mapping pair in the set corresponding to it.

Put **BasicPairs**= \emptyset , **QueryQueue**= $[Q]$.

While **QueryQueue** $\neq \emptyset$ do

{ Remove a query pattern $Q1$ from **QueryQueue** and repeat for it the following two stage process for every possibility of program rule r , index i of body atom of the rule, possible instantiation pattern and possible conjunctive constraint:

STAGE 1. Augmenting the weighted rule graph:

1. Construct the weighted rule graph G of a rule $r : H : - S_1, \dots, S_n$ such that H has the same predicate as $Q1$.
2. Blacken the nodes of the head that are instantiated enough according to the query pattern, and add arcs and edges for the constraints that appear in the query pattern.
3. Propagate the information about black nodes to infer, if possible, further black nodes.
4. For each j , $1 \leq j < i$, choose an instantiation pattern, given by the instantiation analysis, that is compatible with the black nodes of S_j in the sense that if an argument in S_j is black then the corresponding argument in the instantiation pattern is *ie*. If a node of S_j is white and the corresponding argument in the instantiation pattern is *ie*, blacken that node and propagate the information.
5. In case constraint inference was performed, insert for each S_j , $1 \leq j < i$, edges and potential arcs in accordance with one of the disjuncts of the constraint inferred for its predicate. (Note that if it is possible to prove termination without using the constraint inference, it is usually preferable to avoid it.)
6. Turn all potential arcs or potential weighted arcs to arcs, respectively weighted arcs, if their endpoints are black, i.e. instantiated enough.
7. Add to G all the inferred edges and arcs between nodes of S_i .

8. If the predicate symbols of H and S_i are in the same strongly connected component of the predicate dependency graph, add to G all the inferred edges and arcs between nodes of H and S_i .

STAGE 2: Getting a new query pattern and possibly a new query-mapping pair from the augmented weighted rule graph:

1. Convert all weighted arcs to arcs by deleting their labels.
2. Delete all nodes except those corresponding to argument positions of H and S_i .
3. Delete labels of nodes, leaving only their being black or white.
4. Delete all edges and arcs except for edges that connect existing nodes and arcs that connect existing black nodes.
5. If the predicates of H and S_i are in the same strongly connected component of the predicate dependency graph put in **BasicPairs** a query-mapping pair for which the query is $Q1$ and the mapping is given by the nodes of H and S_i and the edges and arcs between them.
6. Generate a new query pattern with the argument positions of S_i and the edges and arcs between them. If this query pattern has not been investigated before, put it in **QueryQueue**.

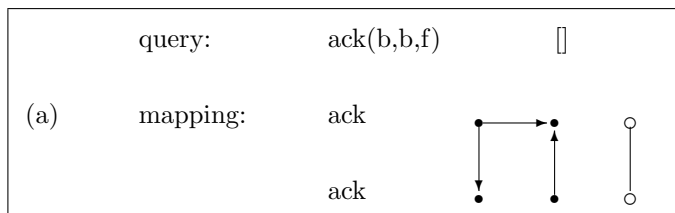
}

Note that this process must terminate, because there is only a finite number of query patterns that can be formed with the predicate symbols of the program.

Let us return to the Ackermann function program in section 2 and let us use the term-size norm. The query pattern corresponding to $\text{ack}(s(0), s(s(0)), A)$ is the pattern $\text{ack}(b, b, f) \ \square$ (note that \square denotes the empty constraint list). Using the weighted rule graph for the rule

(ii) $\text{ack}(s(M), 0, A) \text{ :- } \text{ack}(M, s(0), A).$

and inserting the information from the query pattern we get the query-mapping pair (a):



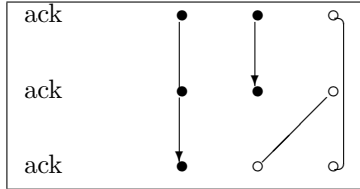
The 'new' query generated in this case is the same as the query we started with.

By the way, this is a circular idempotent pair which satisfies the condition of Theorem 3.4 — there is an arc between the first argument of the domain and the first argument of the range. If the condition had not been satisfied we could have halted — our method would not have been able to prove termination.

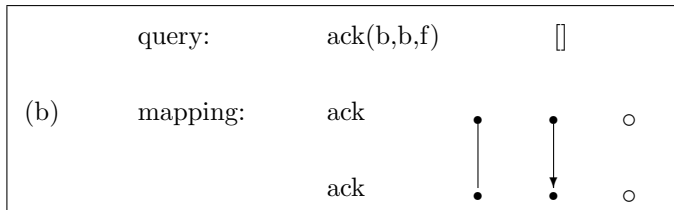
If we take the rule

(iii) $\text{ack}(s(M), s(N), A) :- \text{ack}(s(M), N, A1), \text{ack}(M, A1, A).$

and augment the weighted rule graph on p. 17 with the information from the query, we get:



For the first subgoal of the rule we get from this graph the query-mapping pair

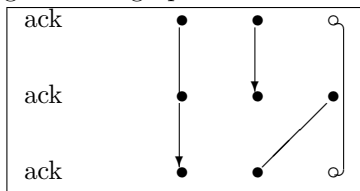


and no new query is generated.

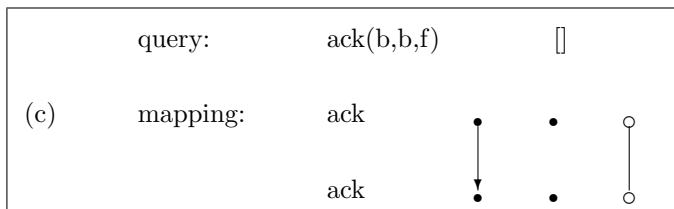
If we do not use the results of the instantiation analysis, we would get from the graph, for the second subgoal of the rule, a new query $\text{ack}(b, f, f)$. For this query we would not have been able to prove termination because it does not terminate. However, if we use the results of the instantiation analysis, we know that there are only two possible instantiation patterns for the predicate *ack* :

$$\text{ack}(ie, ie, ie) \qquad \text{ack}(ie, nie, nie)$$

The only pattern that is compatible with the middle row of our graph is the first one. Using this pattern and propagating the information we get the augmented weighted rule graph



From this graph we get for the second subgoal the old query and the query-mapping pair

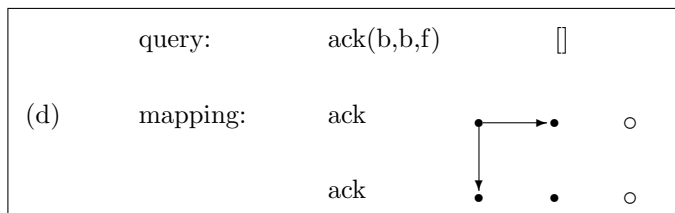


No more queries or basic query-mapping pairs can be generated from the original query pattern. Now we have to use composition in order to get all possible query mapping pairs.

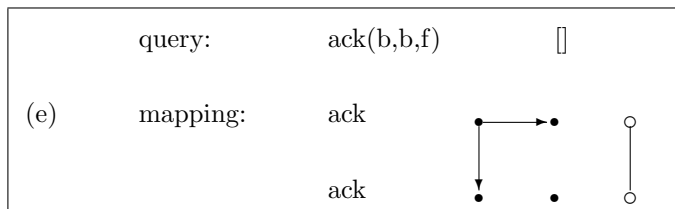
4.4 Creation of All Query-Mapping Pairs by Composition and the Termination Test

Now we have to *compose* the basic query-mapping pairs till no more pairs can be created. Since the number of query-mapping pairs that can be associated with a program is finite this process terminates. Whenever a circular idempotent pair is encountered we check that there is an arc from an argument in the domain to the corresponding argument in the range. If this is not so, we halt — our method cannot prove termination in this case. If all query-mapping pairs have been created, and every circular idempotent pair has an arc from an argument in the domain to the corresponding argument in the range, we know that there is termination.

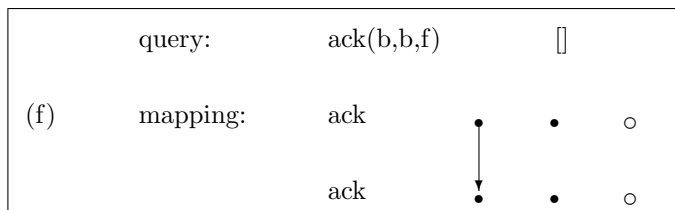
In our example composition of the pairs (a) and (b) gives a new pair (d):



Composing the pairs (a) and (c) gives the pair (e):



Composing the pairs (b) and (a) gives the pair (f):



No further pairs can be created by composition as the following 'multiplication table' of the pairs shows:

	a	b	c	d	e	f
a	a	d	e	d	e	d
b	f	b	f	f	f	f
c	c	f	c	f	c	f
d	d	d	d	d	d	d
e	e	d	e	d	e	d
f	f	f	f	f	f	f

In this case all the query-mapping pairs are circular and idempotent and satisfy the condition of Theorem 3.4, so we get that there is termination for the query pattern $ack(b, b, f)$.

It should be noted that by the method we outlined of generating basic query-mapping pairs and then composing them till no new pairs can be obtained, we can assign a pair to each call branch. This pair gives a not necessarily complete description of arguments that are instantiated enough and size relations between the norms of these arguments. The pair assigned to a call branch in this case depends not only on the endpoints of the branch (as was the case in the examples in Section 2), but also on the location of the branch in the tree. It is a sound approximation to the relation between the calls at the ends of the branch.

The number of query-mapping pairs that can be formed with the predicate symbols of a program is exponential in their arities, and there are cases in which the number of query-mapping pairs relevant to a program and query is large. The following result about complexity is relevant. In [41] size-change graphs are used to prove termination of functional programs. These graphs are the counterpart of our query-mapping pairs in the simpler context of functional programming, where all the problems connected with instantiation fall away. It is proved there that the analysis is PSPACE hard. However, if the maximal arity of a predicate, the maximal number of different variables in a clause and the maximal number of subgoals in a clause is limited by a relatively small number, we can expect our method to behave reasonably, as illustrated by our experiments.

4.5 Possible Optimizations

The optimization of Theorem 3.5, that only query-mapping pairs in which the predicates of the domain and range belong to the same strongly connected component of the predicate dependency graph need be considered, was implemented in the *TermiLog* system from the beginning.

Another optimization, implemented recently, is the following. Let us define

Definition 4.6 (Weaker Version). *Suppose that two query-mapping pairs P_1 and P_2 have identical queries and identical ranges of the mappings, and that every edge in P_1 is also included in P_2 and every arc in P_1 is also included in P_2 . Then we will say that P_1 is a weaker version of P_2 .*

Suppose that we discover that a query-mapping pair P_1 is a weaker version of a pair P_2 . Then the pair P_2 can be discarded, since in the termination proof the

weaker P_1 can be used in any place P_2 is used, and if termination can be proved, the edges and arcs of P_1 must be sufficient for the proof.

In the above example of the Ackermann function the basic query-mapping pairs are (a), (b) and (c). The pair (c) is a weaker version of the pair (a), so we need only use composition for the pairs (b) and (c). By composing (b) and (c) we get the pair (f). The pair (f) is a weaker version of the pair (c), so it is enough to consider all the compositions of (b) and (f). This does not give rise to further pairs. Since the pairs (b) and (f) satisfy the condition of Theorem 3.4 we get termination of the query pattern $ack(b, b, f)$.

4.6 Instantiation analysis

Instantiation analysis of a given program is a preliminary step for our algorithm. We prove termination by using norm inequalities between sufficiently instantiated terms. The method used for the instantiation analysis is abstract interpretation as outlined in [24].

Definition 4.7 (Galois Connection). *Given two partially ordered sets $P^b(\preceq^b)$ and $P^\sharp(\preceq^\sharp)$ a Galois connection between them is a pair of maps*

$$\begin{aligned}\alpha : P^b &\longrightarrow P^\sharp \\ \gamma : P^\sharp &\longrightarrow P^b\end{aligned}$$

such that

$$\forall p^b \in P^b : \forall p^\sharp \in P^\sharp : \alpha(p^b) \preceq^\sharp p^\sharp \iff p^b \preceq^b \gamma(p^\sharp)$$

Both for the instantiation analysis and the constraint inference we use the following Theorem from [24]:

Theorem 4.1 (Fixpoint Abstraction). *If $P^b(\preceq^b)$ and $P^\sharp(\preceq^\sharp)$ are complete lattices⁶, there is a Galois connection between P^b and P^\sharp given by α and γ , and F^b is a monotone map from P^b to P^b , then*

$$\alpha(\text{lfp}(F^b)) \preceq^\sharp \text{lfp}(\alpha \circ F^b \circ \gamma).$$

Groundness analysis has been handled by several authors (cf. for example [24], [20]). Here we extend the ideas used for groundness analysis to the more general case of being instantiated enough for an arbitrary symbolic norm we have chosen. An argument is instantiated enough if its symbolic norm is an integer.

The usual Herbrand base whose atoms are all ground is not useful in this case and we have to extend it. We define a semantics which is similar, although not identical, to the c-semantics of [34, 15]. As extended Herbrand base \mathcal{B} we take all atoms that can be built from the constant, function, and predicate symbols of the program P and variables from an infinite set $\{X1, X2, \dots\}$. The

⁶ A complete lattice is a partially ordered set such that every subset has a least upper bound and a greatest lower bound (cf. [48]).

difference from the s-semantics and c-semantics is that there equivalence classes of atoms modulo variance are taken, while we take atoms with variables from the infinite set. The redundancy does not disturb us, since we are interested in the abstraction, which is finite. We define the immediate consequence operator T_P for any $I \subseteq \mathcal{B}$ in the following way:

$$T_P(I) = \{A \in \mathcal{B} : A \leftarrow A_1, \dots, A_n \text{ is an instance of a clause in } P \\ \text{and } \{A_1, \dots, A_n\} \subseteq I\}.$$

The least fixed point of T_P consists exactly of those elements of \mathcal{B} that have an SLD-refutation with the identity substitution as computed answer. We shall call it $\mathcal{H}(P)$, the minimal extended Herbrand model for the program. Now consider the complete lattice P^b of all subsets of \mathcal{B} , with the inclusion order. We define a Galois connection between P^b and P^\sharp , the power set (i.e. the set of subsets) of the set of all atoms whose predicate symbol is one of the predicates of the program, and whose arguments are *ie*, representing an argument that is instantiated enough for its symbolic norm to be an integer, and *nie*, representing an argument which is not instantiated enough. (In the case of groundness analysis usually *g* for *ground* and *ng* for *non-ground* are used instead of our *ie* and *nie*.) We define

for a term T

$$\alpha(T) = ie \quad \text{if the symbolic norm of } T \text{ is an integer,} \\ \alpha(T) = nie \quad \text{otherwise,}$$

for predicate symbol p

$$\alpha(p(T_1, \dots, T_n)) = p(\alpha T_1, \dots, \alpha T_n),$$

and for a set of atoms S

$$\alpha(S) = \{\alpha(s) \mid s \in S\}.$$

For $p^\sharp \in P^\sharp$ we define

$$\gamma(p^\sharp) = \{A \in \mathcal{B} : \alpha(A) \in p^\sharp\},$$

that is, $\gamma(p^\sharp)$ consists of all the atoms in \mathcal{B} that have the instantiation patterns included in p^\sharp . These α and γ determine a Galois connection between P^b and P^\sharp .

T_P is a monotone map from P^b to P^b , hence we get from the Fixpoint Abstraction Theorem that

$$\alpha\left(\bigcup_{i=1}^{\infty} T_P^i(\emptyset)\right) \subseteq \bigcup_{i=1}^{\infty} (\alpha \circ T_P \circ \gamma)^i(\emptyset)$$

Since P^\sharp is finite, the right hand side can be computed by a finite number of steps.

Define $T^\sharp = \alpha \circ T_P \circ \gamma$.

Algorithm for the Computation of the least fixed point of T^\sharp :

For each clause in the program, supposing it contains n different variables, we create all 2^n instances of substituting *ie* and *nie* for each of them. For each

instance of a clause we then substitute for the arguments of the predicates *ie* or *nie* in accordance to whether or not they are instantiated enough for the norm (this can be decided by replacing the *nie*'s in the clause instance with a variable, and then computing norms—the result should be *ie* if the computed norm is an integer and *nie* otherwise). This gives us the clauses of a new program, that has only ground arguments, and hence its success set is finite and can be computed as the least fixed point of the immediate consequence operator. But this is exactly the least fixed point of T^\sharp , because the new program describes its action on the atoms of P^\sharp .

Consider for example the *append* program

Example 4.1.

```
append([], X, X).
append([H|X], Y, [H|Z]) :- append(X, Y, Z).
```

□

with the term-size norm. A term is instantiated enough with respect to this norm if and only if it is ground. Hence, we use in this example *g* and *ng* instead of *ie* and *nie*. From the clause

```
append([], Y, Y).
```

we get for the new program

```
append(g, g, g).
append(g, ng, ng).
```

In the clause

```
append([H|X], Y, [H|Z]) :- append(X, Y, Z).
```

we have to substitute all 2^4 possibilities of *g* and *ng* for its variables. For instance the substitution

$$H \mapsto ng, \quad X \mapsto g, \quad Y \mapsto g, \quad Z \mapsto g.$$

would give us

```
append([ng|g], g, [ng|g]) :- append(g, g, g).
```

which would give us the rule for T^\sharp

```
append(ng, g, ng) :- append(g, g, g).
```

We get

$$T^\sharp(\emptyset) = \{\text{append}(g, g, g), \text{append}(g, ng, ng)\}$$

$$T^{\sharp^2}(\emptyset) = \{\text{append}(g, g, g), \text{append}(g, ng, ng), \text{append}(ng, g, ng), \text{append}(ng, ng, ng)\}$$

$$T^{\sharp^3}(\emptyset) = T^{\sharp^2}(\emptyset) \quad \bigcup_{i=1}^{\infty} T^{\sharp^i}(\emptyset) = T^{\sharp^2}(\emptyset)$$

On the other hand if we used the list-size norm we would get, for example, from the above rule and the substitution

$$H \mapsto nie, X \mapsto ie, Y \mapsto ie, Z \mapsto ie.$$

the instance

`append([nie|ie],ie,[nie,ie]) :- append(ie,ie,ie).`

and hence, since for the list-size norm $\|[H|T]\| = 1 + \|T\|$, the rule

`append(ie,ie,ie) :- append(ie,ie,ie).`

In this case

$$T^\#(\emptyset) = \{\text{append}(ie, ie, ie), \text{append}(ie, nie, nie)\}$$

$$\bigcup_{i=1}^{\infty} T^{\#^i}(\emptyset) = T^{\#^2}(\emptyset) = T^\#(\emptyset)$$

It should be noted that in the conclusion of the Fixpoint Abstraction Theorem we have *inclusion*, so this process gives us a superset of instantiations that may occur. To give an example where the inclusion is proper consider the following example:

Example 4.2.

`p(X) :- g(X), h(X).`

`g(a).`

`h(b).`

□

with the term-size norm. In this case we get the instantiation $p(ie)$ although there is no solution for $p(X)$.

The way we will use the instantiation analysis is as follows. Suppose we use the term-size norm and are given a subgoal to the *append* program with certain bindings, say *append(b, b, f)*, where *b* denotes a ground argument and *f* denotes an argument for which we do not know if it is ground. Then we know from the instantiation analysis what bindings cannot result for the arguments of this subgoal if it terminates successfully. For instance in the above case, we can infer that the third argument will become ground, because the only instantiation pattern in which the first two arguments are ground is *append(g, g, g)*.

4.7 The inference of constraints

Inference of monotonicity constraints was treated in [16] and inference of inequality constraints was treated in [17]. Monotonicity constraints have the advantage that once an atom is given, there is only a finite number of possibilities for them, but they are weak. Inequality constraints, say something like $\|arg1\| > \|arg2\| + n$ where *n* is a number, give more information, but there

are infinitely many possibilities for them. The algorithm for constraint inference proposed here tries to get the best of both worlds—in the derivation step it uses quantitative norm computations, while its conclusions are formulated as monotonicity constraints. This enables us, for instance, to show termination of **quicksort**.

We again use abstract interpretation, only things are more complicated. $P^b = 2^{\mathcal{B}}$ and T_P are as before. The set \mathcal{C} of abstractions of elements of \mathcal{B} consists of mixed graphs, whose nodes are the argument positions of some predicate in the program and whose edges and arcs form a consistent set of constraints between the nodes. Such graphs can be denoted by a pair consisting of the predicate (with arity) and the list of edges and arcs. An edge connecting the i 'th and j 'th nodes will be given by $eq(i, j)$ if $i < j$ and by $eq(j, i)$ if $i > j$, and an arc going from the i 'th node to the j 'th node will be given by $gt(i, j)$ (remember that our arcs go from a node of larger norm to a node of smaller norm). We will call such graphs *conjunctive predicate constraints*.

Consider $2^{\mathcal{C}}$, whose elements are sets of conjunctive constraints. An element of $2^{\mathcal{C}}$ is interpreted as the *disjunction* of the conjunctive constraints of which it consists. For $c_1, c_2 \in 2^{\mathcal{C}}$ define an equivalence $c_1 \sim c_2$ iff the set of all ground atoms that satisfy a constraint in c_1 is equal to the set of all ground atoms that satisfy a constraint in c_2 . For instance

$$\{(p/2, [])\} \sim \{(p/2, [gt(2, 1)]), (p/2, [eq(1, 2)]), (p/2, [gt(1, 2)])\}$$

Let $P^\# = 2^{\mathcal{C}}/\sim$, that is the set of equivalence classes of elements of $2^{\mathcal{C}}$. For an element $c \in 2^{\mathcal{C}}$ we denote by c_\sim its equivalence class. As α we take the map that assigns to each atom in \mathcal{B} the unique element in \mathcal{C} that has the constraints that can be inferred for the atom, using the particular symbolic norm we have chosen. Given an atom $A \in \mathcal{B}$ we compute the symbolic norms of its arguments. If the predicate of A is p/n then $\alpha(A)$ will consist of the pair $(p/n, List)$, where List contains elements $eq(i, j)$ ($i < j$) if the symbolic norm of the i 'th argument of A equals the norm of its j 'th argument, and contains $gt(i, j)$ if the normalized form of the difference between the norms of the i 'th and j 'th arguments of A , which is a linear expression in the variables, has non-negative coefficients and the constant term is positive (say something like $1 + X$ or $2 + 6X + 5Y$). Note that we assign to each atom one conjunctive predicate constraint. For instance, with the term-size or list-size norms,

$$\alpha(append([], Y, Y)) = \{(append/3, [eq(2, 3)])\}$$

$$\alpha(p([H|X], X, Y, [H|Y], X)) = \{(p/5, [gt(1, 2), gt(4, 3), gt(1, 5), eq(2, 5)])\}$$

(We could have chosen another definition of α , which would have assigned to an atom in \mathcal{B} a set of several elements in \mathcal{C} .) We extend α to a map from $2^{\mathcal{B}}$ to $P^\#$ in the obvious way.

The order in P^b is the inclusion order. In $P^\#$ we define the order $p_1^\# \preceq^\# p_2^\#$ iff the set of all ground atoms that satisfy one of the constraints in $p_1^\#$ is included in the set of all ground atoms that satisfy one of the constraints in $p_2^\#$. For example

$$\{(p/3, [eq(1, 2), gt(3, 2), gt(3, 1)])\}_\sim \preceq^\# \{(p/3, [gt(3, 2)])\}_\sim$$

$$\{(p/3, [eq(1, 2)]), (p/3, [gt(2, 1)])\} \sim \preceq^\# \{(p/3, [])\} \sim$$

For $c \in \mathcal{C}$ we define

$$\gamma(c) = \{A \in \mathcal{B} : \{\alpha(A)\} \preceq^\# c\}$$

and extend the definition to $P^\#$ in the obvious way. It is easy to see that we get a Galois connection and hence, by the Fixpoint Abstraction Theorem, if we define $T^\# = \alpha \circ T_P \circ \gamma$,

$$\alpha(lfp(T_P)) \preceq^\# lfp(T^\#)$$

Now we will define an operator

$$\tau : 2^{\mathcal{C}} / \sim \longrightarrow 2^{\mathcal{C}} / \sim$$

that approximates $T^\#$ from above in the sense that for each $I \in P^\#$ we have $T^\#(I) \preceq^\# \tau(I)$.

Definition 4.8 (Inferred Head Constraint). *Given $I \in 2^{\mathcal{C}}$ and a weighted rule graph we insert for the nodes of each body atom in the weighted rule graph the edges and (potential) arcs of one relevant conjunctive constraint from I . With the help of the weighted rule graph we infer the edges and (potential) arcs for the nodes of the head atom. The resulting conjunctive constraint is an inferred head constraint.*

We define

$$\tau(\emptyset) = \{\alpha(F) : F \text{ is a fact of the program}\}$$

For a non-empty set $I \in 2^{\mathcal{C}}$ (actually we are dealing with equivalence classes) we define

$$\tau(I) = I \cup \{c : c \text{ is an inferred head constraint relative to } I$$

and the weighted rule graph of some program rule\}.

Since τ is monotone and \mathcal{C} is finite the least fixed point of τ exists and the computation always terminates.

Algorithm for the inference of constraints—computation of the least fixed point of τ :

$$Old_Constraints = \emptyset$$

$$New_Generation_Constraints = \{\alpha(F) : F \text{ is a fact of the program}\}$$

While $New_Generation_Constraints \neq \emptyset$ do

1. Let *Derived* be all those constraints that can be inferred by taking the weighted rule graph for some program rule and inserting constraints for the subgoals according to elements of

$$Old_Constraints \text{ and } New_Generation_Constraints,$$

taking care that at least one constraint from the latter is used.

2. $Old_Constraints = Old_Constraints \cup New_Generation_Constraints$

3. *New_Generation_Constraints = Derived - Old_Constraints*

Return *Old_Constraints*.

For each $I \in P^\sharp$ we have $T^\sharp(I) \preceq^\sharp \tau(I)$, because by our methods we may not be inferring all the constraints and hence, because the least fixed points exist,

$$lfp(T^\sharp) \preceq^\sharp lfp(\tau)$$

To give an example where $T^\sharp(I)$ differs from $\tau(I)$ take the *append* program (Example 4.1).

$$\begin{aligned} \tau(\emptyset) &= \{\text{append}/3, [eq(2, 3)]\} \\ T^\sharp(\emptyset) &= \{\text{append}/3, [eq(1, 2), eq(2, 3), eq(1, 3)], \\ &\quad (\text{append}/3, [gt(2, 1), gt(3, 1), eq(2, 3)])\} \end{aligned}$$

Using the Fixpoint Abstraction Theorem we get

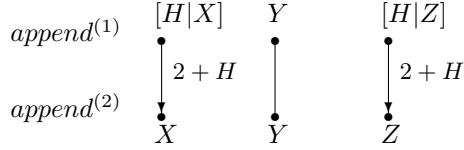
$$\alpha(lfp(T_P)) \preceq^\sharp lfp(T^\sharp) \preceq^\sharp lfp(\tau)$$

This means that every atom in the success set of the program satisfies at least one of the conjunctive predicate constraints in $lfp(\tau)$.

Consider for example the *append* program (Example 4.1). Then

$$\tau(\emptyset) = \{\text{append}/3, [eq(2, 3)]\}$$

The weighted rule graph for the second clause of the program is



Putting for the body nodes (the ones with $\text{append}^{(2)}$) the one constraint we got thus far we get the constraint $(\text{append}/3, [gt(3, 2)])$, so we get

$$\tau^2(\emptyset) = \{(\text{append}/3, [eq(2, 3)]), (\text{append}/3, [gt(3, 2)])\}$$

Applying τ to this new set we realize that no new constraint can be derived, so we have found a fixed point for τ . So we have inferred a disjunctive constraint for *append* consisting of two conjunctive constraints.

We found it advantageous to keep track of whether a constraint is obtained just once during the inference or more than once. Consider for example the program **mergesort**.

```
mergesort([], []).
mergesort([X], [X]).
mergesort([X, Y|Xs], Ys) :- split([X, Y|Xs], X1s, X2s),
    mergesort(X1s, Y1s), mergesort(X2s, Y2s), merge(Y1s, Y2s, Ys).
```

```

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

```

```

merge([], Xs, Xs).
merge(Xs, [], Xs).
merge([X|Xs], [Y|Ys], [X|Zs]) :- X=<Y, merge(Xs, [Y|Ys], Zs).
merge([X|Xs], [Y|Ys], [Y|Zs]) :- X>Y, merge([X|Xs], Ys, Zs).

```

If we try to infer constraints for the predicate *split* we get after the first application of T^{\sharp} to \emptyset the constraint $[eq(2, 1), eq(3, 1)]$, after a second application the constraint $[eq(2, 1), gt(1, 3)]$, and for all further applications only the constraint $[gt(1, 2), gt(1, 3)]$. The disjunction of these three constraints is not sufficient for inferring the termination of **mergesort** by the query-mapping pairs method, while the last constraint by itself is. In this case we can separate by unfolding the predicate **split** into three predicates and obtain the program **mergesort1**,

```

mergesort([], []).
mergesort([X], [X]).
mergesort([X, Y|Xs], Ys) :- split2([X, Y|Xs], X1s, X2s),
    mergesort(X1s, Y1s), mergesort(X2s, Y2s), merge(Y1s, Y2s, Ys).

```

```

split(Xs, Ys, Zs) :- split0(Xs, Ys, Zs).
split(Xs, Ys, Zs) :- split1(Xs, Ys, Zs).
split(Xs, Ys, Zs) :- split2(Xs, Ys, Zs).

```

```

split0([], [], []).
split1([X], [X], []).
split2([X, Y|Xs], [X|Ys], [Y|Zs]) :- split(Xs, Ys, Zs).

```

```

merge([], Xs, Xs).
merge(Xs, [], Xs).
merge([X|Xs], [Y|s], [X|Zs]) :- X=<Y, merge(Xs, [Y|Ys], Zs).
merge([X|Xs], [Y|Ys], [Y|Zs]) :- X>Y, merge([X|Xs], Ys, Zs).

```

for which we can prove termination by the query-mapping pairs method. More details on this unfolding-based technique can be found in [47].

5 The Implementation and Experimental Evaluation

5.1 The Implementation

The *TermiLog* system [46]), is based on the approach outlined in this paper. The version available on the web ([74]) can use the term-size norm and the list-size norm. In the full version (that is an off-line version available on request) it is also possible for the user to define other linear norms. Moreover there is a possibility to handle programs that use modules. First the module is analyzed

and the results are put in a file, which is then used when the program calling the module is analyzed. For handling larger programs there is an option of using a 'big version', that infers the subqueries that will be created from the original query by using only the results of the instantiation analysis, and then checking only those subqueries that have a recursive predicate.

Predefined predicates can be handled if their instantiation patterns are supplied to the system (recall that the instantiation patterns depend on the norm). In the current implementation, the instantiation patterns of most predefined predicates are already included in the system. Constraints of predefined predicates may also be included, but this is not necessary. Operator declarations that appear in a given program are asserted as facts of the system.

Control predicates have to be dealt with in a special way, since they are not part of the declarative semantics of logic programs. Cuts are simply ignored. Of course, if the semantics of cut is needed to show termination, then our system will not be able to determine that the given program terminates.

Other control predicates are handled by transforming the given program into a new program that does not have these control predicates, such that if termination can be shown for the new program, then the original program also terminates.

If a negated subgoal appears in a clause, say

```
A :- B, C, \+D, E,F.
```

then the above clause is replaced with the following two clauses:

```
A :- B, C, D.  
A :- B, C, E, F.
```

If several negations appear in the same clause, they can be handled by repeated application of the above transformation.

There is one point here that should be taken into account. The clause

```
A :- B, C, D
```

should not be used in the instantiation analysis and constraint inference, since for A to succeed D should fail, that is D cannot provide any bindings.

Disjunction, "if-then" and "if-then-else" are handled in an analogous way.

We will mention just one little example illustrating the usefulness of the system.

Take the following predicate *sublist*, whose intended function is to find whether one list is a sublist of another:

```
sublist(X,Y) :- append(X1,X,X2), append(X2,X3,Y).  
append([],X,X).  
append([H|X],Y,[H|Z]) :- append(X,Y,Z).
```

This is probably the most natural way to express the *sublist* relation (cf. [77]). However, if the query pattern *sublist(b,b)* is given, our system will say that there

may be non-termination because the first *append* gets the binding *append(f, b, f)*, and indeed a query like *sublist([1], [2, 3])* does not terminate.

If we switch the subgoals (as it is done in [73])

```
sublist(X,Y) :- append(X2,X3,Y), append(X1,X,X2).
```

our system shows that the query *sublist(b, b)* terminates.

This example shows how a user who writes a program according to the logic of a problem without thinking about the execution mechanism of Prolog can benefit from our system. This is especially true for predicates that succeed on the first call but go into an infinite loop when backtracked into. Switching the order of the subgoals does not change the logic of the program, but it may, as it does in this example, improve the termination behaviour.

5.2 Comparison with Other Automatic Approaches to Termination Analysis

When our system was developed, the CLP(R) package of SICStus Prolog [68] was not yet available, so the constraints we inferred, within the framework of standard Prolog, were only equality or monotonicity relations between arguments (this is in contrast with later systems like *TerminWeb* and *cTI*, that use CLP(R)). Therefore our system cannot handle the following program from [58]:

```
perm([], []).
perm(L, [H|T]) :- append(V, [H|U], L),
                  append(V, U, W),
                  perm(W, T).
append([], Y, Y).
append([H|X], Y, [H|Z]) :- append(X, Y, Z).
```

The query *perm(b, f)* terminates for this program, but our system cannot show termination. To prove termination we need the fact that for *append* the sum of the term-sizes of the first and second arguments equals the term-size of the third.

We can transform the definition of *append* to a form in which our system, which only infers term-size equality for arguments, will be able to infer the above linear equality:

```
perm([], []).
perm(L, [H|T]) :- append1(p(V, [H|U]), s(s(L))),
                  append1(p(V, U), s(s(W))),
                  perm(W, T).
append1(p([], Y), s(s(Y))).
append1(p([H|X], Y), s(s([H|Z]))) :- append1(p(X, Y), s(s(Z))).
```

(The functors *p* and *s* are used so that, for atoms with predicate *append1* in the success set, the norms of the two arguments will be equal.) For the transformed

program, which is clearly equivalent to the original one, our system easily proves termination of $perm(b, f)$.

In [9] argument size relationships are inferred with CLP(R). In *TerminWeb*'s old version (cf. [23]) ideas similar to our query-mapping pairs method are augmented with the analysis of [9], so termination of the example can be proved. It also can be proved with *cTI*.

cTI is a bottom-up constraint-based inference tool for Prolog (cf. [51, 50]). This tool is goal independent — it infers sufficient universal left-termination conditions for all the predicates in the given program (the recent implementation of *TerminWeb* does so too — cf. [37]). The new version of *cTI*, that uses the Parma Polyhedra Library (cf. [7]), is very efficient. However, it cannot prove termination for the program *vangelder* with query $q(b, b)$ (see Table 4), which *TermiLog* can do. *cTI* uses the term-size norm only (but as we have seen this is the most useful of the linear norms).

In contrast with the termination inference systems, *TermiLog* is goal directed, and in case it suspects non-termination it produces to the user the circular idempotent query-mapping pair that did not satisfy the termination test, thus showing him which predicate he should suspect.

The constraint based approach [27] is implemented but not publicly available. It starts with a general level mapping and general linear norm and infers the coefficients. This is efficient when it can be done but may run into trouble when there are nested expressions (because then we get products of the coefficients). It cannot handle the program of Ackermann's function (that *TermiLog* can handle) because in that case we need argument-by-argument comparisons instead of weighted sums.

The following two termination analyzers fall in a slightly different category, as both of them impose requirements on the logic programs handled.

TALP ([55]) is a publicly available tool that proves termination of logic programs by transforming them into term-rewriting systems. It requires the program and query to be well-moded. This requirement follows from the fact that term-rewriting systems have a clear distinction between input and output. This strongly differs from logic programs, where the same predicate can be used in different modes.

The compiler of the Mercury programming language contains a termination checker. This is described in [72] and its times are compared to *TermiLog*'s times for the benchmarks in our tables. The Mercury termination checker is usually faster than *TermiLog*, but one must remember that in Mercury the text of the program being checked contains mode informations as part of the language requirements. Another termination checker for Mercury is described in [35].

5.3 Experimental Evaluation

The technique presented here was implemented in the *TermiLog* system [46]. The system has been implemented in SICStus Prolog [68].

The 1996 version of the system has been applied to well over 100 logic programs, some quite long. The detailed experimental results can be found in [43,

45]. It should be noted that the times for the system as it was written in 1996 are now an order of magnitude faster because of improvements in computers and in SICStus Prolog.

We now improved the efficiency of the system in two ways:

1. Instead of checking for the presence of a forward positive cycle in circular pairs (as was done in the old version) we now only check for an arc between corresponding arguments in circular idempotent pairs. It turns out that this not only adds to the efficiency of the system but also to its power (cf. [29]).
2. We implemented the optimization of Section 4.5.

We tested the improved system on the benchmarks we had and the results are reported here. First, classical benchmarks for termination of symbolic computations were studied. Tables 1 and 2 summarise the performance of the system on the programming examples of [5, 30], and [58]. Next (Table 3), we applied our technique to study termination of benchmarks that were originally used for study of parallelism in logic programming [18, 36, 53, 19]. Benchmarks in this collection go back to Ramkumar and Kalé [61] (*occur*), Santos-Costa et al. [67] (*zebra*), Tick [75] (*bid*) and Warren (*palin*, *qplan*, *warplan*). A complete description of this collection may be found in [53]. Finally (Table 4), we have collected a number of programs from different sources, including Prolog textbooks [1, 73] and research papers [2, 4, 77, 83, 82]. Termination of most of the examples considered, with clearly indicated exceptions such as *serialize*, was established by using the term-size norm. Note also that there are cases in which we can establish termination of a query pattern both with the term-size norm and the list-size norm. Since being instantiated enough depends on the norm, the kind of queries corresponding to a query pattern will depend in these cases on the norm.

Note that all the benchmarks referred to in the tables are available, in compressed and uncompressed form, on the homepage [42].

In the tables the following abbreviations are used:

- *Ref* denotes a reference to the paper from which the program is taken.
- *F* and *R* denote, respectively, the numbers of facts and rules in the program.
- *Inst*, *Constr* and *Prs* denote, respectively, the times in seconds for the instantiation analysis, constraint inference and construction of pairs. Since constraint inference is an optional step the corresponding column is often empty, meaning that it has not been applied. Observe that for some examples the time needed to perform the corresponding step of the analysis was too small to be measured exactly. These cases are indicated by 0.00.
- *Pr#* is the number of query-mapping pairs constructed.
- *A* is the answer given by the system. It is either *T*, meaning that queries matching the query pattern terminate, or *N*, meaning that there *may* be queries having this pattern that don't terminate. For *N* we add an indication if there really is non-termination, denoted by *N+*, or if there is termination and the system is not strong enough to see it, denoted by *N-*.
- *Rem* means a remark. For remarks we use the following abbreviations:
 - *...mod* means the module feature was used for the named file.

- *after 4.7 transf* in the *mergesort* example means that the transformation outlined at the end of Subsection 4.7 was used.
- *mem* means that there are memory problems when trying to handle the program.
- *big* means that we used the version for big programs. If the big version has been used, measurements in the *Constr*-column present time spent on subquery generation and constraint inference.
- *no rec* means that there is no recursion in the program. Termination can be, therefore, established trivially.
- *cannot* means that it is clear our methods cannot handle the program. For instance, this is the case if the benchmark reads input and, hence, its termination depends on the presence of the end-of-file. Another case is programs that include *assert*.
- *succ* means transformation of integers to successor notation was applied.
- *=* means that the equalities elimination transformation was used. This transformation performs the unifications given by equalities like $X = Expression$ and thus reduces the number of variables. The *zebra* example shows how useful it can be. In this example the number of variables in the first clause is reduced from 25 to 15, thus speeding up the analysis very much. However, the transformation is not safe, as the following example shows:

```
p(X) :- loop(X), X=a.
loop(b) :- loop(b).
```

Here $p(X)$ does not terminate, while after the transformation it does. In the full system the user can apply this transformation on his own responsibility.

Tests were performed on Intel®Pentium®4 with 1.60GHz CPU and 260Mb memory, running 2.4.20-pre11 Linux, using SICStus Prolog Version 3.10.0.

6 Related Work and Conclusion

In the context of logic languages the ability to program declaratively increases the danger of non-termination. Therefore, termination analysis received considerable attention in logic programming. In our work we have considered universal termination of logic programs with respect to the left-to-right selection rule of Prolog. Early works on termination made no assumptions on the selection rule, that is, required termination with respect to all possible selection rules [11, 2, 12]. However, this notion of termination turned out to be very restrictive—the majority of real-world programs turn out to be non-terminating with respect to it. Thus, most of the authors studied termination with respect to some subset of selection rules. The most popular selection rule is left-to-right, as adopted by most of the Prolog implementations. Termination with respect to non-standard selection rules was considered, for instance, in [3, 38, 57, 65, 69, 70].

Roughly, the existing work on termination analysis proceeded along three important lines: providing necessary and sufficient conditions of termination [27,

14], providing sufficient (but not necessary) conditions for termination that can be verified automatically [37, 41, 52] and proving decidability or undecidability results for special classes of programs [13, 33, 64]. Our work is clearly situated in the second group: the condition presented in Theorems 3.4 and 3.5 implies termination and can be verified automatically.

While considering sufficient conditions for termination found in the literature one can distinguish between *transformational* [63, 6, 40, 55] and *direct* approaches [23, 27, 52]. A transformational approach first transforms the logic program into an “equivalent” term-rewrite system (or, in some cases, into an equivalent functional program). Here, equivalence means that, at the very least, the termination of the term-rewrite system should imply the termination of the logic program, for some predefined collection of queries. The approach of Arts [6] is exceptional in the sense that the termination of the logic program is concluded from a weaker property of *single-redex normalisation* of the term-rewrite system. Direct approaches, including our work, do not include such a transformation, but prove the termination directly on the basis of the logic program. Unlike the transformational approaches they usually do not put restrictions on the programs. Another advantage of the direct approaches is that the termination proof can be presented to the user in terms of the original program. In the case of the transformational approach the user does not necessarily understand the language of the transformed object.

The direct approaches can be classified as *local* and *global*. For local approaches termination is implied by the fact that for each loop there exists a decreasing function (cf. [29, 22]). Global approaches require the existence of a function that decreases along all possible loops in the program (cf. [27]). Correctness of the local approaches is based on Ramsey’s Theorem. Our approach is clearly local. This also means that *TermiLog* can be used not only to prove termination but also to provide a user with the reason why non-termination is suspected.

The query-mapping pairs approach originated in the algorithm of [66]. The original algorithm of [66] was based on an abstraction of a logic program as a datalog program with relations that could be infinite (this type of abstraction was proposed in [60]). The problem with this type of abstraction is that it loses too much valuable information about the original logic program and, in particular, one has to assume that every variable in the head of a rule also appears in the body. In the present approach logic programs are handled directly, so all the information incorporated in them can be used. There are no restrictions whatsoever on the logic programs considered. Moreover, the termination condition in [66, 43, 44], which was formulated in terms of circular variants with positive forward cycle, is replaced here, with the help of Ramsey’s Theorem, by a much simpler condition which gives a stronger termination theorem (cf. [29]).

As far as the power of the approach is concerned one has to remember that termination is undecidable, so one cannot expect too much. It is rather surprising for how many programs the system is applicable. An interesting fact that emerges from the experimentation is that in most cases the use of the the term-size norm

suffices and it is not necessary to use more sophisticated norms. The PSPACE hardness result of [41] applies to *TermiLog*'s analysis. Going over the different parts of the system one can see that if the maximal arity of a predicate, the maximal number of different variables in a clause and the maximal number of subgoals in a clause is limited by a relatively small number, one can expect our method to behave reasonably, as illustrated by the experiments. There are cases in which a linear norm is not sufficient for proving termination. For every linear norm a ground term is instantiated enough. In [29] an example is given of a program such that queries of the form $d(\textit{ground}, \textit{free})$ terminate, but this cannot be proved by any linear norm. There are cases, where the differentiation between arguments that are instantiated enough and those that are not, is not enough. We can use the query-mapping pairs as before with the only difference that we will abstract nodes not to just black and white ones but to a larger, though finite, set. For instance, if we have a program

$$\begin{array}{l} p(1) \text{ :- } p(1). \\ p(0) \text{ . } \quad p(2). \end{array}$$

and take the term-size norm and a query $p(b)$, the query-mapping pair algorithm will say that there may be non-termination. However, we can use the abstractions $1, g, f$, where g means any ground term that is not 1 and f means any term, and apply the above algorithm, with the only difference being in the unification of the abstractions (both when applying the instantiation pattern of the query and when composing query-mapping pairs). In the present case g and 1 will not unify, so we will be able to prove that $p(g)$ terminates.

Acknowledgements

We are grateful to the referees for their careful reading.

References

1. K. R. Apt. *From Logic Programming to Prolog*. Prentice Hall, 1997.
2. K. R. Apt and M. Bezem. Acyclic Programs. *New Generation Computing*, 9:335-363, 1991.
3. K. R. Apt and I. Luitjes. Verification of logic programs with delay declarations. In V. S. Alagar and M. Nivat, editors, *Algebraic Methodology and software Technology, 4th International Conference, 1995, Lecture Notes in Computer Science*, Vol. 936, 66-90, Springer Verlag, 1995.
4. K. R. Apt and D. Pedreschi. Reasoning about Termination of Pure Prolog Programs. *Information and Computation*, 106:109-157, 1993.
5. K. R. Apt and D. Pedreschi. Modular Termination Proofs for Logic and Pure Prolog Programs. In *Advances in Logic Programming Theory*, 183-229, Oxford University Press, 1994.
6. T. Arts. *Automatically proving termination and innermost normalisation of term rewriting systems*. PhD thesis, Universiteit Utrecht, 1997.

7. R. Bagnara, E. Ricci, E. Zaffanella and P. M. Hill. Possibly not closed convex polyhedra and the Parma Polyhedra Library. In *Static Analysis: Proceedings of the 9th International Symposium*, M. V. Hermenegildo and G. Puebla, eds., LNCS, Vol. 2477, 213–229, Springer Verlag, 2002.
8. M. Baudinet. Proving termination properties of Prolog programs: a semantic approach. *Journal of Logic Programming*, 14:1-29, 1992.
9. F. Benoy and A. King. Inferring argument size relationships with CLPR(R). *International Workshop on Logic Program Synthesis and Transformation (LOPSTR'96)*, 1996, 204-223.
10. Y. Benyamini and J. Lindenstrauss. *Geometric Nonlinear Functional Analysis, Vol. 1*. AMS Colloquium Publications Vol. 48. Providence, Rhode Island, 2000.
11. M. Bezem. Characterizing termination of logic programs with level mappings. In E. L. Lusk and R.A. Overbeek, editors, *Logic Programming, Proceedings of the North American Conference 1989*, 69-80, MIT Press, 1989.
12. M. Bezem. Strong termination of logic programs. *J. Logic Programming*, 15:79-97, 1993.
13. E. Börger. Unsolvable decision problems for Prolog programs. In E. Börger, ed., *Computation Theory and Logic, Lecture Notes in Computer Science*, Vol. 270, 37-48, Springer Verlag, 1987.
14. A. Bossi, N. Cocco, S. Etalle and S. Rossi. On modular termination proofs of general logic programs. *Theory and Practice of Logic Programming*, 2(3):263-291, 2002.
15. A. Bossi, M. Gabrielli, G. Levi, M. Martelli. The s-semantics approach: theory and Applications. *J. Logic Programming*, 19/20:149-198, 1994.
16. A. Brodsky and Y. Sagiv. Inference of monotonicity constraints in Datalog programs. *Proceedings of the Eighth ACM SIGACT-SIGART-SIGMOD Symposium on Principles of Database Systems*, 1989, 190-199.
17. A. Brodsky and Y. Sagiv. Inference of inequality constraints in logic programs. *Proceedings of the Tenth ACM SIGACT-SIGART-SIGMOD Symposium on Principles of Database Systems*, 1991, 227-240.
18. F. Bueno, M. García de la Banda and M. Hermenegildo. Effectiveness of Global Analysis in Strict Independence-Based Automatic Program Parallelization. *International Symposium on Logic Programming*, 320-336. MIT Press, 1994.
19. M. Codish, M. Bruynooghe, M. J. García de la Banda, and M. V. Hermenegildo. Exploiting Goal Independence in the Analysis of Logic Programs. *Journal of Logic Programming*, 32(3):247–262, 1997.
20. M. Codish and B. Demoen. Analyzing Logic Programs using “Prop”-ositional Logic Programs and a Magic Wand. *Proceedings International Logic Programming Symposium*, Vancouver, 1993.
21. M. Codish and B. Demoen. Collection of benchmarks.
22. M. Codish, S. Genaim, M. Bruynooghe, J. Gallagher and W. Vanhoof. One Loop at a Time. *6th International Workshop on Termination*, 2003.
23. M. Codish and C. Taboch. A semantic basis for termination analysis of logic programs. *Journal of Logic Programming* 41, 1, 103-123.
24. P. Cousot and R. Cousot. Abstract interpretation and application to logic programs. *J. Logic Programming*, 13:103-179, 1992.
25. S. Decorte and D. De Schreye. Automatic Inference of Norms: a Missing Link in Automatic Termination Analysis. *Logic Programming: Proceedings of the 1993 International Symposium*, ed. D. Miller. MIT Press, 1993.

26. S. Decorte and D. De Schreye. Demand-driven and constraint-based automatic left-termination analysis for Logic Programs. *Proceedings of the 1997 International Conference on Logic Programming*. MIT Press, 1997.
27. D. Decorte, D. De Schreye and H. Vandecasteele. Constraint-based termination analysis of logic programs. *ACM TOPLAS*, 21(6):1137-1195, 1999.
28. N. Dershowitz. Orderings for term-rewriting systems. *Theoretical Computer Science*, 17:279-301, 1982.
29. N. Dershowitz, N. Lindenstrauss, Y. Sagiv and A. Serebrenik. A general framework for automatic termination analysis of logic programs. *Applicable Algebra in Engineering, Communication and Computing*, 12, 1-2, 2001.
30. D. De Schreye and S. Decorte. Termination of Logic Programs: the Never-Ending Story. *J. Logic Programming*, 19/20:199-260, 1994.
31. D. De Schreye and A. Serebrenik. Acceptability with general orderings. *Computational Logic. Logic Programming and Beyond. Essays in Honour of Robert A. Kowalski, Part I. Lecture Notes in Computer Science*, Vol. 2407, 187-210, Springer Verlag, 2002.
32. D. De Schreye, K. Verschaeetse and M. Bruynooghe. A framework for analyzing the termination of definite logic programs with respect to call patterns. *Proc. of the Int. Conf. on Fifth Generation Computer Systems*, 481-488. IOS Press, 1992.
33. P. Devienne, P. Lebègue and J. C. Routier. Halting problem of one binary Horn clause is undecidable. STACS 93, *Lecture Notes in Computer Science*, Vol. 665, 48-57, Springer Verlag, 1993.
34. M. Falaschi, G. Levi and C. Palamidessi. Declarative modeling of the operational behavior of logic languages. *Theoretical Computer Science*, 69:289-318, 1989.
35. J. Fischer. Termination analysis for Mercury using convex constraints. Master's thesis, University of Melbourne, Department of Computer Science and Software Engineering, 2002.
36. M. J. García de la Banda, K. Marriott, P. Stuckey, and H. Søndergaard. Differential methods in logic programming analysis. *Journal of Logic Programming*, 35(1):1-38, April 1998.
37. S. Genaim and M. Codish. Inferring termination conditions for logic programs using backwards analysis. *Logic for Programming, Artificial Intelligence and Reasoning, 8th International Proceedings, Lecture Notes in Computer Science*, Vol. 2250, 685-694, Springer Verlag, 2001.
38. R. Gori. An abstract interpretation approach to termination of logic programs. *Logic for Programming and Automated Reasoning, 7th International Conference, Springer Lecture Notes in Computer science*, Vol. 1955, 362-380, Springer Verlag, 2000.
39. R. L. Graham. *Rudiments of Ramsey theory*. Number 45 in regional conference series in mathematics. American Mathematical Society, 1980.
40. M. Krishna Rao, D. Kapur and R. Shyamansundar. Transformational methodology for proving termination of logic programs. *Journal of Logic Programming*, 34:1-41, 1998.
41. C. S. Lee, N. D. Jones and A. M. Ben-Amram. The Size-Change Principle for Program Termination. *ACM Symposium on Principles of Programming Languages 2001*, 81-92.
42. N. Lindenstrauss. Homepage: <http://www.cs.huji.ac.il/~naomil/>
43. N. Lindenstrauss and Y. Sagiv. Checking Termination of Queries to Logic Programs, 1996. <http://www.cs.huji.ac.il/~naomil/>

44. N. Lindenstrauss and Y. Sagiv. Automatic Termination Analysis of Logic Programs. In *Proceedings of the 14th International Conference on Logic Programming*, ed. L. Naish, MIT Press, 1997, 63-77.
45. N. Lindenstrauss and Y. Sagiv. Automatic Termination Analysis of Logic Programs — version with Appendix. <http://www.cs.huji.ac.il/~naomil/>
46. N. Lindenstrauss, Y. Sagiv and A. Serebrenik. *TermiLog*: A System for Checking Termination of Queries to Logic Programs. In *Computer Aided Verification, 9th International Conference*, ed. O. Grumbach, LNCS 1254, 63–77, Springer Verlag, 1997.
47. N. Lindenstrauss, Y. Sagiv and A. Serebrenik. Unfolding the Mystery of *Mergesort*. In *Proceedings of the 7th International Workshop on Logic Program Synthesis and Transformation*, ed. N. Fuchs, LNCS 1463, 206–225, Springer Verlag, 1998.
48. J. W. Lloyd. *Foundations of Logic Programming*. Springer Verlag, second edition, 1987.
49. M. Marchiori. Proving existential termination of normal logic programs. In M. Wirsing and M. Nivat, eds., *Proceedings of the 5th International Conference on Algebraic Methodology and Software Technology, Lecture Notes in Computer Science*, Vol. 1101, 375-390, Springer Verlag, 1996.
50. F. Mesnard and R. Bagnara. cTI: A constraint-based termination inference tool for ISO-Prolog. *TPLP*, to appear, 2004.
51. F. Mesnard and U. Neumerkel. Applying static analysis techniques for inferring termination conditions of logic programs. In *Static Analysis, 8th International Symposium*, LNCS, Vol. 2126, 93–110, Springer Verlag, 2001.
52. F. Mesnard and S. Ruggieri. On proving left termination of constraint logic programs. *ACM Transactions on Computational Logic*, 4(2):207-259, 2003.
53. K. Muthukumar, F. Bueno, M. J. García de la Banda, and M. V. Hermenegildo. Automatic compile-time parallelization of logic programs for restricted, goal level, independent and parallelism. *Journal of Logic Programming*, 38(2):165–218, February 1999.
54. N. J. Nilsson. *Principles of Artificial Intelligence*. Tioga Publishing Company, 1980.
55. E. Ohlebusch, C. Claves and C. Marché. TALP: A tool for the termination analysis of logic programs. *11th International Conference on Rewriting Techniques and Applications, Lecture Notes in Computer Science*, Vol. 1833, 270-273, Springer Verlag, 2000.
56. R. A. O’Keefe. *The Craft of Prolog*. MIT Press, 1990.
57. D. Pedreschi, S. Ruggieri and J.-G. Smaus. Classes of terminating logic programs. *Theory and Practice of Logic Programming*, 2(3):369-418, 2002.
58. L. Plümer. *Termination Proofs for Logic Programs*. Springer Verlag, LNAI 446, 1990.
59. L. Plümer. Automatic Termination Proofs for Prolog Programs Operating on Nonground Terms. In *International Logic Programming Symposium*. MIT Press, 1991.
60. R. Ramakrishnan, F. Bancilhon, and A. Silberschatz. Safety of recursive Horn clauses with infinite relations. In *Proceedings of the ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, 1987.
61. B. Ramkumar and L. V. Kalé. Compiled execution of the reduce-OR process model on multiprocessors. In E. L. Lusk and R. A. Overbeek, editors, *Logic Programming, Proceedings of the North American Conference*, pages 313–331. MIT Press, 1989.
62. F. P. Ramsey. On a Problem of Formal Logic. *Proc. of London Math. Soc.*, 30:264-286, 1928.

63. U. S. Reddy. Transformation of logic programs into functional programs. In *Proceedings of the 1984 International Conference on Logic Programming*, 187–196, IEEE-CS, 1984.
64. S. Ruggieri. Decidability of logic program semantics and application to testing. *Journal of Logic Programming*, 46(1-2):103-137, 2000.
65. S. Ruggieri. \exists -universal termination of logic programs. *Theoretical Computer Science*, 254(1-2):273-296,2001.
66. Y. Sagiv. A termination test for logic programs. In *International Logic Programming Symposium*. MIT Press, 1991.
67. V. Santos Costa, D. H. D. Warren, and R. Yang. Andorra-I: A parallel Prolog system that transparently exploits both And- and Or-parallelism. In *Proceedings of the Third ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming (PPOP)*, pages 83–93. ACM Press, 1991.
68. *SICStus Prolog User's Manual*. Swedish Institute of Computer Science.
69. J.-G. Smaus. *Modes and Types in Logic Programming*. PhD thesis, University of Kent at Canterbury, 1999.
70. J.-G. Smaus. Proving termination of input-consuming logic programs. In D. De Schreye, ed., *Proceedings of the International Conference on Logic Programming*, pages 335–349. MIT Press, 1999.
71. K. Sohn and A. Van Gelder. Termination Detection in Logic Programs using Argument Sizes. *Proceedings of the Tenth ACM SIGACT-SIGART-SIGMOD Symposium on Principles of Database Systems*, 1991, 216–226.
72. C. Speirs, Z. Somogyi and H. Søndergaard. Termination Analysis for Mercury. In *Proc. of the 1997 Intl. Symp. on Static Analysis*, P. van Hentenrick, ed., LNCS, Vol. 1302, 157–171, Springer Verlag.
73. L. Sterling and E. Shapiro. *The Art of Prolog*. MIT Press, 1986.
74. TermiLog. <http://www.cs.huji.ac.il/~naomil/termilog.php>
75. E. Tick and C. Banerjee. Performance evaluation of Monaco compiler and runtime kernel. In D. S. Warren, editor, *Logic Programming, Proceedings of the Tenth International Conference on Logic Programming*, pages 757–773, 1993.
76. J. D. Ullman and A. Van Gelder. Efficient tests for top-down termination of logical rules. *JACM* 35:2(1988), 345-373.
77. M. H. Van Emden. An Interpretation Algorithm for Prolog Programs. In *Implementations of Prolog*, ed. J. A. Campbell, 1984.
78. A. Van Gelder. Deriving constraints among argument sizes in logic programs. *Annals of Mathematics and Artificial Intelligence*, 3:361–392, 1991.
79. A. Van Gelder. Personal communication.
80. T. Vasak and J. Potter. Characterisation of terminating logic programs. In *Proceedings of the 1986 Symposium on Logic Programming*, 140–147, 1986.
81. K. Verschaetse. Static termination analysis for definite Horn clause programs. PhD thesis, Department of Computer Science, K. U. Leuven, Belgium, 1992.
82. K. Verschaetse, S. Decorte, and D. De Schreye. Automatic Termination Analysis. *LOPSTR*, 1992, 168-183.
83. K. Verschaetse and D. De Schreye. Deriving Termination Proofs for Logic Programs, using Abstract Procedures. In *Proc. of the 8th ICLP*, 301-315, 1991.

Table 1. Examples from [5, 30]

Program	F	R	Query	Inst	Constr	Prs	Pr#	A	Rem			
<i>Examples of [5]</i>												
append	1	1	app(b,f,f)	0.00		0.01	1	T				
			app(f,f,b)			0.01	1	T				
curry	1	4	type(b,b,f)	0.03		0.36	15	T				
			general norm: list-size for lists, term-size otherwise									
dc_schema	0	2	dc_solve(b,f)	0.01		0.03	1	T	using <i>dc_mod</i>			
fold	2	1	fold(b,b,f)	0.00		0.02	1	T				
gtsolve	0	1	gtsolve(b,f)	0.00		0.00	0	T	using <i>gt_mod</i>			
list	1	1	list(b)	0.00		0.00	1	T				
lte	2	2	goal	0.00		0.01	2	T				
map	2	1	map(b,f)	0.00		0.01	1	T				
member	1	1	member(f,b)	0.00		0.01	1	T				
mergesort	5	4	mergesort(b,f)					N-				
mergesort	5	4	mergesort(b,f)			0.90	9	T	after 4.7 transf			
naive_rev	2	2	reverse(b,f)	0.00		0.01	2	T				
ordered	2	1	ordered(b)	0.00		0.01	1	T				
overlap	1	3	overlap(b,b)	0.00		0.01	2	T				
permutation	2	2	perm(b,f)	0.01	0.11			N-				
quicksort	3	4	qs(b,f)	0.01	5.23	5.06	6	T				
select	1	1	select(f,b,f)	0.00		0.01	1	T				
subset	2	2	subset(b,b)	0.00		0.02	2	T				
			subset(f,b)							N+		
sum	1	1	sum(f,b,f)	0.00		0.01	1	T				
			sum(f,f,b)							T		
<i>Examples of [30]</i>												
append	1	1	append(b,f,f)	0.00		0.01	1	T				
			append(f,f,b)							0.01	1	T
			append(f,b,f)									N+
bool	2	4	dis(b)	0.00		0.02	5	T				
			con(b)							0.02	5	T
duplicate	1	1	duplicate(b,f)	0.00		0.00	1	T				
merge	2	2	merge(b,b,f)	0.00		0.06	3	T				
permute	2	2	permute(b,f)	0.01	0.02	0.01	2	T				
reverse	1	1	reverse(b,f,b)	0.00		0.01	1	T				
sum	1	1	sum(b,b,f)	0.00		0.02	1	T				

Table 2. Examples from [58]

Program	Ref	F R	Query	Inst	Constr	Prs	Pr#	A
append	1.1	1 1	append(b,f,f)	0.00		0.01	1	T
			append(f,f,b)			0.01	1	T
perm	1.2	2 2	perm(b,f)	0.01	0.11			N-
perm_t		2 2	perm(b,f)	0.02	0.03	0.04	3	T
transitivity	2.3.1	2 1	p(f,b)	0.00				N+
lcl_list	3.5.6	3 2	p(f)	0.00				N+
append3	4.0.1	1 2	append3(b,b,b,f)	0.01		0.01	1	T
			append3(f,b,b,b)					N+
merge	4.4.3	2 2	merge(b,b,f)	0.00		0.07	3	T
perm_a	4.4.6a	3 2	perm(b,f)	0.00		0.02	2	T
arithmetic	4.5.2	1 4	s(b,f)	0.00				N+
loops	4.5.3a	1 1	p(b)	0.00				N+
	4.5.3b	2 1	goal \equiv p(X),q(X)	0.00				N+
	4.5.3c	2 1	goal \equiv p(X),q(X)	0.00				N+
turing	5.2.2	1 6	turing(b,b,b,f)	0.52				N+
quicksort	6.1.1	3 4	qsort(b,f)	0.01	5.38	5.84	6	T
mult	7.2.9	2 2	mult(b,b,f)	0.01		0.02	2	T
reach1	7.6.2a	1 3	reach(b,b,b)	0.01				N+
reach2	7.6.2b	1 3	reach(b,b,b,b)	0.01				N-
reach3	7.6.2c	2 4	reach(b,b,b,b)	0.01	0.09	0.23	6	T
mergesort1	8.2.1	5 4	mergesort(b,f)					N-
mergesort1	8.2.1	5 4	mergesort(b,f)			after 4.7 transf 0.90	9	T
mergesort2	8.2.1a	5 4	mergesort(b,f)	0.02	0.79	0.52	6	T
mergesort_t		6 7	mergesort(b,f)	0.04	0.53	0.15	9	T
minsort	8.3.1	4 6	minsort(b,f)	0.01	0.15			N-
minsort1	8.3.1a	3 6	minsort(b,f)	0.00	0.09	0.11	5	T
evenodd	8.4.1	1 2	even(b)	0.00		0.01	4	T
			odd(b)			0.01	4	T
parser	8.4.2	3 6	e(b,f)	0.00	0.03	0.10	14	T

Table 3. Examples from [18]

Prog	F	R	Query	Inst	Constr	Prs	Pr#	A	Rem
aiakl	3	10	init_vars(b,b,f,f)	0.27		24.11	215	T	
ann	101	76	go(b)	84.43				N+	
bid	24	26	bid(b,f,f,f)	0.7		0.37	7	T	
boyer	63	73	tautology(b)	0.23				N-	
browse	4	25	main	2.25				N-	
deriv	2	16	d(b,b,f)	0.03		0.23	1	T	
fib_t	2	4	fib(b,f)	0.00		0.06	4	T	succ
grammar	12	4						T	no rec
hanoiapp_suc	2	2	shanoi(b,b,b,b,f)	0.06		0.72	7	T	succ
mmatrix	7	8	mmultiply(b,b,f)	0.02		0.04	3	T	
			trans_m(b,f)					N+	
money	6	8	money(f,f,f, f,f,f,f,f)	0.35	0.93	0.85	2	T	
					0.63+0.02	0.02	2	T	big
occur	3	6	occurall(b,b,f)	0.01		0.06	3	T	
peephole	72	62	popt1(b,f)						mem
progeom	4	14	pds(b,f)	0.08	5.1			N	
qplan	63	85	qplan(b,f)	73.26				N	
qsortapp	3	4	qsort(b,f)	0.01	5.55	4.97	6	T	
query	50	2						T	no rec
rdtok	7	48							cannot
read	15	73							cannot
serialize	5	9	serialize0(b,f)	0.04	2.87	2.64	8	T	
			general norm: term-size except $\ pair(X, Y)\ = 1 + \ X\ $						
tak	0	3							cannot
tictactoe	26	43							cannot
warplan	43	55							mem
zebra	14	4	zebra(f,f,f,f,f,f)	1440	5.36+0.01	0.02	4	T	big
				0.96	9.57	0.76	2	T	=
					0.68+0.01	0.03	2	T	= and big
zebra.pt	2	3	houses(f)	0.00	0.02	0.03	2	T	

Table 4. Examples from [45]

Program	Ref	F	R	Query	Inst	Constr	Prs	Pr#	A
ack	[73]	1	2	ack(b,b,f)	0.00		0.05	3	T
arit_exp	[82]	0	6	e(b)	0.01		0.05	12	T
associative		1	3	normal_form(b,f)	0.01	0.03	0.02	2	T
		general norm: term-size except $\ op(X, Y)\ = 1 + 2\ X\ + \ Y\ $							
blocks	[54]	12	5	tower(b,b,b,f)	0.02	0.13			N+
credit	[73]	21.1/2	33	24 credit(b,f)	0.06		1.12	4	T
deep_rev		1	3	deep(b,f)	0.00		0.07	7	T
game	[4]	0	1	win(b)	0.00		0.01	1	T
		Using <i>game_mod</i>							
huffman		2	8	huffman(b,f)	0.12	0.09	0.07	2	T
				code(b,f,f)			0.01	1	T
p	[21]	1	2	p	0.01				N+
pql		2	2	p(b,f)	0.01	0.02	0.21	13	T
				q(b,f)		0.02	0.24	14	T
				q(f,b)			0.17	9	T
				p(f,b)					N+
				q(f,f)					N+
queens	[21]	4	5	queens(b,f)	0.01	0.21	0.38	4	T
sicstus1	[68]	3	4	concatenate(b,f,f)	0.00		0.01	1	T
				concatenate(f,f,b)			0.01	1	T
				member(f,b)			0.01	1	T
				reverse(b,f)			0.03	1	T
				concatenate(f,b,f)					N+
				member(b,f)					N+
				reverse(f,b)					N+
sicstus2	[68]	4	2	descendant(b,b)	0.00				N+
sicstus3	[68]	2	7	put_assoc(b,b,b,f)	0.14		0.61	9	T
				get_assoc(b,b,f)			0.26	8	T
sicstus4	[68]	0	7	d(b,b,f)	0.01		0.18	1	T
sublist	[77]	1	2	sublist(b,b)	0.01				N+
vangelder	[79]	1	10	q(b,b)	0.00		8.92	153	T
yale.s_p	[2]	2	3	holds(b,b)	0.00		0.48	9	T
				holds(f,b)			0.82	18	T
				holds(b,f)					N+