On Termination of Constraint Logic Programs

Livio Colussi[1], Elena Marchiori[2], Massimo Marchiori[1]

[1] Dept. of Pure and Applied Mathematics, Via Belzoni 7, 35131 Padova, Italy
e-mail: {colussi,max}@euler.math.unipd.it
[2] CWI, P.O. Box 94079, 1090 GB Amsterdam, The Netherlands
e-mail: elena@cwi.nl

Abstract. This paper introduces a necessary and sufficient condition for termination of constraint logic programs. The method is based on assigning a dataflow graph to a program, whose nodes are the program points and whose arcs are abstractions of the rules of a transition system, describing the operational behaviour of constraint logic programs. Then termination is proven using a technique inspired by the seminal approach of Floyd for proving termination of flowchart programs.

1 Introduction

The aim of this paper is to introduce a sufficient and necessary condition for termination of constraint logic programs (clp's for short). Termination of clp's is a fairly recent topic, and the only contribution we are aware of is by Mesnard [Mes93], cited in the recent survey [DSD94] on termination of logic programs. However, the aim of that work is different, namely to provide sufficient conditions for the termination problem of clp's, based on approximation techniques. Here we aim at an exact description of terminating clp's, to provide a better understanding of the termination problem for clp's, and to provide a basis for the development of formal methods for reasoning about run-time properties of clp's.

Termination behaviour of clp's is more subtle than that of logic programs. For instance, the presence of some constraints can turn an execution into a (finite) failure, because the actual state does not satisfy a constraint. A similar behaviour can be observed in some built-in's of Prolog (see e.g. [AMP94]). Moreover, in most CLP systems, the state is divided into two components containing the so-called *active* and *passive* constraint, and only the consistency of the active constraint is checked. Then the fact that satisfiability of passive constraints is not checked, affects the termination behaviour of the program: a constraint in the passive component might lead to an inconsistency which is never detected, and which would otherwise have led to termination (with failure). These observations show that the presence of constraints plays a crucial role in the termination behaviour of a clp, and that methods for proving termination for logic programs cannot be applied to deal with clp's in full generality.

In this paper we give a necessary and sufficient condition for the termination problem of clp's. We consider termination w.r.t. an initial set of states (the

precondition). Our approach is built on four main notions. First, the elementary computational steps of a clp are described by means of a *transition system*. Next, a *dataflow graph* is assigned to a program. Its nodes are the program points and its arcs are abstractions of the rules of the transition system. Further, a tuple of sets of states, called *invariant* is assigned to the dataflow, one set for each node of the dataflow graph. A set assigned to a node describes the final states of partial computations ending in that node. Finally, a function from states to a well-founded set W, called *W-function*, is associated with each node of the graph. These notions are combined in the definition of *termination triple*, which provides a characterization of terminating clp's (w.r.t. a precondition). Our approach is inspired by the technique introduced by Floyd [Flo67] to prove termination of flowchart programs. Intuitively, in a termination triple the invariants and the W-functions are chosen in such a way that every computation of the program is mapped into a decreasing chain of W. Then by the fact that W is well-founded it follows that every computation is finite.

The notion of termination triple provides a formal basis for reasoning about run-time properties of clp's. We introduce a methodology for finding termination triples, and we show how this method can be modified to yield a practical sufficient criterion for proving termination of normal clp's. To help the reader to focus more on the approach than on the technicalities, the presentation deals with *ideal* CLP systems, where the constraint inference mechanism does not distinguish between active and passive constraints. We discuss in the Conclusion how to extend the results to more general CLP systems.

We have organized the paper as follows. After a few preliminaries on notation and terminology, three sections present the main notions of our approach: Section 3 introduces our transition system, Section 4 the notion of dataflow graph of a program, and Section 5 introduces the notion of invariant for a program. Then in Section 6 we introduce the notion of termination triple, Section 7 contains a methodology for finding termination triples, Section 8 discusses the sufficient criterion. Finally, Section 9 discusses the results and related approaches to study termination of logic programs. For lack of space, we omitted the proofs. They can be found in the full version of the paper.

2 Preliminaries

Let Var be an (enumerable) set of variables, with elements denoted by x, y, z, u, v, w. We shall consider the set $VAR = Var \cup Var^0 \cup \ldots \cup Var^k \cup \ldots$, where $Var^k = \{x^k \mid x \in Var\}$ contains the so-called *indexed variables* (`i`-variables for short) of *index* k. These special variables will be used to describe the standardization apart process, which distinguishes copies of a clause variable which are produced at different calls of that clause. Thus x^k and x^j will represent the same clause variable at two different calls. This technique is known as 'structure-sharing', because x^k and x^j share the same structure, i.e. x. For an index k and a syntactic object E, E^k denotes the object obtained from E by replacing every variable x with the `i`-variable x^k. We denote by $Term(VAR)$ (resp. $Term(Var)$)

the set of terms built on VAR (resp. Var), with elements denoted by r, s, t.

A sequence $E_1, \ldots, E_k$ of syntactic objects is denoted by $\overline{E}$ or $\langle E_1, \ldots, E_k \rangle$, and $(s_1 = t_1 \wedge \ldots \wedge s_k = t_k)$ is abbreviated by $\overline{s} = \overline{t}$.

Constraint Logic Programs

The reader is referred to [JM94] for a detailed introduction to Constraint Logic Programming. Here we present only those concepts and notation that we shall need in the sequel.

A constraint c is a (first-order) formula on $Term(VAR)$ built from primitive constraints. We shall use the symbol $\mathcal{D}$ both for the domain and the set of its elements. We write $\mathcal{D} \models c$ to denote that c is valid in all the models of $\mathcal{D}$.

A *constraint logic program* $\mathcal{P}$, simply called program or clp, is a (finite) set of clauses $H \leftarrow A_1, \ldots, A_k$ (denoted by C, D), together with one goal-clause $\leftarrow B_1, \ldots, B_m$ (denoted by G), where H and the A_i's and B_i's are atoms built on $Term(Var)$ (primitive constraints are considered to be atoms as well) and H is not a constraint. Atoms which are not constraints are also denoted by $p(\overline{s})$, and $pred(p(\overline{s}))$ denotes p; for a clause C, $pred(C)$ denotes the predicate symbol of its head. A clause whose body either is empty or contains only constraints is called *unitary*.

3 Operational Semantics

To design our method for characterizing the termination behaviour of clp's, we start with a description of the operational behaviour of a clp by means of a transition system. In this transition system standardization apart plays a central role. The reason is that we want to use a suitable representation of program variables during the execution, which will be used in Section 7 where we study how to render *practical* our characterization.

As in the standard operational model states are consistent constraints, i.e.

$$States \stackrel{\text{def}}{=} \{c \in \mathcal{D} \mid c \text{ consistent}\},$$

denoted by c or α. We use the two following operators on states:

$$push, pop : States \rightarrow States,$$

where $push(\alpha)$ is obtained from α by increasing the index of all its `i`-variables by 1, and $pop(\alpha)$ is obtained from α by first replacing every `i`-variable of index 0 with a new fresh variable, and then by decreasing the index of all the other `i`-variables by 1. For instance, suppose that α is equal to $(x^1 = f(z^0) \wedge y^0 = g(x^2))$. Then $push(\alpha)$ is equal to $(x^2 = f(z^1) \wedge y^1 = g(x^3))$ and $pop(\alpha)$ to $(x^0 = f(u) \wedge v = g(x^1))$, where u and v are new fresh variables. These operators are extended in the obvious way to sets of states. *Push* and *pop* are used in the rules of our transition system to describe the standardization apart mechanism. The rules of this transition system, called TS, are given in Table 1. In a pair

R $(\langle p(\overline{s})\rangle \cdot \overline{A},\, \alpha\,) \longrightarrow (\overline{B} \cdot \langle pop\rangle \cdot \overline{A},\, push(\alpha) \wedge \overline{s}^1 = \overline{t}^0\,)$, if $C = p(\overline{t}) \leftarrow \overline{B}$ is in $\mathcal{P}$ and $push(\alpha) \wedge \overline{s}^1 = \overline{t}^0$ consistent
S $(\langle\, pop\rangle \cdot \overline{A},\, \alpha\,) \longrightarrow (\,\overline{A},\, pop(\alpha)\,)$
C $(\langle\, d\,\rangle \cdot \overline{A},\, \alpha\,) \longrightarrow (\,\overline{A},\, \alpha \wedge d^0\,)$, if d is a constraint and $\alpha \wedge d^0$ consistent

Table 1. Transition rules for CLP.

$(\overline{A}, \alpha)$, α is a state, and $\overline{A}$ is a sequence of atoms and possibly of tokens of the form *pop*, whose use is explained below. We fix a suitable standardization apart mechanism: In the standard operational semantics of (C)LP, every time a clause is called it is renamed apart, generally using indexed variables. Here if a clause is called then *push* is first applied to the state, and if it is released then *pop* is applied to the state. To mark the place at which this should happen the symbol *pop* is used. As mentioned above, this formalization will lead to an elegant method in Section 7. The rules of TS describe the standard operational behaviour of a clp (cf. e.g. [JM94]): Rule **R** describes a resolution step. Note that, the way the operators *push* and *pop* are used guarantees that every time an atom is called, its variables can be indexed with index equal to 0. Then, in rule **R** the tuple of terms $push(\overline{s}^0) (= \overline{s}^1)$ is considered, because a *push* is applied to the state. Rule **S** describes the situation where an atom has concluded with success its computation, i.e. when the control reaches a *pop*. In this case, the operator *pop* is applied to the state. Finally, rule **C** describes the execution of a constraint. Observe that we do not describe failure explicitly, by adding a corresponding *fail* state. Instead, a failure here occurs when no rule is applicable.

To refer unambiguously to clause variables, the following non-restrictive assumption is used.

Assumption 1 Different clauses of a program have disjoint sets of variables.

We call *computation*, denoted by τ, any sequence $\langle conf_1, \ldots, conf_k, \ldots\rangle$ of configurations s.t. for $k \geq 1$ we have that $conf_k \rightarrow conf_{k+1}$. We consider an operational semantics $\mathcal{T}(\mathcal{P}, \phi)$ for a program $\mathcal{P}$ w.r.t. a set ϕ of states, called *precondition*. This semantics describes all the computations starting in $(G,\, \alpha)$ (recall that G denotes the goal-clause of $\mathcal{P}$) with α in ϕ. It is defined as follows. We use $\cdot$ for the concatenation of sequences.

Definition 2. (partial trace semantics) $\mathcal{T}(\mathcal{P}, \phi)$ is the least set T s.t. $\langle (G,\, \alpha)\rangle$

is in T, for every $\alpha \in \phi$, and if $\tau = \tau' \cdot \langle(\overline{A},\, \alpha)\rangle$ is in T and $(\overline{A},\, \alpha) \rightarrow (\overline{B},\, \beta)$, then $\tau \cdot \langle(\overline{B},\, \beta)\rangle$ is in T. □

Observe that this is a very concrete semantics: the reason is that it is not meant for the study of program equivalence, but for the study of run-time properties of clp's, namely to characterize termination of clp's. Indeed, $\mathcal{T}(\mathcal{P}, \phi)$ will be used in Section 5 to define the notion of invariant for a program. This latter notion will play a central role in giving (in Section 6) a necessary and sufficient condition for the termination (w.r.t. a precondition) of clp's.

4 A Dataflow Graph for clp's

In this section we introduce the second notion used in our method, namely the dataflow graph of a program. Graphical abstractions of programs have been often used for static analysis of run-time properties. Here, we assign to a program a directed graph, whose nodes are the program points and whose arcs are abstractions of the transition rules of Table 1. In this choice, we have been inspired by the seminal work of Floyd [Flo67] for flowchart programs. To study termination, in general information on the form of the program variables bindings before and after the program atoms calls is needed. Methods for proving termination of logic programs based on graph abstraction, like for instance [BCF94, WS94], use inductive proof methods for proving such run-time properties, and use the graph only to detect possible sources of divergence by considering its cycles. Instead, in our approach, the graph is used both to derive run-time properties and for detecting possible sources of divergence.

We consider the leftmost selection rule, and view a program clause C : $H \leftarrow A_1, \ldots, A_k$ as a sequence consisting alternatingly of (labels l of) *program points* (pp's for short) and atoms,

$$H \leftarrow {}_{l_0}\, A_1\, {}_{l_1}\; \ldots\; {}_{l_{k-1}}\, A_k\, {}_{l_k}.$$

The labels l_0 and l_k indicate the *entry point* and the *exit point* of C, denoted by *entry*(C) and *exit*(C), respectively. For $i \in [1, k]$, l_{i-1} and l_i indicate the *calling point* and *success point* of A_i, denoted by *call*(A_i) and *success*(A_i), respectively. Notice that $l_0 = \mathit{entry}(C) = \mathit{call}(A_1)$ and $l_k = \mathit{exit}(C) = \mathit{success}(A_k)$. In the sequel *atom*$(l)$ denotes the atom of the program whose calling point is equal to l. For notational convenience the following assumptions are used. Note that they do not imply any loss of generality.

Assumption 3 $l_0, \ldots, l_k$ are natural numbers ordered progressively; distinct clauses of a program are decorated with different pp's; the pp's form an initial segment, say $\{1, 2, \ldots, n\}$ of the natural numbers; and 1 denotes the leftmost pp of the goal-clause, called the *entry point of the program.* Finally, to refer unambiguously to program atom occurrences, all atoms occurring in a program are supposed to be distinct.

In the sequel, $\mathcal{P}$ denotes a program and $\{1, \ldots, n\}$ the set of its pp's. Program points are used to define the notion of dataflow graph.

Definition 4. (dataflow graph) The *dataflow graph* $dg(\mathcal{P})$ *of* $\mathcal{P}$ is a directed graph $(Nodes, Arcs)$ s.t. $Nodes = \{1, \ldots, n\}$ and $Arcs$ is the subset of $Nodes \times Nodes$ s.t. (i, j) is in $Arcs$ iff it satisfies one of the following conditions:

- i is $call(A)$, where A is not a constraint, j is $entry(C)$ and $pred(C) = pred(A)$;
- i is $exit(C)$, j is $success(A)$ and $pred(A) = pred(C)$;
- i is $call(A)$ for some constraint A and j is $success(A)$.

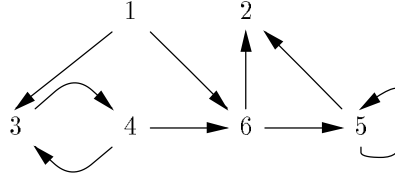
An element (i, j) of $Arcs$ is called *(directed) arc from i to j*. □

Arcs of $dg(\mathcal{P})$ are graphical abstractions of the transition rules of Table 1. Rule **R** is abstracted as an arc from the calling point of an atom to the entry point of a clause. Rule **S** is abstracted as an arc from the exit point of a clause to a success point of an atom. Finally, rule **C** is abstracted as an arc from the calling point of a constraint to its success point.

Example 1. The following program *Prod* is labelled with its pp's.

G: $\leftarrow$ $_1$ `prod(u,v)` $_2$
$C1$: `prod([x|y],z)` $\leftarrow$ $_3$ `z=x*w` $_4$ `prod(y,w)` $_5$
$C2$: `prod([ ],1)` $\leftarrow$ $_6$

The dataflow graph $dg(Prod)$ of *Prod* is pictured below.



□

Remark. One can refine Definition 4 by using also semantic information, i.e. by pruning the arcs stemming from the first two conditions if $\mathcal{D} \models \neg(\overline{s} = \overline{t})$, i.e. if $p(\overline{s})$ and $p(\overline{t})$ do not 'unify', where $p(\overline{s})$ is A and $p(\overline{t})$ is (a variant of) the head of C. □

Our notion of dataflow graph differs from other graphical representations of (c)lp's, as for instance the *predicate dependency graph* or the *U-graph* (see e.g. [DSD94]), mainly because of the presence in $dg(\mathcal{P})$ of those arcs from exit points of clauses to success points of atoms, such as the arc from 5 to 2 in $dg(Prod)$. These arcs are crucial in our method, because we use the graph not only for detecting possible divergences but also for deriving information on the run-time behaviour of the program, information needed in the termination analysis. In

contrast, methods for studying termination of logic programs, based on graph representation, use other static analysis methods for deriving this information.

A *path* is a finite non-empty directed path of $dg(\mathcal{P})$. Paths are denoted by π, and concatenation of paths by $\cdot$. Moreover, $path(i,j)$ denotes the set of all the paths from i to j, and $path(i)$ the set of all the paths from 1 to i.

5 Invariants for clp's

In this section we use the notion of dataflow graph to derive information on the run-time behaviour of programs which is relevant for the study of termination. To this end, we first relate paths of the dataflow graph and computations. Next, we use this relation to define the notion of assertion at a program point, which is the set containing the final states of all the partial traces ending in that program point.

We write *conf*, possibly subscripted, to denote a configuration $(\overline{A},\alpha)$ used in the rules of TS. The relation *Rel* relating paths and computations is defined by induction on the number of elements of a computation as follows.

The base case is $\langle(\langle p(\overline{s})\rangle\cdot\overline{A},\alpha)\rangle \mathit{Rel} \langle call(p(\overline{s}))\rangle$, and the induction case is as follows. Suppose that $\tau'\cdot\langle conf_1\rangle \mathit{Rel}\, \pi$ and that $\tau = \tau'\cdot\langle conf_1, conf_2\rangle$ (by definition this implies $conf_1 \rightarrow conf_2$). Then:

- $\tau \mathit{Rel}\, \pi\cdot\langle entry(C)\rangle$,
  if $conf_1 = (\langle p(\overline{s})\rangle\cdot\overline{A},\ \alpha\,)$ and C is the selected clause;
- $\tau \mathit{Rel}\, \pi\cdot\langle success(A)\rangle$, if $conf_1 = (\langle pop\rangle\cdot\overline{A},\ \alpha\,)$, where if $\pi = \langle l_1,\ldots,l_k\rangle$ then A is s.t.
  $call(A) = l_i$ for some $i\in[1,k]$ and for every B in $\mathcal{P}$
  $$|I_{call(B)}| = |I_{success(B)}|$$
  with $I_{\star(B)} = \{j \mid i<j\le k,\, l_j = \star(B)\}$, and $\star$ in $\{call, success\}$;
- $\tau \mathit{Rel}\, \pi\cdot\langle success(d)\rangle$, if $conf_1 = (\langle d\rangle\cdot\overline{A},\ \alpha\,)$.

In the sequel we refer to a set of states also by calling it *assertion*, to make the reader acquainted with the intuition that an assertion of some specification language could represent a set of states. In particular, we define the notion of assertion at program point. For a partial trace $\tau = \tau'\cdot\langle(\overline{A},\beta)\rangle$, we call β the *final state of* τ, denoted by $\mathit{finalstate}(\tau)$ and for a path π, we denote by $\mathit{lastnode}(\pi)$ its last element.

Definition 5. (assertion at pp) Let l be a pp of $dg(\mathcal{P})$. The *assertion at* l *(w.r.t.* ϕ*)*, denoted by $\mathcal{I}_l(\mathcal{P},\phi)$, is defined as follows:

$$\mathcal{I}_l(\mathcal{P},\phi) \;=\; \{\mathit{finalstate}(\tau) \mid \tau\in\mathcal{T}(\mathcal{P},\phi) \text{ and } \tau \mathit{Rel}\, \pi \text{ for some } \pi \text{ s.t. } l = \mathit{lastnode}(\pi)\}.$$

□

For instance, $\mathcal{I}_1(\mathcal{P}, \phi) = \phi$. The notion of assertion at pp is needed to give a sufficient and necessary condition for termination. However, it is too strong to be practical, because it implies the exact knowledge of the semantics of a program. Indeed, for proving termination, it is often enough to have partial knowledge of the semantics, i.e. to replace assertions at pp with suitable supersets. These supersets form the so-called invariants for $\mathcal{P}$. To define this notion, we need to formalize how paths modify states. We use ϕ, ψ, possibly subscripted, to denote sets of states.

Definition 6. Let π be a path and let $\tau = \langle(\overline{A}, \alpha)\rangle \cdot \tau'$ be a computation s.t. $\tau \; Rel \; \pi$. Then *finalstate*(τ) is called the *output of* π *w.r.t.* α, denoted by *output*(π, α). □

It can be shown that Definition 6 is well-formed, i.e. that if τ and τ' are s.t. both $\tau \; Rel \; \pi$ and $\tau' \; Rel \; \pi$, then $\tau = \tau'$, hence *output*(π, α) is uniquely defined. Observe that in some cases *output*(π, α) is not defined, namely when there is no τ s.t $\tau \; Rel \; \pi$.

Then the notion of invariant for $\mathcal{P}$ is defined as follows.

Definition 7. (invariant for $\mathcal{P}$) Let $\{1, \ldots, n\}$ be the set of nodes of $dg(\mathcal{P})$ and let ϕ be an assertion. We call the tuple $(\phi_1, \ldots, \phi_n)$ of assertions an *invariant for* $\mathcal{P}$ (w.r.t. ϕ) if: $\phi \subseteq \phi_1$; and for every $i, j \in [1, n]$, for every path $\pi \in path(i, j)$, and for every $\alpha \in \phi_i$ we have that if *output*(π, α) is defined, then it is in ϕ_j. □

6 Characterization of Termination

To give a characterization of terminating programs, the dataflow graph $dg(\mathcal{P})$ and an invariant $(\phi_1, \ldots, \phi_n)$ for $\mathcal{P}$ will be used. A function from states to a well-founded set W, called W-function, will be associated to certain nodes of $dg(\mathcal{P})$. The intuition is that for a terminating clp, each path of the graph can be mapped into a decreasing chain of W. For a path π from i to j, the W-function of i applied to a state α in ϕ_i is shown to be strictly greater than the W-function of j applied to *output*(π, α). To examine only a finite number of paths, we adapt a technique introduced by Floyd [Flo67] and formalized by Manna [Man70] to prove termination of flowchart programs: only those nodes of the graph which 'cut' some cycle are considered (see Definition 9), and only suitable paths connecting such nodes, called *smart*, are examined.

First, we define the notion of terminating w.r.t. a precondition clp.

Definition 8. (terminating clp's) Let ϕ be a set of states. A program is *terminating w.r.t.* ϕ if all the computations starting at (G, α), with $\alpha \in \phi$, are finite. □

Next, we define the notion of cutpoint set, originally introduced in [Flo67, Man70].

Definition 9. (cutpoint set) A set $\mathcal{C}$ of pp's of a program $\mathcal{P}$ is called a *cutpoint set for* $\mathcal{P}$ (and its members *cutpoints*) if every cycle of the dataflow graph contains at least one element of $\mathcal{C}$. □

So, cutpoints are meant to be control loci to check for possible nontermination caused by loops (cycles in the dataflow graph). Now, as in Floyd [Flo67], one has to consider the paths connecting two cutpoints, whose internal nodes are not cutpoints. However, observe that a path could describe a possible divergence only if it is contained in a cycle of $dg(\mathcal{P})$. Moreover, only cycles of $dg(\mathcal{P})$ which contain at least one entry point of a non-unitary clause, could represent an infinite computation. Thus we introduce the notion of *smart path*.

Definition 10. (smart path) Let $\mathcal{C}$ be a cutpoint set for $\mathcal{P}$. Let $l, l' \in \mathcal{C}$ and let π be a path in $path(l, l')$. Then π is *smart w.r.t.* $\mathcal{C}$ if the following conditions are satisfied:

1. there is a cycle in $dg(\mathcal{P})$ containing π and containing an entry point of a non-unitary clause of $\mathcal{P}$;
2. $\pi = \langle l \rangle \cdot \pi' \cdot \langle l' \rangle$ and no pp of π' is in $\mathcal{C}$.

□

Now we have all the tools to define the notion of termination triple, which provides a necessary and sufficient condition for the termination of a clp. We call *W-function* a function from states to a well-founded set $(W, <)$. Moreover, for a tuple $\Phi = (\psi_1, \ldots, \psi_n)$ of assertions, and a set $\mathcal{C} = \{i1, \ldots, ik\}$, with $1 \leq i1 < \ldots < ik \leq n$, we call $(\psi_{i1}, \ldots, \psi_{ik})$ the restriction to $\mathcal{C}$ of Φ.

Definition 11. (termination triple) Let ϕ be a set of states. Let $\mathcal{C}$ be a set of nodes of $dg(\mathcal{P})$; let $\Phi = \{\phi_l \,|\, l \in \mathcal{C}\}$ be a set of assertions; and let $\mathbf{w} = \{w_l \,|\, l \in \mathcal{C}\}$ be a set of W-functions. Then $(\mathcal{C}, \Phi, \mathbf{w})$ is a *termination triple for* $\mathcal{P}$ *w.r.t.* ϕ if:

1. $\mathcal{C}$ is a cutpoint set for $\mathcal{P}$;
2. Φ is the restriction to $\mathcal{C}$ of an invariant for $\mathcal{P}$ w.r.t. ϕ;
3. for every $l, l' \in \mathcal{C}$ and smart path (w.r.t. $\mathcal{C}$) $\pi \in path(l, l')$, we have that if $\alpha \in \phi_l$ and $output(\pi, \alpha)$ is defined then $w_l(\alpha) > w_{l'}(output(\pi, \alpha))$.

□

Then we have the following necessary and sufficient condition for termination.

Theorem 12. (termination characterization) *Let* ϕ *be a set of states. Let* $\mathcal{C}$ *be any cutpoint set for the program* $\mathcal{P}$*. Then* $\mathcal{P}$ *terminates w.r.t.* ϕ *if and only if there is a termination triple* $(\mathcal{C}, \Phi, \mathbf{w})$ *for* $\mathcal{P}$ *w.r.t.* ϕ*.*

In Sections 7 we shall introduce a method for finding termination triples. Moreover, in Section 8 we shall introduce a sufficient criterion based on this characterization.

7 Finding Termination Triples

We have seen how termination of a clp can be characterized by means of the notion of termination triple. This result is theoretically interesting. It provides a better understanding of the termination problem for clp's, and it can serve as a theoretical framework on which automatic techniques can be built. The attentive reader, however, will have observed that we have not used the special standardization apart mechanism incorporated in the rules of the transition system TS of Table 1. Indeed, as one would expect, the characterization we have given does not depend on the standardization apart mechanism.

The reason of the introduction of this mechanism is related with the issue of *finding* a termination triple. In this section we shall discuss a powerful methodology to prove that a triple $(\mathcal{C}, \Phi, \mathbf{w})$ is a termination triple. This methodology relies on the specific form of the rules of TS, hence on indexed variables and on the operators *pop* and *push*.

First, we give an inductive description of the strongest postcondition of a path. Next, we introduce a sound and (relatively) complete method to prove that a tuple of assertions is an invariant for the program.

7.1 Outputs of Paths

We show here how the notion of output of a path can be given inductively, without using the relation *Rel*.

We have that $output(\langle 1 \rangle, \alpha) = \alpha$. Moreover, when the initial state α satisfies suitable conditions, then the $output(\pi, \alpha)$ can be inductively computed. To this end, the following set of states is needed:

$$free(x) = \{\alpha \mid \mathcal{D} \models \alpha \rightarrow \forall x.\alpha\};$$

it describes those states where x is a free variable. The intuition is that x is free in a state if it can be bound to any value without affecting that state. For instance, $y = z$ is in $free(x)$, because x does not occur in the formula. Also $y = z \wedge x = x$ is in $free(x)$, because $\mathcal{D} \models (y = z \wedge x = x) \rightarrow \forall x\, (y = z \wedge x = x)$. Further, we write $\phi \wedge c$ to denote the set $\{\alpha \wedge c \in States \mid \alpha \in \phi\}$. Moreover, it is convenient to make the following assumptions on non-unitary (goal-)clauses.

Assumption 13 The body of every non-unitary clause does not contain two atoms with equal predicate symbol; and at least one argument of its head is a variable.

This assumption is not restrictive. It can be shown that every program can be transformed into one satisfying Assumption 13. The transformation will in general modify the semantics of the original program (the set of pp's changes and new predicates could be introduced). However, it is easy to define a syntactic transformation that allows us to recover the semantics of the original program.

Because of the second assumption, we can fix a variable-argument of the head of a clause C, that we call *the characteristic variable of* C, denoted by x_C.

Also, a new fresh variable x_G is associated with the goal-clause G, called *the characteristic variable of* G. These variables play a crucial role in the following result, to be explained below.

Theorem 14. *Let α be a state and let $\pi = \pi' \cdot \langle l_k \rangle$ be a path, where $\pi' = \langle l_1, \ldots, l_{k-1} \rangle$. Suppose that $\beta = output(\pi', \alpha)$ is defined. Then:*

- *if $l_k = entry(C)$, $l_{k-1} = call(A)$ for some atom $A = p(\overline{s})$, and if $push(\beta) \wedge (\overline{s}^1 = \overline{t}^0)$ is consistent then:*

$$output(\pi, \alpha) = push(\beta) \wedge (\overline{s}^1 = \overline{t}^0),$$

  where $p(\overline{t})$ is the head of C;
- *if $l_k = success(A)$ with A not a constraint, $l_{k-1} = exit(D)$ for some clause D, and if $pop(\beta) \in \neg free(x^0_C)$ where C is the clause containing A, then*

$$output(\pi, \alpha) = pop(\beta);$$

- *if $l_k = success(A)$ with A a constraint, and if $\beta \wedge A^0$ is consistent then:*

$$output(\pi, \alpha) = \beta \wedge A^0.$$

The requirements on the characteristic variables are needed to rule out all those paths which are not semantic, i.e. which do not describe partial traces. Informally, whenever a state is propagated through a semantic path the variable x^0_C is initially free (by assumption). Then, the index of x_C is increased and decreased by means of the applications of the *push* and *pop* operators. When C is called, then x^0_C is bound (because by assumption it occurs in the head of C), hence x^0_C is not free. From that moment on its index will be increased and decreased and it will become 0 *only* if the success point of an atom of the body of C is reached. If the success point of an atom of G is reached, then x^0_G is not free. Moreover, for each clause C different from G, x^0_C is free, because either C was never called, or x^0_C has been replaced with a fresh variable by an application of *pop*.

Example 2. The following example illustrates the crucial role of the characteristic variables to discriminate those paths which are not *semantical paths*. Consider again the program *Prod*. Let $\pi = \langle 1, 3, 4, 6, 2 \rangle$ and let $\alpha = (x^0_G = 0)$, where 0 is a constant. This path is not semantical, i.e. it does not describe a computation. Then, the output of this path w.r.t. α is not defined. Indeed, at program point 2 we obtain that x^0_G is free, thus Theorem 14 is not applicable. The behaviour, with respect to freeness, of the characteristic variables during the propagation of α through π is described in Table 2.

Note instead that the path obtained from π by replacing 2 with 5 is a semantical path (i.e. x^0_{C1} is not free at pp 5). □

at pp	x_G^0	x_G^1	x_G^2	x_{C1}^0	x_{C1}^1	x_{C2}^0
1	not free	free	free	free	free	free
3	free	not free	free	not free	free	free
4	free	not free	free	not free	free	free
6	free	free	not free	free	not free	not free
2	free	not free	free	not free	free	free

Table 2. Characteristic variables through π

7.2 Proving Invariants for clp's

We introduce now a necessary and sufficient condition to prove that an n-tuple $(\phi_1, \ldots, \phi_n)$ of assertions is an invariant for $\mathcal{P}$.

Recall that we denote by $\{1, \ldots, n\}$ the set of pp's of a program $\mathcal{P}$. Moreover, $atom(l)$ denotes the atom of the program whose calling point is l. For a node j of $dg(\mathcal{P})$, let $input(j)$ denote the set of the nodes i s.t. (i, j) is an arc of $dg(\mathcal{P})$. Then we have the following theorem.

Theorem 15. (characterization of invariants for $\mathcal{P}$) *Let $(\phi_1, \ldots, \phi_n)$ be an n-tuple of assertions s.t. $\phi_1 \subseteq \neg free(x_G^0)$, and $\phi_1 \subseteq free(x_C^0)$ for every non-unitary clause C different from G. Then $(\phi_1, \ldots, \phi_n)$ is an invariant for $\mathcal{P}$ if and only if for $i \in [1, n]$ we have that:*

1. *if $i = entry(C)$ then $push(\phi_j) \wedge (\overline{s}^1 = \overline{t}^0) \subseteq \phi_i$, for every $j \in input(i)$, where $p(\overline{t})$ is the head of C and $p(\overline{s}) = atom(j)$;*
2. *if $i = success(A)$ and A is not a constraint then $pop(\phi_j) \cap \neg free(x_C^0) \subseteq \phi_i$, for every $j \in input(i)$,*
   where C is the clause containing A;
3. *if $i = success(A)$ and A is a constraint then $\phi_{i-1} \wedge A^0 \subseteq \phi_i$.*

Let us comment on the above theorem, using the transition system of Table 1: let $\overline{A}$ denote a generic sequence of atoms and/or tokens. Then 1 states that $\phi_{entry(C)}$ contains those states obtained by applying rule **R** to $(\langle atom(j)\rangle \cdot \overline{A}, \alpha)$, for every $\alpha \in \phi_j$ and every $j \in input(entry(C))$. Further, 2 states that when A is not a constraint, then $\phi_{success(A)}$ contains those states obtained by applying rule **S** to $(\langle pop\rangle \cdot \overline{A}, \alpha)$, for every $\alpha \in \phi_j$ and every $j \in input(success(A))$. Finally, 3 states that when A is a constraint, then $\phi_{success(A)}$ contains those states obtained by applying the transition rule **C** to $(\langle A\rangle \cdot \overline{A}, \alpha)$, for every $\alpha \in \phi_{call(A)}$.

This theorem is derived from a fixpoint semantics which has been introduced in a companion paper [CMM95]. The conditions 1-3 of Theorem 15 correspond to the three cases of the definition of an operator F on n-tuples of assertions whose least fixpoint μF yields a semantics equal to $(\mathcal{I}_1(\mathcal{P}, \phi), \ldots, \mathcal{I}_n(\mathcal{P}, \phi))$. For instance, 1 corresponds to the case where F maps a tuple $(\psi_1, \ldots, \psi_n)$ to a tuple $(\phi_1, \ldots, \phi_n)$ s.t. $\phi_i = \cup_{j \in input(i)}(push(\psi_j) \wedge (\overline{s}^1 = \overline{t}^0))$. The other cases of the

definition of F are obtained analogously. Then the proof of Theorem 15 is an easy consequence of the equality between μF and $(\mathcal{I}_1(\mathcal{P},\phi),\ldots,\mathcal{I}_n(\mathcal{P},\phi))$.

7.3 A Methodology

Theorem 15 can be used as a basis for a sound and complete proof method for proving invariants of clp's. One has to define a specification language to express the properties of interest. Then, a formula of the language is interpreted as a set of states, conjunction is interpreted as set intersection, negation as set-complementation, and implication as set inclusion. The predicate relation $free$ has to be in the specification language, and the operators *pop* and *push* should be defined in the expected way on formulas. Simpler methods can be obtained from Theorem 15, by loosing completeness. We shall introduce in the following section one of such methods.

To summarize, we obtain the following methodology to study termination of clp's. To find a termination triple for $\mathcal{P}$ w.r.t. ϕ:

- construct $dg(\mathcal{P})$;
- select a cutpoint set;
- use Theorem 15 to find an invariant for $\mathcal{P}$;
- find a suitable set of W-functions;
- use Theorem 14 to check condition 3. of the definition of termination triple.

We conclude this section with a simple example.

Example 3. Consider the program *Prod* of Example 1. Let $true$ denote the set of all states and let $list(x)$ denote the set of states where x is a list. Take $\phi = (list(u^0) \wedge \neg free(x_G^0) \wedge free(z^0))$. We show that *Prod* terminates w.r.t. ϕ.

The dataflow graph $dg(Prod)$ for *Prod* was already given in Example 3.

$\mathcal{C} = \{3, 5\}$ is a cutpoint set for *Prod*.

Let $\phi_1 = \phi$, $\phi_2 = true$, $\phi_3 = \phi_4 = list(y^0)$, $\phi_5 = \phi_6 = true$. It is easy to check using Theorem 15 that $\Phi = (\phi_1,\ldots,\phi_6)$ is an invariant (w.r.t. ϕ) for *Prod*.

Consider the following W-functions, where the well-founded set W is here the set of natural numbers: $w_3 = w_5 = ||y^0||$, where $||t||$ denotes the length of t if t is a list and 0 otherwise.

In order to show that $(\{3,5\},\{\phi_3,\phi_5\},\{w_3,w_5\})$ is a termination triple, we have only to consider the smart path $\pi = \langle 3, 4, 3\rangle$.
Let α in ϕ_3 and suppose that $w_3(\alpha) = k$. Then α is in $\phi_3 \wedge (||y^0|| = k)$. Using Theorem 14 we have that $\beta = output(\pi, \alpha \wedge ||y^0|| = k)$ is defined, with $\beta = \big(list(y^1) \wedge ||y^1|| = k \wedge z^1 = x^1 * p^1 \wedge y^1 = [x^0|y^0] \wedge p^1 = z^0\big)$. Then $w_3(output(\pi,\alpha)) = (||y^1|| - 1) = (k-1)$; and from $k - 1 < k$ we obtain $w_3(output(\pi,\alpha)) < w_3(\alpha)$.

Thus $(\mathcal{C},\{\phi_3,\phi_5\},\{w_3,w_5\})$ satisfies the three conditions of Definition 11, and hence *Prod* is terminating w.r.t. ϕ. □

8 A Sufficient Criterion

In this section we discuss a variation of the above methodology which will yield a sufficient criterion for termination which is more practical, yet less powerful, than the one given in the previous section. The idea is to extract a small subgraph of the dataflow graph, called *cyclic*, to be used in the termination analysis.

Definition 16. (cyclic dataflow graph) Consider the graph consisting of those arcs (l, l') of $dg(\mathcal{P})$ that belong to a cycle and s.t. l' is the entry-point of a non-unitary clause. This graph is called the *cyclic dataflow graph* of $\mathcal{P}$, denoted by $cdg(\mathcal{P})$. □

The cyclic dataflow of $\mathcal{P}$ extracts the minimal information on the program which is needed to prove termination.

For two W-functions w_1, w_2, we write $w_1 \preceq w_2$ if $w_1(\alpha \wedge c) \leq w_2(\alpha)$, for every state α and constraint c.

Definition 17. (termination pair)

Let ϕ be a set of states. Let N stands for the set of nodes of $cdg(\mathcal{P})$; let $\Phi = \{\phi_l \mid l \in N\}$ be a set of assertions; and let $\mathbf{w} = \{w_l \mid l \in N\}$ be a set of W-functions. Then $(\Phi, \mathbf{w})$ is a *termination pair for* $\mathcal{P}$ *w.r.t.* ϕ if:

1. Φ is the restriction to N of an invariant for $\mathcal{P}$ w.r.t. ϕ;
2. for every $l, l' \in N$, if l and l' belong to the same clause and $l < l'$, then $w_l \succeq w_{l'}$;
3. for every arc (l, l') of $cdg(\mathcal{P})$ and α in ϕ_l, if $push(\alpha) \wedge (\overline{s}^1 = \overline{t}^0)$ is consistent then

$$w_l(\alpha) > w_{l'}(push(\alpha) \wedge (\overline{s}^1 = \overline{t}^0)),$$

where $p(\overline{t})$ is the head of the clause containing l', and $p(\overline{s}) = atom(l)$. □

The definition of termination pair uses $cdg(\mathcal{P})$ to analyze possible divergences (Point 1). Point 3 states that when a pp is reached via a resolution step **R**, then the value of the corresponding W-function decreases steadily. Point 2 deals with the other two transition rules, **C** and **S**, which do not have to increase the value of the W-functions. The notion of termination pair provides a sufficient criterion for proving termination.

Theorem 18. *A program $\mathcal{P}$ terminates w.r.t. ϕ if there is a termination pair for $\mathcal{P}$ w.r.t. ϕ.*

8.1 Negation

In this subsection we show how all the previous results can be extended to provide sufficient criteria for termination of *normal* clp's, that is clp's where body clauses may contain negated atoms $\neg A$. We suppose that negated atoms are solved using the negation as finite failure procedure or one of its modifications which allow to deal also with non-ground literals (see e.g. [AB94]).

A dataflow graph is assigned to a normal clp $\mathcal{P}$, constructed by means of the following steps:

1. consider every negated atom $\neg A$ of the program $\mathcal{P}$ as an atom A and build the dataflow graph using Definition 4;
2. delete from the graph obtained in step 1. every arc (i, j), s.t. j is the success point of a negated atom;
3. add to the graph obtained in step 2. the arcs $(i, i+1)$, for every i which is the calling point of a negated atom.

The three steps above describe the execution of a negated atom $\neg A$ as follows: the execution of A is started, and at the same time also the execution of the next literal is started. In this way, we approximate the real computation of the program, by possibly introducing extra computations, in the case that $\neg A$ would have failed. Note that this technique is also implicitly used in Wang and Shyamasundar [WS94].

Using this definition of dataflow graph, we can obtain a sound description of an invariant for $\mathcal{P}$: Theorem 15 can be restated as sufficient condition, where in case 1. a negative literal is treated as an atom (i.e. $\neg A$ is treated as A) and in case 3. it is treated as the constraint *true*. Thus, the notion of termination triple provides a sufficient criterion for termination. Also Theorem 18 can be extended to normal clp's:

Theorem 19. *A normal program $\mathcal{P}$ terminates w.r.t. ϕ if there is a termination pair for $\mathcal{P}$ w.r.t. ϕ.*

Remark. The above technique is based on the following program transformation. Consider a clause $H \leftarrow L_1, \ldots, L_{k-1}, L_k, L_{k+1}, \ldots, L_m$, where $L_k = \neg A$ is a negative literal. Split this clause as follows:

$$H \leftarrow L_1, \ldots, L_{k-1}, A, new.$$
$$H \leftarrow L_1, \ldots, L_{k-1}, L_{k+1}, \ldots, L_m.$$

where *new* is a new predicate symbol. This corresponds to the intuition that: the first clause starts the execution of A and then does not care about the computation (that is disregarded due to *new*); the second clause allows the execution continue, as if L_k had succeeded. Via repeated applications of this transformation, we can obtain from a normal clp a definite clp s.t. if this transformed program terminates then the original program terminates. □

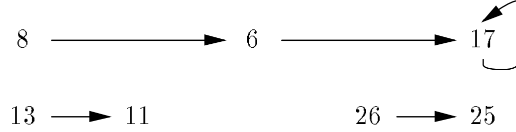
We conclude this section with an example to illustrate the application of this method.

Example 4. Consider the normal program *Fastqueen* solving in an efficient way the N-queens problem.

```
←1 fastqueens(number,solution) 2
fastqueens(num,qns) ←3 range(1,num,ns) 4 queens(ns,[],qns) 5
queens(unplqs,safeqs,qs) ←6 select(q,unplqs,unplqs1) 7
                        ¬ attack(q,safeqs) 8 queens(unplqs1,[q|safeqs],qs) 9
queens([],qs1,qs1) ←10
range(m,n,[m|ns]) ←11 m<n 12 m1=m+1 13 range(m1,n,ns) 14
```

```
range(u,u,[u]) ←15
select(x,[x|xs],xs) ←16
select(v,[y|ys],[y|zs]) ←17 select(v,ys,zs) 18
attack(w,ws) ←19 att(w,1,ws) 20
att(x1,n1,[y1|ys1]) ←21 x1=y1+n1 22
att(x2,n2,[y2|ys2]) ←23 x2+n2=y2 24
att(x3,n3,[y3|ys3]) ←25 n4=n3+1 26 att(x3,n4,ys3) 27
```

One obtains the following cyclic dataflow graph of *Fastqueens*:



Consider the precondition

$\phi = ground(number^0) \wedge \neg free(x_G^0) \wedge free(unplqs^0) \wedge free(m^0) \wedge free(v^0) \wedge free(w^0) \wedge free(x3^0)$.

We show that *Fastqueens* is terminating w.r.t. ϕ. Consider the assertions:

$\phi_6 = \phi_8 = (list(unplqs^0) \wedge list(safeqs^0))$,
$\phi_{11} = \phi_{13} = list(ns^0)$,
$\phi_{17} = list(ys^0)$,
$\phi_{25} = \phi_{26} = list(ys3^0)$.

Consider the following W-functions (here $\|\ \|$ is the 'list-length' map seen in the previous Example 3):

$w_6 = w_8 = \|unplqs^0\|$,
$w_{11} = w_{13} = \|ns^0\|$,
$w_{17} = \|ys^0\|$,
$w_{25} = w_{26} = \|ys3^0\|$.

It is not difficult to check that this is a termination pair for *Fastqueens* w.r.t. ϕ. For instance, for condition 2 of Def. 17 note that whenever two pp's of the *cdg* are on the same clause the corresponding W-functions are equal.

Thus, for Theorem 19, *Fastqueens* terminates w.r.t. ϕ. □

9 Conclusion

In this paper we have provided a characterization of terminating clp's w.r.t. a precondition by means of the notion of termination triple. We have discussed how this characterization can be used in practice, by introducing a methodology for finding termination triples, and a sufficient criterion based on this methodology for proving termination of normal clp's.

A different graphical abstraction has been used to study termination of logic programs ([BCF94, WS94]), under the name of U-graph or specific graph. This notion is based on the so-called dependency graph of a program. In an U-graph, the program atoms are the nodes and there is a directed arc from a node n_1 to

another node n_2 either if n_1 is the head of a clause and n_2 is one of its body atoms, or if n_1 is a body atom and n_2 is the head of a clause s.t. n_1 and n_2 unify. In this representation the first type of arc abstracts a clause, and the second one the flow of control. Then, the graph is used to detect possible divergences, and other proof methods ([BC89] and [DM88]) are used to obtain the information on the operational behaviour of the program which is needed to perform the termination analysis on the graph.

However, for our purpose, namely to give a characterization of terminating clp's, we found advantageous to have an uniform approach based uniquely on the dataflow graph of the program. For this reason, we have introduced a more *concrete* notion of dataflow graph, where also the backwards propagation of the state in a derivation is described.

We conclude by showing how the results can be extended to more general CLP systems.

All major implemented CLP systems are 'quick-check' and 'progressive' (cf. [JM94]). In these kind of systems, the state is divided into two components containing the *active* and the *passive* constraint, and only the consistency of the active constraint is checked. This improves the efficiency of the system. We sketch how our results can be easily extended to deal with 'quick-check' and 'progressive' systems.

$$\mathit{States} = \{(c_1, c_2) \mid c_1 \text{ and } c_2 \text{ are constraints s.t. } \mathit{consistent}(c_1)\},$$

where the test $\mathit{consistent}(c_1)$ checks for (an approximation of) the consistency of c_1.

Rules **R** and **C** are modified as below, where a state is denoted by (α_1, α_2):

$$\mathbf{R} \;\; (\langle\, p(\overline{s})\rangle \cdot \overline{A},\, \alpha\,) \longrightarrow (\,\overline{B} \cdot \langle pop\rangle \cdot \overline{A},\, \mathit{infer}(\alpha'_1, \alpha'_2 \wedge \overline{s}^1 = \overline{t}^0)\,),$$

with $\alpha' = \mathit{push}(\alpha)$, if $C = p(\overline{t}) \leftarrow \overline{B}$ is in $\mathcal{P}$.

$$\mathbf{C} \;\; (\langle\, d\,\rangle \cdot \overline{A},\, \alpha\,) \longrightarrow (\,\overline{A},\, \mathit{infer}(\alpha_1, \alpha_2 \wedge d^0)\,),$$

if d is a constraint.

Finally, the definition of $\phi \wedge c$ has to be changed in:

$$\phi \wedge c = \;\{\alpha' \in \mathit{States} \;\mid\; \alpha' = \mathit{infer}(\alpha_1, \alpha_2 \wedge c) \text{ and } \alpha \in \phi\}.$$

The operator *infer* computes from the current state (c_1, c_2) a new active constraint c'_1 and passive constraint c'_2, with the requirement that $c_1 \wedge c_2$ and $c'_1 \wedge c'_2$ are equivalent constraints. The intuition is that c_1 is used to obtain from c_2 more active constraints; then c_2 is simplified to c'_2.

Acknowledgements: We would like to thank Jan Rutten and the anonymous referees for their helpful comments. The research of the second author was partially supported by the Esprit Basic Research Action 6810 (Compulog 2).

References

[AB94] K.R. Apt and R. Bol. Logic programming and negation: a survey. *JLP* 19,20: 9-72, 1994.

[AMP94] K.R. Apt, E. Marchiori, and C. Palamidessi. A declarative approach for first-order built-in's of Prolog. *Applicable Algebra in Engineering, Communication and Computation*, 5(3/4), pp. 159-191, 1994.

[BC89] A. Bossi and N. Cocco. Verifying correctness of logic programs. *TAPSOFT*, LNCS 352, pp. 96–110, 1989.

[BCF94] A. Bossi, N. Cocco and M. Fabris. Norms on terms and their use in proving universal termination of a logic program. *TCS* 124: 297–328, 1994.

[CMM95] L. Colussi, E. Marchiori and M. Marchiori. A dataflow semantics for constraint logic programs. In *Proceedings of PLILP'95*, to appear, 1995.

[DM88] W. Drabent and J. Małuszyński. Inductive assertion method for logic programs. *TCS*, 59(1):133–155, 1988.

[DSD94] D. De Schreye and S. Decorte. Termination of logic programs: the never-ending story. *JLP* 19,20: 199-260, 1994.

[Flo67] R.W. Floyd. Assigning meanings to programs. In J.T. Schwartz, editor, *Proceedings Symposium in Applied Mathematics*, volume 19 of *Math. Aspects in Computer Science*, pages 19–32. AMS, 1967.

[JM94] J. Jaffar and M.J. Maher. Constraint Logic Programming: A Survey. *JLP* 19,20: 503-581, 1994.

[Man70] Z. Manna. Termination of Programs Represented as Interpreted Graphs. Proc. Spring. J. Comp. Conf., pp.83-89, 1970.

[Mes93] F. Mesnard. *Etude de la terminaison des programmes logiques avec contraintes aux moyens d'approximations.* PhD Thesis, Paris VI, 1993.

[WS94] B. Wang and R.K. Shyamasundar. A methodology for proving termination of logic programs. *JLP* 21(1): 1–30, 1994.

This article was processed using the LaTeX macro package with LLNCS style