Proving Termination One Loop at a Time

Michael Codish[1] and Samir Genaim[2]

[1] The Department of Computer Science
Ben-Gurion University of the Negev
Beer-Sheva, Israel
mcodish@cs.bgu.ac.il

[2] Dipartimento di Informatica
Università degli Studi di Verona
Verona, Italy
genaim@sci.univr.it

Abstract. Classic techniques for proving termination require the identification of a measure that maps program states to the elements of a well-founded domain. Termination is guaranteed if this measure is shown to decrease with each iteration of a loop in the program. This is a *global* termination condition — there exists a measure which is shown to decrease over all of the loops in the program. In recent years, systems based on *local* termination conditions are emerging. Here, termination is guaranteed if for every loop there exists a measure which decreases as execution follows through that loop. In this paper we question the relationship between the two approaches. Reasoning locally is more convenient. But is the local approach really more powerful? We show that for a large class of termination problems the two approaches are equally powerful. To this end we demonstrate that given local conditions which support a proof of termination, a corresponding global condition can always be constructed. On the one hand, the local conditions are simpler and easier to find. Yet on the other hand, in the local approach one must consider a closure operation on loops which may require to consider an exponential number of local conditions.

1 Introduction

Classic techniques for proving termination require the identification of a measure that maps program states to the elements of a well-founded domain. Termination is guaranteed if this measure is shown to decrease with each iteration of a loop in the program. Traditionally, the termination condition is *global* and loops are characterized *syntactically*. Global, because there exists a single measure which is shown to decrease over all of the loops in the program; and syntactic, because loops are defined in terms of the simple cycles in a graph representing, or derived from, the program. Namely, a path with no repeated vertices, except for the initial and terminal vertex.

In recent years, systems based on *local* termination conditions and which characterize loops *semantically* are emerging. Local, because termination is guaranteed if for every loop there exists a (possibly different) measure which decreases as execution follows through that loop; and semantic, because loops are characterized in terms of program executions. In this approach a loop corresponds to a pair of "program states" at which execution visits the same "program point".

This paper investigates the relation between the global and local approaches to proving termination. In theory, if there exist local measures to determine termination then there must exist also a global measure. This follows from the correctness of the local approach and from the completeness of the global approach. On the other hand, in practice, analyzers based on the local approach can often prove termination for programs for which analyzers based on global functions cannot. The question posed is how serious is the rift between the two approaches?

We show that for termination analyzers based on the *size change termination* principle [] or equivalently, on the description of loops in terms of *monotonicity constraints* [], the two approaches are of equal power. To this end we illustrate how to construct a global termination measure from a given finite set of local measures.

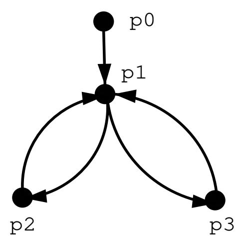
We also show that the construction does not work when loops are described in a richer language. Hence it is not clear, for the general case what is the relation between local and global termination analysis.

In this paper we consider a general setting independently of any particular programming language or paradigm. We assume some syntactic notion of a *program point* and some semantic notion of a *program state*. A program point is a node in the parse tree and represents the point just before (or after) the execution of a program statement. A program state associates values for program variables at a given program point and represents a run-time situation when execution is at the given point. A computation (or program execution) is a, possibly infinite, sequence of program states.

To clarify the terminology and to highlight the difference between the two characterizations of loops consider the following example.

Example 1. Consider the following program fragment, on the left, where p_0, p_1, p_2 and p_3 are program points. The graph, given on the right, highlights the flow of control between the program points. The simple cycles in the graph are $\langle p_1, p_2, p_1\rangle$ and $\langle p_1, p_3, p_1\rangle$. These are the syntactic loops in the program.

```
p0: int x = 1, y = 1;
p1: while (x + y > 0)
    {
      if (x > y)
p2:     { x := x - 1; y := y + 2; }
      else
p3:     { y := y - 1; x := x + 2; }
    }
```



For the program above, execution iterates returning to point p_1. The pair of program states $p_1(1,1) \rightsquigarrow p_1(3,0)$ represents an execution loop because there is a visit to p_1 with $x = 1, y = 1$ and a subsequent visit with $x = 3, y = 0$. Similarly, $p_1(3,0) \rightsquigarrow p_1(2,2)$ and $p_1(1,1) \rightsquigarrow p_1(2,2)$ are also execution loops. The set of all execution loops for the given program is infinite.

In the semantic-based approach to termination analysis, abstract interpretation is applied to provide a finite approximation of a program's (semantic) loops. A concrete semantics that is the basis for the termination analysis of logic programs is given in [4]. In this semantics the semantic objects are pairs of initial and terminal program states in a computation. The special case where the initial and terminal states are associated with the same program point indicates a semantic loop. In the corresponding abstract semantics, loops are described by *abstract binary clauses* of the form $p(\bar{x}) \leftarrow \pi, p(\bar{y})$ where $p(\bar{x})$ and $p(\bar{y})$ represent the program states at point p before and after executing the loop, and π is a constraint describing relations between the sizes of values of the variables of the states.

Termination analyzers based on abstract interpretation differ depending on the type of constraints allowed to appear in π. In [8] the authors use monotonicity constraints which are binary relations on variables. An equivalent approach is presented in [7] where the authors represent constraints using, so called, size change graphs. In such a graph, nodes correspond to the sizes of the values in the program states before and after execution of the loop, and labeled edges to binary relations on variables.

Example 2. Consider the procedure (on the left) defining the Ackerman function where four program points ($acker_0, \ldots, acker_3$) are indicated using subscripts.

```
int acker0(int m, int n)
{  /* restricted for m,n >= 0 */
   if (m == 0)
     return n+1;
   else if (n == 0)
     return acker1(m-1,1);
   else
     return acker3(m-1,acker2(m,n-1));
}
```

$$\ell_1 = \quad \mathtt{acker}_0(x_1, x_2) \leftarrow x_1 \geq 0, x_2 \geq 0, y_1 \geq 0, y_2 \geq 0, x_1 = y_1, x_2 > y_2,\ \mathtt{acker}_0(y_1, y_2).$$

$$\ell_2 = \quad \mathtt{acker}_0(x_1, x_2) \leftarrow x_1 \geq 0, x_2 \geq 0, y_1 \geq 0, y_2 \geq 0, x_1 > y_1,\ \mathtt{acker}_0(y_1, y_2).$$

Each recursive call $acker_i$, with $i > 0$ gives rise to a loop when execution returns to $acker_0$. The two abstract binary clauses (on the right) approximate the infinite number of concrete execution loops at program point $acker_0$ (for arbitrary non-negative initial values of m and n). The first abstract binary clause represents concrete loops in which the size of the first argument is invariant and the size of the second argument decreases. The second abstract binary clause represents concrete loops in which the size of the first argument is decreasing.

The following example illustrates the global and local approaches to proving termination. The formal justification of these are discussed in the next section.

Example 3. The measures $f_1(acker_0(x,y)) = y$ and $f_2(acker_0(x,y)) = x$ from Example 2 decrease respectively for the loop descriptions ℓ_1 and ℓ_2 in the example. These are local measures, one for each of the two loops. This provides a proof of termination by the local approach. A global measure can also be identified. Consider the function $f(acker_0(x,y)) = \langle f_2(acker_0(x,y)), f_1(acker_0(x,y))\rangle$ which decreases for both loops with respect to the lexicographic ordering on pairs. This provides a proof of termination by the global approach. □

Note that the local approach to proving termination is not necessarily correct when loops are represented in the more classic syntactic approach. At least not if we restrict attention only to the simple cycles in the graph.

Example 4. Consider the graph representation of the program from Example 1. The measures $f_1(p(x,y)) = x$ and $f_2(p(x,y)) = y$ decrease respectively for the two cycles in the graph. However the program does not terminate. A proof of termination in the local approach would have to consider also the non-simple loop $p_1p_2p_1p_3p_1$ for which neither x nor y decreases.. □

Reasoning locally about loops does have advantages. Local termination measures are often simpler. They are easier to identify, by the human user as well as when aiming for automated proofs. For example, there are many cases where proofs based on local measures involve linear functions while corresponding global measures involve tuples ordered lexicographically. As a consequence, analyzers implemented to use linear techniques are more successful when based on local instead of global termination conditions.

2 Preliminaries

This section presents the local approach to proving termination, recalls the classic justification for the global approach and explains why the justification of the local approach is problematic.

To simplify the presentation, we assume that the (syntactic) loops of a program P are given as abstract binary clauses, for example a program point p (where the execution iterates) is described using $p(\bar{x}) \leftarrow \pi, p(\bar{y})$, where $\bar{x}$ is the values of the program's variables (a state) when entering the loop, $\bar{y}$ is the state when revisiting the loop, and π is the size-relations between the two states. Formally speaking, (abstract) binary clauses are defined as follows.

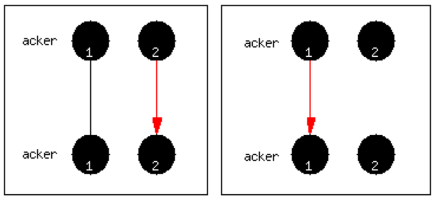
Definition 1 (Abstract binary clauses). *An abstract binary clause C is a clause of the form $p(\bar{x}) \leftarrow \pi, p(\bar{y})$ where $\bar{x}$ and $\bar{y}$ consists of distinct tuples of variables and π are linear constrains over a well-founded domain $D = \langle D, <\rangle$ that involve only the variables of $\bar{x}$ and $\bar{y}$ only. Moreover, we assume that π contains the implicit constraint $\{v \geq 0 | v \in vars(\bar{x}) \cup vars(\bar{y})\}$.*

Often, the constraint domain D that used to describe a binary clause, is further restricted to contain only *monotonicity constraints* [8]. These are conjunctions of binary relations $(<, \leq, =)$ between pairs of variables. This restriction, and in general the choice of D, is sometimes crucial to proving termination as show in the following example.

Example 5. Consider the following binary clause $p(x_1, x_2) \leftarrow \pi, p(y_1, y_2)$ where $\pi \equiv [y_1 = 2 * x_2 + 1 \wedge x_1 = 3 * y_2 + 2]$ and assume it is the only loop in the program. It is easy to verify that $f(a_1, a_2) = a_1 + 3 * a_2$ is decreasing for this binary clause, i.e $\pi \models f(y_1, y_2) > f(x_1, x_2)$, hence the program terminates. But, restricting π to *monotonicity constraints* results in $\pi' \equiv [y_1 > x_2, x_1 < y_2]$, and there is no f such that $[y_1 > x_2, x_1 < y_2] \models f(y_1, y_2) > f(x_1, x_2)$. Therefore, with this restriction we cannot prove termination.

Loop descriptions involving monotonicity constraints can also be represented as size change graphs [7] where directed edge from x to y indicates a constraint $x > y$, and non-directed edge indicates a constraint $x \geq y$.

Example 6. Consider the loop descriptions from Example 2. These will be depicted by the size change graphs.



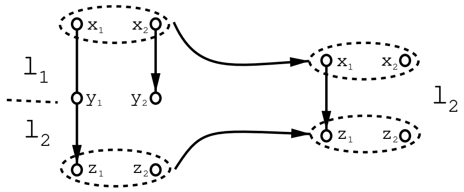
The vertices on the top of the graph correspond to those in the head of the binary clause and those on the bottom to the variables in the body of the clause.

The correctness of the local approach to proving termination relies on the fact that implicit loops are made explicit first, only then we can claim termination if we find (possibly different) decreasing measures for each of these loops. Making the implicit loops explicit is done by closing the set of (syntactic) binary clauses under compositions, where the composition of two loops is defined as follows.

Definition 2 (Composing loops). *Let $\ell_1 = p(\bar{x}) \leftarrow \pi_1, p(\bar{y})$ and $\ell_2 = p(\bar{y}) \leftarrow \pi_2, p(\bar{z})$ be (renaming of) programs loops descriptions (which do not share variables except those in $\bar{y}$). The composition of ℓ_1 and ℓ_2 is defined as $\ell_1 \circ \ell_2 = p(\bar{x}) \leftarrow \exists \bar{y}.\ \pi_1 \wedge \pi_2, p(\bar{z})$.*

The following example illustrates the composition of two loops. Note the projection of the variables (by $\exists \bar{y}$).

Example 7 (Composing loops). Consider the loop descriptions ℓ_1 and ℓ_2 from Example 2. Observe that $\ell_1 \circ \ell_1 = \ell_1$, $\ell_1 \circ \ell_2 = \ell_2 \circ \ell_1 = \ell_2$, and $\ell_2 \circ \ell_2 = \ell_2$. For example assuming that $\ell_1 = p(x_1, x_2) \leftarrow [x_1 = y_2, x_2 > y_2], p(y_1, y_2)$ and $\ell_2 = p(y_1, y_2) \leftarrow [y_1 > z_1], p(z_1, z_2)$, then the composition $\ell_1 \circ \ell_2$ results in $\ell_1 = p(x_1, x_2) \leftarrow [x_1 > z_1], p(z_1, z_2)$ (which is a variant of ℓ_2) as illustrated in the following diagram:



It is important to note that closing a set of loop descriptions under composition can introduce an exponential number of additional descriptions [7].

The following proposition provides conditions when we can claim termination based on the local approach.

Proposition 1. *Let P be a logic program, p a program point at which the execution can loop, and $\mathcal{L}_p$ a closed set (under composition) of binary clauses constructed from the syntactic loops of p, then: If for each $\ell \equiv p(\bar{x}) \leftarrow \pi, p(\bar{y}) \in \mathcal{L}_p$ there exists a function f_ℓ which maps states to a well-founded domain $D = (D, <)$ such that $\pi \models f(p(\bar{y})) < f(p(\bar{x}))$ then p terminates. The set of all such functions f_ℓ is denoted by $\mathcal{C}_p$.*

Before discussing the formal justification of the local approach, let us recall that for the more classic, global approach. The global approach is the special case where in Proposition 1 the set $\mathcal{C}_p$ consists of a single function f which decreases for all of the loop descriptions associated with program point p. The proof of correctness for this case is classic. In brief, consider that if there were to exist an infinite computation for P then there would be a program point p visited infinitely often in that computation. Denote the corresponding sequence of visits at p as

$$p^1 \leadsto p^2 \leadsto \cdots \leadsto p^k \leadsto \cdots$$

Each pair $p^i \leadsto p^{i+1}$ is a concrete loop described by one of the loop descriptions for p and the function f decreases for that pair. But f maps states to a well founded domain and cannot decrease infinitely often. Hence there cannot exist an infinite computation.

For the local approach this argument no longer holds as pairs of states $p^i \leadsto p^{i+1}$ may be associated with different functions and the pairs for which one function decreases may cause another to increase.

In some cases it is easy to work around this type of problem, for example when there is some regularity in the loops describing the corresponding transitions from program point p to program point p. For instance, if we consider a computation in which the concrete loops are described by ℓ_1 and ℓ_2 alternating as illustrated (the transitions are labeled by the corresponding loop description):

$$p \overset{\ell_1}{\leadsto} p \overset{\ell_2}{\leadsto} p \overset{\ell_1}{\leadsto} p \overset{\ell_2}{\leadsto} p \overset{\ell_1}{\leadsto} p \overset{\ell_2}{\leadsto} p \overset{\ell_1}{\leadsto} p \overset{\ell_2}{\leadsto} \cdots$$

then we also have an infinite computation of the form

$$p \overset{\ell_1 \circ \ell_2}{\leadsto} p \overset{\ell_1 \circ \ell_2}{\leadsto} p \overset{\ell_1 \circ \ell_2}{\leadsto} p \overset{\ell_1 \circ \ell_2}{\leadsto} \cdots$$

and there is a function f associated with the loop $\ell_1 \circ \ell_2$ which cannot decrease infinitely often.

But in general one cannot expect to always find such regularity, as for example illustrated by the infinite computation of the form

$$p \overset{\ell_3}{\leadsto} p \overset{\ell_1}{\leadsto} p \overset{\ell_4}{\leadsto} p \overset{\ell_1}{\leadsto} p \overset{\ell_5}{\leadsto} p \overset{\ell_9}{\leadsto} p \overset{\ell_2}{\leadsto} p \overset{\ell_6}{\leadsto} \cdots$$

where the loop descriptions correspond to the digits of the constant Π.

A formal justification of the local approach to proving termination is presented in [5]. The proof is based on an application of Ramsey's theorem [9]. We present here the proof of correctness for termination reasoning about individual loops. The proof is set as an application of Ramsey's Theorem [9]. It is essentially the same as that presented by Dershowitz *et al.*DLSS01 for logic programs in [5].

Theorem 1. *Let P be a program, and suppose we proved termination of P using the local approach as described in Proposition 1. Then P terminates.*

The proof is by contradiction as an application of Ramsey's Theorem [9].

Ramsey's Theorem: Let $\mathcal{A} = \left\{ \langle a,b \rangle \,\middle|\, a,b \in \mathbb{N} \textit{ and } a < b \right\}$, $\mathcal{C}$ be a finite set of colors and let $\mathcal{F} : \mathcal{A} \mapsto \mathcal{C}$ be a function associating the elements of $\mathcal{A}$ with colors from $\mathcal{C}$. Then, there is a color $f \in \mathcal{C}$ and an infinite set $\mathcal{X} \subseteq \mathbb{N}$ such that $\mathcal{F}(\langle a,b \rangle) = f$ for each $a,b \in \mathcal{X}$ for which $a < b$.

Proof. (of Theorem 1)
Assume in contradiction that P has an infinite computation. This implies that there is a program point p which is visited infinitely often. Denote the corresponding sequence of visits at p as

$$\mathcal{S} = \quad p^1 \leadsto p^2 \leadsto \cdots \leadsto p^k \leadsto \cdots$$

To apply Ramsey's Theorem we should fix a finite set of colors $\mathcal{C}$ and a mapping $\mathcal{F}$ of elements of $\mathcal{A} = \left\{ \langle a,b \rangle \,\middle|\, a,b \in \mathbb{N} \textit{ and } a < b \right\}$ to colors. We take as colors the set of functions $\mathcal{C}_p$ prescribed by Proposition 1. To fix the mapping $\mathcal{F}$, we note that each pair of visits p^a and p^b with $a < b$ denotes a (semantic) loop at p and is hence associated with a corresponding loop description and a function $f_i \in \mathcal{C}_p$. Ramsey's Theorem implies the existence of an infinite subsequence of $\mathcal{S}$ of the form

$$\mathcal{S}' = \quad p^{i_1} \leadsto p^{i_2} \leadsto \cdots \leadsto p^{i_k} \leadsto \cdots$$

for $\mathcal{X} = \{i_1, i_2, \ldots, i_k, \ldots\}$ such that the loop corresponding to any pair of visits p^{i_a} and p^{i_b} with $a < b$ decreases with respect to the same function $f \in \mathcal{C}_p$. But since f maps states to a well-founded domain, this is a contradiction. $\square$

3 When Local Meets Global

In this section we illustrate that for termination analysis based on monotonicity constraints (or equivalently, on the size change principle), the local and global approaches are of equivalent power. One direction of the argument is straightforward, as a global measure for termination is also a local measure. To show the other direction, we show that if we can prove termination in the local approach then we can also construct a global termination condition. This provides an alternative justification of the local approach, for the given class of case of monotonicity constraints.

Finding local functions for termination is easy. In [8] the authors show that if adding the "up arrows" in the graphs introduces a cycle (with at least one strict decrease then there exists a function. in [4], the authors add to each π the constraints $x_i < y_i$ and check that the resulting constraint is inconsistent, and in [7] the authors prove that it is sufficient to find a single down arrow in all size change graphs that are idempotent (in the closure). The question is how to find a suitable global function.

Example 8. Consider the three loop descriptions for $i \in \{1, 2, 3\}$

$$\ell_i \equiv p(x_1, x_2, x_3) \leftarrow \pi_i,\ \ p(y_1, y_2, y_3)$$

with $\pi_1 = \quad x_1 > y_1, x_2 > y1$, $\pi_2 = \quad x_1 \geq y2, x_2 \geq y_2, x_3 > y_3$ and $\pi_3 = \ x_1 \geq y2, x_2 > y_2$ where all of the variables are assumed to be non-negative. In the local approach a proof of termination is obtained choosing the following functions corresponding to the three loops: $f_1(p(x, y, z)) = x$, $f_2(p(x, y, z)) = z$ and $f_3(p(x, y, z)) = y$. In the global approach a proof is obtained choosing a global function $f(p(x, y, z)) = \langle min(x, y), y, z\rangle$ with the standard lexicographic ordering.

Definition 3 (Monotonic functions for a set of loops). *Let $\mathcal{L}_p$ be a finite set of loop descriptions with corresponding functions $\mathcal{C}_p$ mapping states to well-founded domains such that for each $\ell = \ \ p(\bar{x}) \leftarrow \pi, p(\bar{y}) \in \mathcal{L}_p$ and corresponding function $f_\ell : State \to D$ it holds that $\pi \models f(p(\bar{y})) < f(p(\bar{x}))$. We say that $\mathcal{C}_p$ is* monotonic *for $\mathcal{L}$ if for any composition $\ell = \ell_1 \circ \cdots \circ \ell_k$ of (a subset of) loops from $\mathcal{L}_p$, the function f_ℓ associated with the loop ℓ satisfies: (1) $\forall 1 \leq i \leq k.\pi_i \models f_\ell(p(\bar{x})) \geq f_\ell(p(\bar{y}))$; and (2) $\exists 1 \leq i \leq k.\pi_i \models f_\ell(p(\bar{x})) > f_\ell(p(\bar{y}))$.*

$\square$

Example 9. Consider again the three loops from Example 8.

1. The set of functions $f_1(p(x, y, z)) = x$, $f_2(p(x, y, z)) = z$ and $f_3(p(x, y, z)) = y$ is note monotonic for these three loops. Observe that $\ell_3 \circ \ell_2 \circ \ell_1 = \ell_1$ and f_1 does not decrease (even weakly) for ℓ_2.
2. The single function $f1(p(x, y, z)) = f_2(p(x, y, z)) = f_3(p(x, y, z)) = \langle min(x, y), y, z\rangle$ is monotonic for the three loops.

Theorem 2. *Let $\mathcal{C}_p$ be a monotonic set of functions for loop descriptions $\mathcal{L}$. Then there exists an ordered tuple $\langle f_1, \ldots, f_k \rangle$ of functions from $\mathcal{C}_p$ such that the function f defined by $f(s) = \langle f_1(s), \ldots, f_k(s) \rangle$ decreases globally for all of the loop descriptions in $\mathcal{L}_p$ with respect to the corresponding lexicographic order. Moreover, this ordered tuple can be constructed with time complexity $\mathcal{O}(|\mathcal{L}_p|^2)$.* □

The algorithm presented in Figure 1 constructs the required tuple of functions from $\mathcal{C}_p$ and provides the proof. The basic idea is that if we compose a *monotonic* set of local loop descriptions from $\mathcal{L}_p$ then we identify a function from $\mathcal{C}_p$ which decreases when passing through at least one of the composed loops but never increases when going through these loops. Hence this function may be placed to the right of these functions in the tuple we are constructing.

(0) $S = \mathcal{L}_p$ and $\Pi = \emptyset$
(1) WHILE ($S \neq \emptyset$) DO
(2) $\quad \ell = compose\_loops(S)$ * compose in any order *\
(3) $\quad$ IF ($\ell \in S$) THEN
(4) $\qquad \Pi = \Pi.f_\ell$
(5) $\qquad S = S \setminus \{\ell\}$
(6) $\quad$ ELSE
(7) $\qquad S = \{\ p(\bar{x}) \leftarrow \pi, p(\bar{y}) \in S \ \mid \ \pi \not\models f_\ell(p(\bar{x})) > f_\ell(p(\bar{y}))\ \}$
(8) $\qquad$ * remove from S all loops that decrease for f_ℓ *\
(9) DONE

Fig. 1. Constructing a global decreasing measure

The algorithm proceeds as follows: We start (at line 0) with $S = \mathcal{L}_p$ and an empty tuple Π. At each iteration of the while loop (lines 1-9) we compose the loops of S (line 2). If the result ℓ is an element of S then we add it to the right in the tuple Π (line 4) and remove it from S (line 5). Otherwise, if $\ell \notin S$, then (since S is monotonic) there must be at least one loop $\ell' \equiv p(\bar{x}) \leftarrow \pi, p(\bar{y}) \in S$ such that $\pi \models f_\ell(p(\bar{x})) > f_\ell(p(\bar{y}))$. This means whenever traversing a loop described by ℓ', both of the functions $f_{\ell'}$ and f_ℓ decrease. Hence, we can remove any such ℓ' from S (lines 7-8).

The correctness of the algorithm stems from the following: (1) At each step we remove at least one loop from S so the algorithm terminates; (2) The fact that S is monotonic together with lines 3-5 guarantees that whenever a function f_i (in the tuple) decreases, any function to its left either decreases or remains invariant; and (3) lines 7-8 guarantees that for each loop description there is at least one corresponding function in the tuple.

Since we remove at least one loop at each iteration, the complexity of the algorithm is $\mathcal{O}(|\mathcal{L}_p|^2)$ – we count the number of times we compose two loops (the basic step of *compose_loops*) plus the operations on sets at lines 4, 7 and 8.

Example 10. Consider the following abstract loop descriptions (where all of the variables are assumed to be non-negative).

$$\begin{array}{ll} \ell_1 = & p(x_1, x_2, x_3) \leftarrow x_1 > y_1, x_2 > y_2, p(y_1, y_2, y_3). \\ \ell_2 = & p(x_1, x_2, x_3) \leftarrow x_1 = y_1, x_3 > y_3, p(y_1, y_2, y_3). \\ \ell_3 = & p(x_1, x_2, x_3) \leftarrow x_1 > y_1, p(y_1, y_2, y_3). \end{array}$$

As local functions associated with ℓ_1, ℓ_2 and ℓ_3 we take $f_1(p(x,y,z)) = x$, $f_2(p(x,y,z)) = z$, $f_3(p(x,y,z)) = x$. The algorithm starts with $S = \{\ell_1, \ell_2, \ell_3\}$ and proceeds as follows:

1. $\ell_1 \circ \ell_2 \circ \ell_3 = \ell_3$, so we take f_3 as the first function in the tuple and remove ℓ_3 from S.
2. $\ell_1 \circ \ell_2 = \ell_3$ and ℓ_3 is not in S. We see that f_3 decreases on ℓ_1 (as well as on ℓ_3), so we remove ℓ_1 from S.
3. ℓ_2 is the only remaining loop in S so we take f_2 as the next function in the tuple and remove ℓ_2 from S.

The function $f(s) = \langle f_3(s), f_2(s) \rangle$ decreases (globally) for all of the loops in the original description. □

It still remains unclear how to find a suitable set of functions which is monotonic for the given loops. For the special case where all binary constraints are between the same arguments (x_i with y_i), the task is more simple.

Lemma 1. *If all of the constraints occurring in the loop descriptions $\mathcal{L}_p$ are monotonicity constraints of the form $x_i < y_i$ or $x_i \leq y_i$ (namely between the same argument positions) then $\mathcal{L}_p$ is a monotonic set of loops.* □

To extend the result for the full class of monotonicity constraints we need to consider all local functions of the form $min_{i \in I}(x_i)$, $max_{i \in I}(x_i)$, $sum_{i \in I}(x_i)$ for I a subset of the arguments in the state. In addition, we do not want to fix from the start a set of functions which is monotonic for the full set of loop descriptions, but rather at each stage in the algorithm of Figure 1 we need a set of functions monotonic for the loops in the set S (those which still remain in the game). Finally, consider the i^{th} iteration of the algorithm and assume that the set of functions F have already been introduced to the tuple Π and that the loop descriptions left are in S. Hence, by the construction the functions in F remain invariant for the loops in S (otherwise they would have been removed in a previous stage of the algorithm). We may take this into account when looking for the monotonic set of functions in the next iteration for S.

4 The General Case

Theorem 2 does not hold for the general case for example, when loops are described are expressed using linear constraints (but not necessarily monotonicity constraints). We illustrate this by example.

In the following example, loop descriptions are given in a richer domain allowing linear constraints. In this case it is not obvious to detect a global function with which to prove termination. Existing analyzers based on the global approach do not succeed to provide the proof.

Example 11. Consider the following three loop descriptions where the variables x_i, y_i, z_i are assumed to be non-negative:

$$\begin{aligned}
\ell_1 \equiv \; & p(x_1, y_1, z_1, k_1, l_1) \leftarrow \\
& \qquad [k_1 = k_2, l_1 = l_2, x_1 > x_2, k_1 = 0, l_1 = 0], \\
& \qquad p(x_2, y_2, z_2, k_2, l_2).
\end{aligned}$$

$$\begin{aligned}
\ell_2 \equiv \; & p(x_1, y_1, z_1, k_1, l_1) \leftarrow \\
& \qquad [k_1 = k_2, l_1 = l_2, y_1 > y_2, x_1 > x_2 + k_1, k_1 + l_1 = 0], \\
& \qquad p(x_2, y_2, z_2, k_2, l_2).
\end{aligned}$$

$$\begin{aligned}
\ell_3 \equiv \; & p(x_1, y_1, z_1, k_1, l_1) \leftarrow \\
& \qquad [k_1 = k_2, l_1 = l_2, z_1 > z_2, x_1 = x_2 + l_1, k_1 = l_1], \\
& \qquad p(x_2, y_2, z_2, k_2, l_2).
\end{aligned}$$

With the local approach we can prove termination taking $f_{\ell_1}(p(x,y,z,k,k)) = x$, $f_{\ell_2}(p(x,y,z,k,l)) = y$ and $f_{\ell_3}(p(x,y,z,k,l)) = z$. Composing any two (different) loops results in ℓ_1. So, the set of abstract binary clauses is closed under composition. But there is no permutation $\langle f_{i_1}, f_{i_2}, f_{i_3} \rangle$ such that $f(s) = \langle f_{i_1}(s), f_{i_2}(s), f_{i_3}(s) \rangle$ is guaranteed to decrease globally on all three loops. $\square$

5 Conclusion

This paper poses a question concerning the relation between the global and local approaches to proving termination. We show that in many cases we can combine local termination conditions into a tuple of functions ordered such that the tuple function decreases globally with respect to the lexicographic ordering.

While the results of the paper strengthen our belief that reasoning with local conditions has clear practical advantages, still, we lack the hard evidence that the local approach is more powerful than the global one or not. This is the effort of our ongoing research.

The very elegant result of Theorem 1 is not well known and is due to Dershowitz *et al.* [5]. It has subsequently been worked out independently by others working on the analysis of termination using local conditions as described in [3], [1] and [6].

Ben-Amram in [2] presents a transformation so that any program P for which the local approach works based on monotonicity constraints (or the size change principle) is transformed to a semantically equivalent program P^* with the same termination behavior for which termination can be identified in terms of lexicographic descent. This is the case handled by the special case of Lemma 1.

References

1. Maurice Bruynooghe, Michael Codish, John Gallagher, Samir Genaim, and Wim Vanhoof. Termination analysis through combination of type based norms, May 2003.
2. Amir M. Ben-Amram. General size-change termination and lexicographic descent. In Torben Mogensen, David Schmidt, and I. Hal Sudborough, editors, *The Essence of Computation: Complexity, Analysis, Transformation. Essays Dedicated to Neil D. Jones*, volume 2566 of *Lecture Notes in Computer Science*, pages 3–17. Springer-Verlag, 2002.
3. Michael Codish, Samir Genaim, Maurice Bruynooghe, John Gallagher, and Wim Vanhoof. One loop at a time. In *6th International Workshop on Termination (WST 2003)*, June 2003.
4. Michael Codish and Cohavit Taboch. A semantic basis for the termination analysis of logic programs. *The Journal of Logic Programming*, 41(1):103–123, 1999.
5. N. Dershowitz and N. Lindenstrauss and Y. Sagiv and A. Serebrenik. A General Framework for Automatic Termination Analysis of Logic Programs. *Applicable Algebra in Engineering, Communication and Computing*, Springer-Verlag Heidelberg, 12(1-2):117–156, 2001.
6. Chin Soon Lee. *Program Termination Analysis and Termination of Offline Partial Evaluation*. PhD thesis, University of Western Australia, August 2002.
7. Chin Soon Lee, Neil D. Jones, and Amir M. Ben-Amram. The size-change principle for program termination. *ACM SIGPLAN Notices*, 36(3):81–92, 2001.
8. N. Lindenstrauss and Y. Sagiv. Automatic termination analysis of logic programs. L. Naish, editor. *Proceedings of the Fourteenth International Conference on Logic Programming*, pages 63–77. Leuven, Belgium, 1997. The MIT Press.
9. F.P. Ramsey. On a problem of formal logic. In *Proceedings London Mathematical Society*, volume 30, pages 264–286, 1930.