A

On the Termination of Integer Loops

Amir M. Ben-Amram, The Academic College of Tel-Aviv Yaffo
Samir Genaim, Complutense University of Madrid
Abu Naser Masud, Technical University of Madrid

In this paper we study the decidability of termination of several variants of simple integer loops, without branching in the loop body and with affine constraints as the loop guard (and possibly a precondition). We show that termination of such loops is undecidable in some cases, in particular, when the body of the loop is expressed by a set of linear inequalities where the coefficients are from $\mathbb{Z} \cup \{r\}$ with r an arbitrary irrational; when the loop is a sequence of instructions, that compute either linear expressions or the step function; and when the loop body is a piecewise linear deterministic update with two pieces. The undecidability result is proven by a reduction from counter programs, whose termination is known to be undecidable. For the common case of integer linear-constraint loops with rational coefficients we have not succeeded in proving either decidability or undecidability of termination, but we show that a Petri net can be simulated with such a loop; this implies some interesting lower bounds. For example, termination for a partially-specified input is at least EXPSPACE-hard.

Categories and Subject Descriptors: F.2.0 [**Analysis of Algorithms and Problem Complexity**]: General; F.3.1 [**Logics and Meanings of Programs**]: Specifying and Verifying and Reasoning about Programs; F.4.1 [**Mathematical Logic**]: Computability theory

General Terms: Verification, Theory.

Additional Key Words and Phrases: Integer loops, Termination, Linear constraints.

ACM Reference Format:
ACM Trans. Program. Lang. Syst. V, N, Article A (January YYYY), 23 pages.
DOI = 10.1145/0000000.0000000 http://doi.acm.org/10.1145/0000000.0000000

1. INTRODUCTION

Termination analysis has received a considerable attention for at least two decades, and nowadays several powerful tools for the automatic termination analysis of different programming languages and computational models exist [Lindenstrauss et al. 1997; Giesl et al. 2004; Cook et al. 2006; Albert et al. 2007; Spoto et al. 2010; Giesl et al. 2011]. Two important aspects of termination analysis tools are their scalability and their ability to handle a large class of programs. The theoretical limits of the underlying techniques, regarding, respectively, complexity and completeness, directly affect these two aspects. Since termination of general programs is undecidable, every attempt at solving it in practice will have at its core certain restricted problems, or

This work is an extended and revised version of [Ben-Amram et al. 2012]. Part of this work was done while Amir Ben-Amram was visiting DIKU, the department of Computer Science at the University of Copenhagen. Work of Samir Genaim and Abu Naser Masud was funded in part by the Information & Communication Technologies program of the EC, Future and Emerging Technologies (FET), under the ICT-231620 *HATS* project, by the Spanish Ministry of Science and Innovation (MICINN) under the TIN-2008-05624 *DOVES* project, the UCM-BSCH-GR35/10-A-910502 *GPD* Research Group and by the Madrid Regional Government under the S2009TIC-1465 *PROMETIDOS-CM* project.

DOI 10.1145/0000000.0000000 http://doi.acm.org/10.1145/0000000.0000000

classes of programs, that the algorithm designer targets. To understand the theoretical limits of an approach, we are looking for the decidability and complexity properties of these restricted problems. Note that understanding the boundaries set by inherent undecidability or intractability of problems yields more profound information than evaluating the performance of one particular algorithm.

Much of the recent development in termination analysis has benefited from techniques that deal with one simple loop at a time, where a simple loop is specified by (optionally) some initial conditions, a loop guard, and a "loop body" of a very restricted form. Very often, the state of the program during the loop is represented by a finite set of scalar variables (this simplification may be the result of an abstraction, such as size abstraction of structured data [Lindenstrauss and Sagiv 1997; Lee et al. 2001; Bruynooghe et al. 2007; Spoto et al. 2010]).

Regarding the representation of the loop body, the most natural one is, perhaps, a block of straight-line code, namely a sequence of assignment statements, as in the following example:

$$\mathit{while}\ X > 0\ \mathit{do}\ \{\ X := X + Y;\ Y := Y - 1;\ \} \tag{1}$$

To define a restricted problem for theoretical study, one just has to state the types of loop conditions and assignments that are admitted.

By symbolically evaluating the sequence of assignments, a straight-line loop body may be put into the simple form of a simultaneous deterministic update, namely loops of the form

$$\mathit{while}\ C\ \mathit{do}\ \langle x_1, \ldots, x_n\rangle := f(\langle x_1, \ldots, x_n\rangle) \tag{2}$$

where f is a function of some restricted class. For function classes that are sufficiently simple to analyze, one can hope that termination of such loops would be decidable; in fact, the main motivation for this paper has been the remarkable results by Tiwari [2004] and Braverman [2006] on the termination of *linear loops*, a kind of loops where the update function f is linear. The loop conditions in these works are conjunctions of linear inequalities. Specifically, Tiwari proved that the termination problem is decidable for loops of the following form:

$$\mathit{while}\ (B\vec{x} > \vec{b})\ \mathit{do}\ \vec{x} := A\vec{x} + \vec{c} \tag{3}$$

where the arithmetic is done over the reals; thus the variable vector $\vec{x}$ has values in $\mathbb{R}^n$, and the constant matrices in the loop are $B \in \mathbb{R}^{m\times n}$, $A \in \mathbb{R}^{n\times n}$, $\vec{b} \in \mathbb{R}^m$ and $\vec{c} \in \mathbb{R}^n$.

Subsequently, Braverman proved decidability of termination of loops of the following form:

$$\mathit{while}\ (B_s\vec{x} > \vec{b}_s) \wedge (B_w\vec{x} \geq \vec{b}_w)\ \mathit{do}\ \vec{x} := A\vec{x} + \vec{c} \tag{4}$$

where the constant matrices and vectors are *rational*, and the variables are of either real or rational type; moreover, in the homogeneous case ($\vec{b}_s, \vec{b}_w, \vec{c} = 0$) he proved decidability when the variables range over $\mathbb{Z}$. This is a significant and non-trivial addition, since algorithmic methods that work for the reals often fail to extend to the integers (a notorious example is finding the roots of polynomials; while decidable over the reals, over the integers, it is the undecidable *Hilbert* 10^{th} *problem*[1]). Regarding the loop form (4), we note that the constant vector $\vec{c}$ may be assumed to be zero with no loss of generality, since variables can be used instead, and constrained by the loop guard to have the desired (constant) values. Over the integers it is also sufficient to have only $\geq$ or only $>$ in the loop guard. However, replacing $>$ by $\geq$ (or vice versa) alters the

[1] Over the rationals, the problem is still open, according to Matiyasevich [2000].

homogeneous loop to a non-homogeneous one, which is why including both inequality types is important in the context of Braverman [2006].

Going back to program analysis, we note that it is typical in this field to assume that some degree of approximation is necessary in order to express the effect of the loop body by linear arithmetics alone. Hence, rather than loops with a linear update as above, one defines the representation of a loop body to be a set of *constraints* (again, usually linear). The general form of such a loop is

$$\mathit{while}\ (B\vec{x} \geq \vec{b})\ \mathit{do}\ A \begin{pmatrix} \vec{x} \\ \vec{x'} \end{pmatrix} \leq \vec{c} \tag{5}$$

where the loop body is interpreted as expressing a relation between the new values $\vec{x}'$ and the previous values $\vec{x}$. Thus, in general, this representation is a non-deterministic kind of program and may over-approximate the semantics of the source program analyzed. But this is a form which lends itself naturally to analysis methods based on linear programming techniques, and there has been a series of publications on proving termination of such loops [Sohn and Gelder 1991; Podelski and Rybalchenko 2004; Mesnard and Serebrenik 2008] — all of which rely on the generation of *linear ranking functions*. For example, the termination analysis tools *Terminator* [Cook et al. 2006], *COSTA* [Albert et al. 2007], and *Julia* [Spoto et al. 2010] are based on proving termination of such loops by means of a linear ranking function.

It is known that the linear-ranking approach cannot completely resolve the problem [Podelski and Rybalchenko 2004; Braverman 2006], since not every terminating program has such a ranking function — this is the case, for example, for loop (1) above. Moreover, the linear-programming based approaches are not sensitive to the assumption that the data are integers. Thus, the problem of decidability of termination for linear-constraint loops, as loop form (5) above, stays open, in its different variants. We feel that the most intriguing problem is the following:

> Is the termination of a single linear-constraint loop decidable, when the coefficients are rational numbers and the variables range over the integers?

The problem may be considered for a given initial state, for any initial state, or for a (linearly) constrained initial state.

Our contribution. In this research, we focus on hardness proofs. Our basic tool is a new simulation of counter programs (also known as counter machines) by a simple integer loop. The termination of counter programs is a well-known undecidable problem. While we have not been able to fully answer the major open problem above, this technique led to some interesting results which improve our understanding of the simple-loop termination problem. We next summarize our main results. All concern integer variables.

(1) We prove undecidability of termination, either for all inputs or a given input, for simple loops (a variation of loop form (4)) which iterate a straight-line sequence of simple assignment instructions. The right-hand sides are integer linear expressions except for one instruction type, which computes the step function

$$f(x) = \begin{cases} 0 & x \leq 0 \\ 1 & x > 0 \end{cases}$$

At first sight it may seem like the inclusion of such an instruction is tantamount to including a branch on zero, which would immediately allow for implementing a counter program. This is not the case, because the result of the function is put into a variable which can only be combined with other variables in a very limited way. We complement this result by pointing out other natural instructions that can

be used to simulate the step function. This include integer division by a constant (with truncation towards zero) and truncated subtraction.

(2) Building upon the previous result, we prove undecidability of termination, either for all inputs or for a given input, of linear-constraint loops (a variation of loop form (5)) where *one irrational number may appear* (more precisely, the coefficients are from $\mathbb{Z} \cup \{r\}$ for an arbitrary irrational number r).

(3) Instead of moving to linear-constraint loops, we can also achieve the undecidability result with loops whose body is a deterministic update of the form "if $x > 0$ then (one linear update) else (another linear update)." Thus, the update function consists of two linear pieces. This is a non-trivial refinement of the first result, which uses the step function several times in the loop body.

(4) We observe that while linear-constraint loops (5) with rational coefficients seem to be insufficient for simulating *all* counter programs, it is possible to simulate a subclass, namely Petri nets, leading to the conclusion that termination for a given input is at least EXPSPACE-hard.

(5) Finally, we review our undecidability results and express the hardness of the corresponding problems in terms of the arithmetic and the analytic hierarchy.

We would like to highlight the relation of our results to a discussion by Braverman [2006, Section 6], where he notes that linear-constraint loops are non-deterministic and asks:

> How much non-determinism can be introduced in a linear loop with no initial conditions before termination becomes undecidable?

It is interesting that our reduction to linear-constraint loops, when using the irrational coefficient, produces constraints that are *deterministic*. The role of the constraints is not to create non-determinism; it is to express complex relationships among variables. We may also point out that some limited forms of linear-constraint loops (that are very non-deterministic since they are weaker constraints) have a *decidable* termination problem (see Section 8). Braverman also discusses the difficulty of deciding termination for a given input, a problem that he left open. Our results apply to this variant, providing a partial answer to this open problem.

The rest of this paper is organized as follows. Section 2 presents some preliminaries; Section 3 studies the termination of straight-line while loops with a "built-in" function that represents the step function; Section 4 attempts to apply the technique of Section 3 to the case of linear-constraint loops, and discusses the extension with one irrational coefficient, while Section 5 proves the result on updates with two linear pieces. Section 6 describes how a Petri net can be simulated with linear-constraint loops; Section 7 reviews our undecidability result and express the hardness of these problems in terms of the arithmetic and the analytic hierarchy; Section 8 discusses some related work; and Section 9 concludes.

2. PRELIMINARIES

In this section we define the syntax of integer piecewise linear while loops, integer linear-constraint loops, and counter programs.

2.1. Integer piecewise linear loops

An integer piecewise linear loop (IPL loop for short) with integer variables $X_1, \ldots, X_n$ is a while loop of the form

$$while\ b_1 \wedge \cdots \wedge b_m\ do\ \{c_1; \ldots; c_n\}$$

where each condition b_i is a linear inequality $a_0 + a_1 * X_1 + \cdots + a_n * X_n \geq 0$ with $a_i \in \mathbb{Z}$, and each c_i is one of the following instructions

$$X_i := X_j + X_k \mid X_i := a * X_j \mid X_i := a \mid X_i := isPositive(X_j)$$

such that $a \in \mathbb{Z}$ and

$$isPositive(X) = \begin{cases} 0 & X \leq 0 \\ 1 & X > 0 \end{cases}$$

We consider $isPositive$ to be a primitive, but in the next section we will consider alternatives. The semantics of an IPL loop is the obvious: starting from initial values for the variables $X_1, \ldots, X_n$ (the input), if the condition $b_1 \wedge \cdots \wedge b_n$ (the loop guard) holds (we say that the loop is enabled), instructions $c_1, \ldots, c_n$ are executed sequentially, and the loop is restarted at the new state. If the loop guard is false, the loop *terminates*. For simplicity, we may use composite expressions, e.g., $X_1 := 2 * X_2 + 3 * X_3 + 1$, which should be taken to be syntactic sugar for a series of assignments, possibly using temporary variables.

2.2. Integer linear-constraint loops

An integer linear-constraint loop (ILC loop for short) over n variables $\vec{x} = \langle X_1, \ldots, X_n \rangle$ has the form

$$while\ (B\vec{x} \geq \vec{b})\ do\ A \begin{pmatrix} \vec{x} \\ \vec{x'} \end{pmatrix} \leq \vec{c}$$

where for some $m, p > 0$, $B \in \mathbb{R}^{m \times n}$, $A \in \mathbb{R}^{p \times 2n}$, $\vec{b} \in \mathbb{R}^m$ and $\vec{c} \in \mathbb{R}^p$. The case we are most interested in is that in which the constant matrices and vectors are composed of rational numbers; this is equivalent to assuming that they are all integer (just multiply by a common denominator).

Semantically, a state of such a loop is an n-tuple $\langle x_1, \ldots, x_n \rangle$ of integers, and a transition to a new state $\vec{x'} = \langle x'_1, \ldots, x'_n \rangle$ is possible if $\vec{x}, \vec{x'}$ satisfy all the constraints in the loop guard and the loop body. We say that the loop terminates for a given initial state if all possible executions from that state are finite, and that it universally terminates if it terminates for every initial state. We say that the loop is *deterministic* if there is at most one successor state to any state. Note that the guard $B\vec{x} \geq \vec{b}$ is actually redundant, since its constraints can be incorporated in those of the loop body. However, we prefer to keep this form for its similarity with other loop forms studied in previous works, as well as ours (see (1)–(5) in the introduction).

2.3. Counter programs

A (deterministic) counter program P_C with n counters $X_1, \ldots, X_n$ is a list of labeled instructions $1{:}I_1, \ldots, m{:}I_m, m{+}1{:}stop$ where each instruction I_k is one of the following:

$$incr(X_j) \mid decr(X_j) \mid if\ X_j > 0\ then\ k_1\ else\ k_2$$

with $1 \leq k_1, k_2 \leq m{+}1$ and $1 \leq j \leq n$. A state is of the form $(k, \langle a_1, \ldots, a_n \rangle)$ which indicates that Instruction I_k is to be executed next, and the current values of the counters are $X_1 = a_1, \ldots, X_n = a_n$. In a valid state, $1 \leq k \leq m + 1$ and all $a_i \in \mathbb{N}$ (it will sometimes be useful to also consider invalid states, and assume that they cause a halt). Any state in which $k = m + 1$ is a halting state. For any other valid state $(k, \langle a_1, \ldots, a_n \rangle)$, the successor state is defined as follows.

— If I_k is $decr(X_j)$ (resp. $incr(X_j)$), then X_j is decreased (resp. increased) by 1 and the execution moves to label $k + 1$.

— If I_k is "*if* $X_j > 0$ *then* k_1 *else* k_2", then the execution moves to label k_1 if X_j is positive, and to k_2 if it is 0. The values of the counters do not change.

The following are known facts about the halting problem for counter programs.

THEOREM 2.1 (MINSKY [1967]). *The halting problem for counter programs with $n \geq 2$ counters and the initial state $(1, \langle 0, \ldots, 0\rangle)$ is undecidable.*

The *termination problem* is the problem of deciding whether a given program halts for every input[2]. The *Mortality* problem asks whether the program halts when started at any state (even a state that cannot be reached in a valid computation).

THEOREM 2.2 (BLONDEL ET AL. [2001]). *The mortality problem for counter programs with $n \geq 2$ counters is undecidable.*

As mentioned in the introduction, the *termination problem* usually addressed in the context of program analysis is close (or even identical) to the mortality problem, since one takes a program loop (possibly without any context) and asks whether it can be shown to halt on *every* initial state. Hence, the last theorem is useful for proving undecidability of such termination problems.

3. TERMINATION OF IPL LOOPS

In this section, we investigate the decidability of the following problems: given an IPL loop P,

(1) Does P terminate for a given input?
(2) Does P terminate for all inputs?

We show that both problems are undecidable by reduction from the halting and mortality problems, respectively, for counter programs. To see where the challenge in this reduction lies, note that the loops under consideration iterate a fixed block of straight-line code, while a counter program has a program counter that determines the next instruction to execute. While one can easily keep the value of the program counter in a variable, it is not obvious how to make the computation depend on this variable, and how to simulate branching.

3.1. The reduction

Given a counter program $P_C \equiv 1{:}I_1, \ldots, m{:}I_m, m{+}1{:}stop$ with counters $X_1, \ldots, X_n$, we generate a corresponding IPL loop $\mathcal{T}(P_C)$ as follows:

```
while ( A_1 ≥ 0 ∧ ··· ∧ A_m ≥ 0 ∧ A_1 + ··· + A_m = 1 ∧ X_1 ≥ 0 ∧ ··· ∧ X_n ≥ 0 ) do {
  N_0 := 0; N_1 := A_1; ... N_m := A_m;
  F_1 := isPositive(X_1); ... F_n := isPositive(X_n);
  T(1:I_1)
    ⋮
  T(m:I_m)
  A_1 := N_0; ... A_m := N_{m-1}
}
```

where $\mathcal{T}(k{:}I_k)$ is defined as follows

[2]We also use this term when considering a given input and the termination of all paths of a non-deterministic program.

— If $I_k \equiv incr(X_j)$, then $\mathcal{T}(k{:}I_k)$ is $X_j := X_j + A_k$;
— If $I_k \equiv decr(X_j)$, then $\mathcal{T}(k{:}I_k)$ is $X_j := X_j - A_k$;
— If $I_k \equiv \mathit{if}\ X_j > 0\ \mathit{then}\ k_1\ \mathit{else}\ k_2$, then $\mathcal{T}(k{:}I_k)$ is

$$\begin{array}{l} T_k := \mathit{isPositive}(A_k + F_j - 1); \\ R_k := \mathit{isPositive}(A_k - F_j); \\ N_k := N_k - A_k; \\ N_{k_1-1} := N_{k_1-1} + T_k; \\ N_{k_2-1} := N_{k_2-1} + R_k; \end{array}$$

Example 3.1. Consider the following 2-counter program P_C, which decrements x and y until one of them reaches 0

```
1: x=x-1
2: if x>0 then 3 else 5
3: y=y-1
4: if y>0 then 1 else 5
5: stop
```

Applying $\mathcal{T}(P_C)$ results in the following *IPL* loop

```
1  while(A_1 ≥ 0 ∧ ··· ∧ A_4 ≥ 0 ∧ A_1 + ··· + A_4 = 1 ∧ x ≥ 0 ∧ y ≥ 0) do {
2    N_0 := 0; N_1 := A_1; N_2 := A_2; N_3 := A_3; N_4 := A_4;
3    F_x := isPositive(x); F_y := isPositive(y);
4
5    x := x − A_1;
6
7    T_2 := isPositive(A_2 + F_x − 1);
8    R_2 := isPositive(A_2 − F_x);
9    N_2 := N_2 − A_2;
10   N_0 := N_0 + T_2;
11   N_4 := N_4 + R_2;
12
13   y := y − A_3;
14
15   T_4 := isPositive(A_4 + F_y − 1);
16   R_4 := isPositive(A_4 − F_y);
17   N_4 := N_4 − A_4;
18   N_0 := N_0 + T_4;
19   N_4 := N_4 + R_4;
20
21   A_1 := N_0; A_2 := N_1; A_3 := N_2; A_4 := N_3;
22 }
```

Line 5 corresponds to instruction I_1, lines 7–11 to instruction I_2, Line 13 to instruction I_3, and lines 15–19 to instruction I_4.

Let us first state, informally, the main ideas behind the reduction, and then prove it more formally.

(1) Variables $A_1, \ldots, A_m$ are flags that indicate the instruction to be executed next. They take values from $0, 1$, and only one of them can be 1 as stated by the loop guard. Note that an operation $X_j := X_j + A_k$ (resp. $X_j := X_j - A_k$) will have effect only when $A_k = 1$, and otherwise is a no-op. This is a way of simulating only one instruction in every iteration.

(2) The values of A_i are modified in a way that simulates the control of the counter machine. Namely, if $A_k = 1$, and the instruction I_k is $incr(X_j)$ or $decr(X_j)$, then the last line in the loop body sets A_{k+1} to 1 and the rest to 0. If I_k is a condition, it will set A_{k_1} or A_{k_2}, depending on the tested variable, to 1, and the rest to 0. The variables F_k, N_k, R_k, and T_k are auxiliary variables for implementing this.

LEMMA 3.2. *Let P_C be a counter program, $\mathcal{T}(P_C)$ its corresponding IPL loop, $S \equiv (k, \langle a_1, \ldots, a_n\rangle)$ a valid state for P_C, and $S_\mathcal{T}$ a state of $\mathcal{T}(P_C)$ where $A_1 = 0, \ldots, A_k = 1, \ldots, A_m = 0$, $X_1 = a_1, \ldots, X_n = a_n$. If S has a successor state $(k', \langle a'_1, \ldots, a'_n\rangle)$ in P_C, then the loop of $\mathcal{T}(P_C)$ is enabled at $S_\mathcal{T}$ and its execution leads to a state in which $A_1 = 0, \ldots, A_{k'} = 1, \ldots, A_m = 0$, $X_1 = a'_1, \ldots, X_n = a'_n$. If S is a halting configuration of P_C, the loop of $\mathcal{T}(P_C)$ is disabled at $S_\mathcal{T}$.*

PROOF. It is clear that if an execution step is possible in P_C then $0 \le k \le m$ and all X_j are non-negative, and thus the condition of the loop $\mathcal{T}(P_C)$ is true. Now note that when $A_k = 0$ the encoding of I_k does not change the value of any N_i or X_j, and consider the following two cases: (1) If I_k is $incr(X_j)$ (resp. $decr(X_j)$), then P_C increments (resp. decrements) X_j and moves to label $k' = k + 1$. Clearly the encoding of I_k increments (resp. decrements) X_j and all N_i are not modified. Since $N_k = A_k = 1$, the last line of the loop sets A_{k+1} to 1 (unless $k + 1 = m + 1$) and all other A_i to 0. (2) if I_k is *if* $X_j > 0$ *then* k_1 *else* k_2, then the counter machine moves to k_1 (resp. k_2) if $X_j > 0$ (resp. $X_j = 0$). Suppose $X_j > 0$, then $T_k = 1$ and $R_k = 0$, $N_{k_1-1} = 1$ and $N_{k_2-1} = 0$. Thus, when reaching the last line the instruction $A_{k_1} := N_{k_1-1}$ sets A_{k_1} (unless $k_1 = m + 1$). The case where $X_j = 0$ is similar. In a halting state, $k = m + 1$ which means that $A_1, \ldots, A_m = 0$. Hence, the loop is disabled. □

LEMMA 3.3. *A counter program P_C with $n \ge 2$ counters terminates for the initial state $(k, \langle a_1, \ldots, a_n\rangle)$ if and only if $\mathcal{T}(P_C)$ terminates for input $A_1 = 0, \ldots, A_k = 1, \ldots, A_m = 0$, $X_1 = a_1, \cdots, X_n = a_n$.*

PROOF. An immediate consequence of Lemma 3.2. □

Note that when values of the variables in $\mathcal{T}(P_C)$ do not correspond to a valid state for P_C, then the guard of $\mathcal{T}(P_C)$ is disabled and thus $\mathcal{T}(P_C)$ terminates for such input. This, together with Lemma 3.3, and theorems 2.1 and 2.2, imply

THEOREM 3.4. *The halting problem and the termination problem for IPL loops are undecidable.*

3.2. Examples of piecewise-linear operations

The $isPositive$ operation can easily be simulated by other natural instructions, yielding different instruction sets that suffice for undecidability.

Example 3.5 (*Integer division*). Consider an instruction that divides an integer variable by an integer constant and truncates the result towards zero (also if it is negative). Using this kind of division, we have

$$isPositive(X) = X - \frac{2 * X - 1}{2}$$

and thus, termination is undecidable for loops with linear assignments and integer division of this kind.

Example 3.6 (*Truncated subtraction*). Another common piecewise-linear function is *truncated subtraction*, such that $x \dot{-} y$ is the same as $x - y$ if it is positive, and otherwise 0. This operation allows for implementing $isPositive$ thus: $isPositive(X) = 1 \dot{-} (1 \dot{-} X)$.

4. REDUCTION TO ILC LOOPS

In this section we turn to integer linear-constraint loops. We first attempt to modify the reduction described in Section 3 to produce ILC loops in which all coefficients are rational, and explain where and why it fails. So we do not obtain undecidability for ILC loops with rational coefficients, but we show that if there is one irrational number that we are allowed to use in the constraints (any irrational will do) the reduction can be completed and undecidability of termination proved.

In Section 6 we describe another way of handling the failure of the reduction with rational coefficients only: reducing from a weaker model, and thereby proving a lower bound which is weaker than undecidability (but still non-trivial).

Observe that the loop constructed in Section 3 uses non-linear expressions only for setting the flags T_k,R_k and F_j, the rest is clearly linear. Assuming that we can encode these flags with integer linear constraints, adapting the rest of the reduction to ILC loops is straightforward: it can be done by rewriting $\mathcal{T}(P_C)$ to avoid multiple updates of a variable (that is, to *static single assignment* form) and then representing each assignment as an equation instead. Thus, in what follows we concentrate on how to represent those flags using integer linear constraints.

4.1. Encoding T_k and R_k using integer linear constraints

In Section 3, we defined T_k as $isPositive(A_k+F_j-1)$ and R_k as $isPositive(A_k-F_j)$. Since $0 \leq A_k \leq 1$ and $0 \leq F_j \leq 1$, it is easy to verify that this is equivalent to respectively imposing the constraint $A_k+F_j-1 \leq 2\cdot T_k \leq A_k+F_j$ and $A_k-F_j \leq 2\cdot R_k \leq A_k-F_j+1$.

4.2. Encoding F_j with integer linear constraints with rational coefficients

Now we discuss the difficulty of encoding the flag F_j using integer linear constraints with rational coefficients only. The following lemma states that such encoding is not possible.

LEMMA 4.1. *Given non-negative integer variables X and F, it is impossible to define a system of integer linear constraints Ψ (with rational coefficients) over X, F, and possibly other integer variables, such that $\Psi \wedge (X = 0) \rightarrow (F = 0)$ and $\Psi \wedge (X > 0) \rightarrow (F = 1)$.*

PROOF. The proof relies on a theorem by Meyer [1975] which states that the following piecewise linear function

$$f(x) = \begin{cases} 0 & x = 0 \\ 1 & x > 0, \end{cases}$$

where x is a non-negative *real* variable, cannot be defined as a minimization mixed integer programming (*MIP* for short) problem with rational coefficients only. More precisely, it is not possible to define $f(x)$ as

$$f(x) = \text{minimize } g \text{ w.r.t. } \Psi$$

where Ψ is a system of linear constraints with rational coefficients over x and other integer and real variables, and g is a linear function over $vars(\Psi)$. Now suppose that Lemma 4.1 is false, i.e., there exists Ψ such that $\Psi \wedge (X = 0) \rightarrow (F = 0)$ and $\Psi \wedge (X > 0) \rightarrow (F = 1)$, then the following *MIP* problem

$$f(x) = \text{minimize } F \text{ w.r.t. } \Psi \wedge (x \leq X)$$

defines the function $f(x)$, which contradicts the result of Meyer [1975]. □

4.3. ILC loops with an irrational constant

There are certain extensions of the ILC (with rational coefficients) model that allow our reduction to be carried out. Basically, the extensions should allow for encoding the flag F_j. The extension which we describe in this section allows the use of a single, arbitrary irrational number r (we do not require the specific value of r to represent any particular information). Thus, the coefficients are now over $\mathbb{Z} \cup \{r\}$. The variables still hold integers.

LEMMA 4.2. *Let r be an arbitrary positive irrational number, and let*

$$\Psi_1 \equiv\ (0 \le F_j \le 1) \wedge (F_j \le X)$$
$$\Psi_2 \equiv\ (rX \le B) \wedge (rY \le A) \wedge (-Y \le X) \wedge (A + B \le F_j)\,.$$

Then $(\Psi_1 \wedge \Psi_2 \wedge X = 0) \to F_j = 0$ *and* $(\Psi_1 \wedge \Psi_2 \wedge X > 0) \to F_j = 1$.

PROOF. The constraint Ψ_1 forces F_j to be 0 when X is 0, and when X is positive F_j can be either 0 or 1. The role of Ψ_2 is to eliminate the non-determinism for the case $X > 0$, namely, for $X > 0$ it forces F_j to be 1. The property that makes Ψ_2 work is that for a given *non-integer* number d, and two integers A and B, the condition $-A \le d \le B$ implies $A + B \ge 1$, whereas for an integer d the sum may be zero.

To prove the desired result, we first show that if $X = 0$, $F_j = 0$ is a solution. In fact, one can choose $B = A = Y = 0$ and all conditions are then fulfilled. Secondly, we consider $X > 0$. Note that rX is then a non-integer number, so necessarily $B > rX$. Similarly, $A > rY$, or equivalently $-A < r(-Y) \le rX$. Thus, $-A < B$, and $A + B \le F_j$ implies $0 < F_j$. Choosing $B = \lceil rX \rceil$, $Y = (-X)$ and $A = \lceil rY \rceil$ yields $A + B = 1$, so $F_j = 1$ is a solution. □

Remark: the variable Y was introduced in order to avoid using another irrational coefficient $(-r)$.

Example 4.3. Let us consider $r = \sqrt{2}$. When $X = 0$, Ψ_1 forces F_k to be 0, and it is easy to verify that Ψ_2 is satisfiable for $X = Y = A = B = F_k = 0$. Now, for the positive case, let for example $X = 5$, then Ψ_1 limits F_k to the values 0 or 1, and Ψ_2 implies $(\sqrt{2} \cdot 5 \le B) \wedge (-\sqrt{2} \cdot 5 \le A)$ since $Y \ge -5$. The minimum values that A and B can take are respectively -7 and 8, thus it is not possible to choose A and B such that $A + B \le 0$. This eliminates $F_k = 0$ as a solution. However, for these minimum values we have $A + B = 1$ and thus $A + B \le F_k$ is satisfiable for $F_k = 1$.

THEOREM 4.4. *The termination of ILC loops where the coefficients are from $\mathbb{Z} \cup \{r\}$, for a single arbitrary irrational constant r, is undecidable.*

We have mentioned, above, Meyer's result that MIP problems with rational coefficients cannot represent the step function over reals. Interestingly, he also shows that it *is* possible using an irrational constant, in a manner similar to our Lemma 4.2. Our technique differs in that we do not make use of minimization or maximization, but only of constraint satisfaction, to define the function.

5. LOOPS WITH TWO LINEAR PIECES

The reduction in Section 3 presented the loop body as a sequence of instructions that compute either linear or piecewise-linear operations. This means that the loop body, considered as a function from the entry state to the exit state, is piecewise-linear. In order to get closer to the simplest form where decidability is open, namely a body which is an affine-linear deterministic update, in Section 4 we have considered functions represented by integer linear constraints instead of affine functions. Another manner of

getting closer to the simplest case is to reduce the number of non-linearities. More precisely, we consider the update to be a function, the union of several linear pieces, and ask how many such pieces make the termination problem undecidable. Next, we improve the proof from Section 3 in this respect, reducing the usage of the *step function*. This will imply the following theorem.

THEOREM 5.1. *The halting problem and the termination problem are undecidable for loops of the following form*

$$\textit{while } (B\vec{x} \geq \vec{b}) \textit{ do } \vec{x} := \begin{cases} A_0\vec{x} & X_i \leq 0 \\ A_1\vec{x} & X_i > 0 \end{cases}$$

where the state vector $\vec{x} = \langle X_1, \ldots, X_n \rangle$ ranges over $\mathbb{Z}^n$, $A_0, A_1 \in \mathbb{Z}^{n\times n}$, $\vec{b} \in \mathbb{Z}^p$ for some $p > 0$, $B \in \mathbb{Z}^{p\times n}$, and $X_i \in \vec{x}$.

The proof is a reduction from the corresponding problems for *two-counter* machines. Recall that Minsky [1967] proved that halting for a given input is undecidable with two counters, and Blondel et al. [2001] proved it for mortality. The reduction shown in Section 3, instantiated for the case of two counters, almost establishes the result. Observe that if the values of F_1 and F_2 are known, then the flags T_k and R_k can be set to a linear function of A_k, e.g., $T_k := \mathit{isPositive}(A_k + F_1 - 1)$ can be rewritten to $T_k := A_k$ when $F_1 = 1$, and to $T_k := 0$ when $F_1 = 0$.

Thus, the body of the loop can be expressed by a linear function in each of the four regions determined by the signs of X_1 and X_2 (which define the values of F_1 and F_2). In what follows we modify the construction to reduce the four regions to only two regions.

The basic idea is to replace the two instructions $F_1 := \mathit{isPositive}(X_1)$ and $F_2 := \mathit{isPositive}(X_2)$ by the single instruction $F := \mathit{isPositive}(X_1)$, which will compute the signs of both X_1 and X_2. This is done by introducing an auxiliary iteration such that in one iteration F is set according to the sign of X_2, and in the next iteration it is set according to the sign of X_1 (by swapping the values of X_1 and X_2).

We now assume given a counter program $P_C \equiv 1{:}I_1, \ldots, m{:}I_m, m{+}1{:}\mathit{stop}$ with two counters X_1 and X_2. We first extend the set of flags A_k to range from A_1 to A_{2m}, and N_k to range from N_0 to N_{2m}. We also let $k_1, \ldots, k_i$ be indices of all instructions that perform a zero-test. Then, P_C is translated to an *IPL* loop $\mathcal{T}'(P_C)$ as follows

```
while (A_1 ≥ 0 ∧ ··· ∧ A_2m ≥ 0 ∧ A_1 + ··· + A_2m = 1 ∧ X_1 ≥ 0 ∧ X_2 ≥ 0 ∧
       0 ≤ T_k1 + R_k1 ≤ A_2k1 ∧ ··· ∧ 0 ≤ T_ki + R_ki ≤ A_2ki)
  N_0 := 0; N_1 := A_1; ... N_2m := A_2m;
  (X_2,X_1) := (X_1,X_2); // swap X_1, X_2
  F := isPositive(X_1);
  T'(1:I_1);
    ⋮
  T'(m:I_m)
  A_1 := N_0; A_2 := N_1; ... A_2m := N_2m-1
}
```

The translation $\mathcal{T}'$ of counter-program instructions follows. For increment and decrement, it is similar to what we have presented in Section 3, we only modify the indexing of the A_k variables.

— If $I_k \equiv \mathit{incr}(X_j)$, then $\mathcal{T}'(k{:}I_k)$ is $X_j := X_j + A_{2k}$

— If $I_k \equiv \mathit{decr}(X_j)$, then $\mathcal{T}'(k{:}I_k)$ is $X_j := X_j - A_{2k}$

For the conditional instruction, there are different translations for a test on X_1 and for a test on X_2:

— If $I_k \equiv \mathit{if}\ X_1 > 0\ \mathit{then}\ k_1\ \mathit{else}\ k_2$, then $\mathcal{T}'(k{:}I_k)$ is

$$\begin{aligned} T_k &:= \mathit{isPositive}(A_{2k} + F - 1);\\ R_k &:= \mathit{isPositive}(A_{2k} - F);\\ N_{2k} &:= N_{2k} - A_{2k};\\ N_{2k_1-2} &:= N_{2k_1-2} + T_k;\\ N_{2k_2-2} &:= N_{2k_2-2} + R_k; \end{aligned}$$

— If $I_k \equiv \mathit{if}\ X_2 > 0\ \mathit{then}\ k_1\ \mathit{else}\ k_2$, then $\mathcal{T}'(k{:}I_k)$ is

$$\begin{aligned} N_{2k} &:= N_{2k} - A_{2k};\\ N_{2k_1-2} &:= N_{2k_1-2} + T_k;\\ N_{2k_2-2} &:= N_{2k_2-2} + R_k;\\ T_k &:= \mathit{isPositive}(A_{2k-1} + F - 1);\\ R_k &:= \mathit{isPositive}(A_{2k-1} - F); \end{aligned}$$

Note that the above *IPL* loop can be represented in the form described in Theorem 5.1. This is because when the value of F is known, each of T_k and R_k can be set to a linear function of the corresponding A_k.

Example 5.2. Consider again the counter program of Example 3.1, and note that, in $\mathcal{T}(P_C)$, the loop body can be expressed as a four-piece linear function depending on the signs of x and y. This is because, as we have mentioned before, once the flags F_x and F_y are known, then the flags T_2, R_2, T_3 and R_3 can be defined by means of linear expressions. Applying the new transformation $\mathcal{T}'$ results in the following *IPL* loop:

```
1  while(A_1 ≥ 0 ∧ ... ∧ A_8 ≥ 0 ∧ A_1 + ··· + A_8 = 1 ∧
2                  x ≥ 0 ∧ y ≥ 0 ∧ 0 ≤ T_2 + R_2 ≤ A_4 ∧ 0 ≤ T_4 + R_4 ≤ A_8) do {
3    N_0 := 0; N_1 := A_1; ...; N_8 := A_8;
4    (y, x) := (x, y); // swap x and y
5    F := isPositive(x);
6
7    x := x − A_2;
8
9    T_2 := isPositive(A_4 + F − 1);
10   R_2 := isPositive(A_4 − F);
11   N_4 := N_4 − A_4;
12   N_4 := N_4 + T_2;
13   N_8 := N_8 + R_2;
14
15   y := y − A_6;
16
17   N_8 := N_8 − A_8;
18   N_0 := N_0 + T_4;
19   N_8 := N_8 + R_4;
20   T_4 := isPositive(A_7 + F − 1);
21   R_4 := isPositive(A_7 − F);
22
23   A_1 := N_0; A_2 := N_1; ... A_8 := N_7;
24 }
```

Line 7 corresponds to instruction I_1, lines 9–13 to instruction I_2, Line 15 to instruction I_3, and lines 17–21 to instruction I_4. Note that the body of this loop can be expressed as a two-piece linear function depending on the sign of x, since once the value of F is known, the values of T_2, R_2, T_3 and R_3 can be defined by linear expressions.

Let us explain the intuition behind the above reduction. First note that even indices for A_k represent labels in the counter program, while odd indices are used to introduce the extra iteration that computes the sign of X_2. Suppose the counter program is in a state $(k, \langle a_1, a_2 \rangle)$. To simulate one execution step of the counter program, we start the IPL loop from a state in which $A_{2k-1} = 1$ (all other A_i are 0), $X_1 = a_1$, $X_2 = a_2$, and all T_i and R_i are set to 0. Starting from this state, in the first iteration the counter variables are swapped, F is set according to the sign of X_2, and executing the encodings of all instructions is equivalent to no-op, except when I_k is a test on X_2 in which case the corresponding R_k and T_k record the result of the test. At the end of this iteration the last line of the loop body sets A_{2k} to 1. In the next iteration, the counter variables are swapped again, and F is set to the sign of X_1. Then

— if $I_k \equiv incr(X_j)$ or $I_k \equiv decr(X_j)$, then $\mathcal{T}'(I_k)$ simulates the corresponding counter-program instruction (since in such encoding we use the flag A_{2k}), and $A_{2(k+1)-1}$ is set to 1.
— if $I_k \equiv \mathit{if}\ X_1 > 0\ \mathit{then}\ k_1\ \mathit{else}\ k_2$, then $\mathcal{T}'(I_k)$, as in Section 3, sets either A_{2k_1-1} or A_{2k_2-1} to 1, i.e., it simulates a jump to k_1 or k_2.
— if $I_k \equiv \mathit{if}\ X_2 > 0\ \mathit{then}\ k_1\ \mathit{else}\ k_2$, then the first 3 lines of $\mathcal{T}'(I_k)$, together with the last line of the loop body, set either A_{2k_1-1} or A_{2k_2-1} to 1, i.e., it simulates a jump to k_1 or k_2. Note that it uses the values of T_k and R_k computed in the previous iteration. In addition, T_k and R_k are set to 0.

This basically implies that if one execution step of the counter program leads to a configuration $(k', \langle a'_1, a'_2 \rangle)$, then two iterations of the IPL loop lead to a state in which $A_{2k'-1} = 1$ (and all other A_i are 0), $X_1 = a'_1$, $X_2 = a'_2$, and all R_i and T_i are 0. Thus, with a proper initial state, we obtain a step-by-step simulation of the counter program, proving that the halting problem has been reduced correctly.

Recall that we prove undecidability of the termination problem for our loops by reducing from the mortality problem for counter programs, in which any initial configuration of the counter program is admissible. We have seen that every initial state in which only one A_{2k-1} is set to 1, for any k, and all T_k and R_k (when I_k is a test on X_2) are 0, simulates a possible state of the counter program. To establish correctness of the reduction, we should extend the argument to cover the cases that the program is started with A_{2k} set to 1, or some T_k and R_k are not 0. We refer to such states as *improper* since they do not arise in a proper simulation of the counter program.

— When A_{2k-1} is set to 1, and some T_k and R_k are not 0, the condition $0 \leq T_k + R_k \leq A_{2k}$ is false, and thus the loop is not enabled.
— When A_{2k} is set to 1, it is easy to verify that after one iteration: if I_k is increment (or decrement), then A_{2k+1} is set to 1 (unless $k = m$). If I_k is a test, then either A_{2k_1-1} or A_{2k_2-1}, or none of the A_i, is set to 1, depending on the values of T_k and R_k (at most one of them can be 1). In all cases, all T_k and R_k are set to the intended values.

We conclude that starting at an improper state either leads to immediate termination, or into a proper state. Thus, termination of the loop for all initial states reflects correctly the mortality of the counter program.

6. SIMULATION OF PETRI NETS

Let us consider a counter machine as defined in Section 2, but with a *weak* conditional statement "*if* $X_j > 0$ *then* k_1 *else* k_2" which is interpreted as: if X_j is positive then the execution may continue to *either* label k_1 or label k_2, otherwise, if it is zero, the execution *must* continue at label k_2. This computational model is equivalent to a Petri net. From considerations as those presented in Section 4, we arrived at the conclusion that the weak conditional, and therefore Petri nets, *can* be simulated by an ILC loop with rational coefficients. In this section, we describe this simulation and its implications.

A (place/transition) Petri net [Reisig 1985] is composed of a set of counters $X_1, \ldots, X_n$ (known as *places*) and a set of transitions $t_1, \ldots, t_m$. A transition is essentially a command to increment or decrement some places. This may be represented formally by associating with transition t its set of decremented places ${}^\bullet t$ and its set of incremented places $t^\bullet$. A transition is said to be enabled if all its decremented places are non-zero, and it can then be *fired*, causing the decrements and increments associated with it to take place. Starting from an initial marking (values for the places), the state of the net evolves by repeatedly firing one of the enabled transitions.

LEMMA 6.1. *Given a Petri net P with initial marking M, a simulating ILC loop (with rational coefficients) with an initial condition Ψ_M can be constructed in polynomial time, such that the termination of the loop from an initial state in Ψ_M is equivalent to the termination of P starting from M.*

PROOF. The ILC loop will have variables $X_1, \ldots, X_n$ that represent the counters in a straight-forward way, and flags $A_1, \ldots, A_m$ that represent the choice of the next transition much as we did for counter programs. The body of the loop is $\Delta \wedge \Psi \wedge \Phi$ where

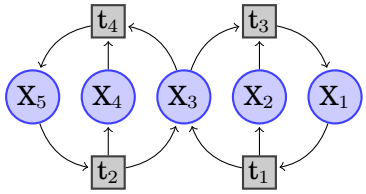
$$\Delta \equiv \bigwedge_{k=1}^{m} (A'_k \geq 0) \wedge (A'_1 + ... + A'_m = 1)$$

$$\Psi \equiv \bigwedge_{i=1}^{n} (X_i \geq \sum_{k:i \in {}^\bullet t_k} A'_k)$$

$$\Phi \equiv \bigwedge_{i=1}^{n} (X'_i = X_i - \sum_{k:i \in {}^\bullet t_k} A'_k + \sum_{k:i \in t_k{}^\bullet} A'_k)$$

The loop guard is $X_1 \geq 0 \wedge \cdots \wedge X_n \geq 0$. The initial state Ψ_M simply forces each X_i to have the value as stated by the initial marking M. Note that the initial values of A_i are not important since they are not used (we only use A'_k). As before, the constraint Δ ensures that one and only one of the A'_k will equal 1 at every iteration. The constraint Ψ ensure that A'_k may receive the value 1 only if transition k is enabled in the state. The constraint Φ (the update) simulates the chosen transition. □

Example 6.2. Consider the following Petri net



which has 5 places $X_1, \ldots, X_5$ and 4 transitions $t_1, \ldots, t_4$. The translation, as described above, of this net to an ILC loop results in

```
1  while( X1 ≥ 0 ∧ X2 ≥ 0 ∧ X3 ≥ 0 ∧ X4 ≥ 0 ∧ X5 ≥ 0 ) do {
2    A'1 ≥ 0 ∧ A'2 ≥ 0 ∧ A'3 ≥ 0 ∧ A'4 ≥ 0 ∧ A'1 + A'2 + A'3 + A'4 = 1 ∧
3
4    X1 ≥ A'1 ∧
5    X2 ≥ A'3 ∧
6    X3 ≥ A'3 + A'4 ∧
7    X4 ≥ A'4 ∧
8    X5 ≥ A'2 ∧
9
10   X'1 = X1 + A'3 − A'1 ∧
11   X'2 = X2 + A'1 − A'3 ∧
12   X'3 = X3 + A'1 + A'2 − A'3 − A'4 ∧
13   X'4 = X4 + A'2 − A'4 ∧
14   X'5 = X5 + A'4 − A'2
15 }
```

Line 2 corresponds to Δ, lines 4–8 to Ψ, and lines 10–14 to Φ.

The importance of this result lies in the fact that complexity results for Petri net are now lower bounds on the complexity of the corresponding problems for ILC loops, and in particular, from a known result about the termination problem [Esparza 1998; Lipton 1976], we obtain the following.

THEOREM 6.3. *The termination problem for ILC loops (with rational coefficients), for a given input, is at least EXPSPACE-hard.*

Note that the reduction does not provide useful information on universal termination of ILC loops with rational coefficients, since universal termination of Petri nets (also known as *structural boundedness*) is PTIME-decidable [Memmi and Roucairol 1980; Esparza and Nielsen 1994].

6.1. A lower bound for deterministic updates

In the introduction, we noted the fact that our use of constraints for the undecidability result in Section 4 did not involve non-determinism. The ILC loop we constructed to prove Theorem 6.3 *was* non-deterministic, but we will now show that the result also holds for loops which are deterministic (though defined by constraints). The result will require, however, that the loop precondition be non-deterministic, that is, we ask about termination for a set of states, not for a single state (and not for all possible states, either).

To explain the idea, we look at the Petri nets constructed in Lipton's hardness proof. This proof is a reduction from the halting problem for counter programs with a certain space bound (note that the halting problem for a space-bounded model is the canonical complete problem for a space complexity class). Given a counter program P, the reduction constructs a Petri net N_P that has the following behavior when started at an appropriate initial state. N_P has two kinds of computations, *successful* and *failing*. Failing computations are caused by taking non-deterministic branches which are not the correct choice for simulating P. Failing computations always halt. The (single) successful computation simulates P faithfully. If (and only if) P halts, the successful computation reaches a state in which a particular flag, say *HALT*, is raised (that is, incremented from 0 to 1). This flag is never raised in failing computations.

This network N_P can be translated into an ILC loop L_P as previously described. We eliminate the non-determinism from L_P by using an unconstrained input variable O as an oracle, to guide the non-deterministic choices. In addition, we reverse the program's

behaviour: our loop will terminate (on all states of interest) if and only if P does *not* terminate (note that P is presumably input-free and deterministic).

THEOREM 6.4. *The termination problem for ILC loops (with rational coefficients), for a partially-specified input, is at least EXPSPACE-hard, even if the update is deterministic.*

We describe the changes to the previous reduction. We use assignment commands for convenience. We will later show that they can all be translated into linear constraints. We assume that N_P has m transitions and n places. The construction of L_P is obtained as in the previous reduction with the following changes: (1) we introduce a new variable O, and include $O > 0$ in the loop guard; and (2) Δ is replaced by

$$\begin{aligned} PC &:= O \bmod (m+2) \\ O &:= O \operatorname{\mathbf{div}} (m+2) \\ A_k &:= [PC = k] \qquad \text{(for all } 1 \le k \le m+1\text{)} \\ O &:= O + (m+1) \cdot \textit{HALT} \end{aligned}$$

The notation $[PC = k]$ means 1 if the $PC = k$ and 0 otherwise. Also, A_{m+1} is a new flag which is not associated with any transition of N_P; it represents a do-nothing transition (the iteration does, however, decrease O).

Let Ψ_M be $HALT = 0 \wedge X_1 = a_1 \wedge \cdots \wedge X_n = a_n \wedge O > 0$ where a_i is the initial value of place X_i in M. We claim that L_P terminates for all input in Ψ_M if and only if N_P does not terminate for M (or equivalently, P does not halt).

Clearly, O guides the choice of transitions. It makes our loop deterministic, but any sequence of net transitions can be simulated: Suppose this sequence is $k_1, k_2, \ldots, k_n$. An initial value for O of $k_1 + (k_2 + (k_3 + \cdots) \cdot (m+2)) \cdot (m+2)$ will cause exactly these transitions to be taken. As long as *HALT* is not set, O also keeps descending. Since the loop condition includes $O > 0$, a non-halting simulation will become a terminating loop. A halting simulation will reach the point where *HALT* $= 1$, provided the initial value of O indicated the correct execution trace. Note that O reaches the value 0 exactly when *HALT* is set. In this iteration, only A_{m+1} is set (so counters will not be modified), while O is restored to $m+1$. In the next iteration, O remains $m+1$, A_{m+1} is set, and $HALT$ is set. Thus, the loop will not terminate.

Finally, the above assignments can be translated to integer linear constraints as follows:

$$\begin{aligned} &(O = (m+2) \cdot O'' + PC') \wedge (1 \le PC' \le m+1) \wedge \\ &(\textstyle\bigwedge_{i=1}^{m+1} A'_i \ge 0) \wedge (1 = A'_1 + \cdots + A'_{m+1}) \wedge (PC' = 1 \cdot A'_1 + \cdots + (m+1) \cdot A'_{m+1}) \wedge \\ &O' = O'' + (m+1) \cdot HALT' \,. \end{aligned}$$

7. UP THE UNSOLVABILITY HIERARCHY

In this section we will review our undecidability results, and express the hardness of these problems in terms of the arithmetic and the analytic hierarchy. This classification reveals distinctions between problems that are all undecidable: some are more undecidable than others. We cite definitions briefly, for more background see [Shoenfield 1993].

7.1. In the Arithmetic Hierarchy

Definition 7.1. Σ_1^0 is the class of decision problems that can be expressed by a formula of the form $(\exists y)P(x, y)$ where P is a recursive (decidable) predicate. This class coincides with the class RE of recursively-enumerable (aka computably enumerable)

sets. Π_2^0 is the class of decision problems that can be expressed by a formula of the form $(\forall z)(\exists y)P(x, y, z)$ with P recursive.

A standard RE-complete program is the halting problem for Turing machines, or any equivalent model. A standard Π_2^0-complete program is the termination problem for Turing machines, or any equivalent model (the problem is also known as *totality* in computability circles). Kurtz and Simon extended this result to mortality:

THEOREM 7.2 (KURTZ AND SIMON [2007]). *The mortality problem for counter programs with $n \geq 2$ counters is Π_2^0-complete.*

Using our reduction from Section 3, we obtain:

THEOREM 7.3. *The halting problem for IPL loops is RE-complete; the termination problem is Π_2^0-complete.*

PROOF. For the halting problem, RE-hardness follows from the reduction, while inclusion in RE follows from a reduction *to* Turing-machine halting (after all, an IPL loop is just a program). For termination, we get Π_2^0-completeness in the same way, using Theorem 7.2. □

The same arguments work for the loops with a two-piece-linear update as discussed in Section 5.

7.2. In the Analytic Hierarchy

The *analytic hierarchy* is obtained by considering computation with an "oracle" that is a function α from $\mathbb{N}_+$, the set of positive integers, to $\mathbb{N}_+$. The oracle can be considered a special kind of input: this input is not initially stored in a register but can be queried during the computation, using a new instruction of the form $query(X_j)$. The instruction causes $\alpha(X_j)$ to be placed in X_j. We distinguish this input by the use of the letter α. If a machine that has ordinary input $\vec{x}$ and oracle access to α decides the predicate $P(\alpha, \vec{x})$, we say that P is recursive.

Definition 7.4. Π_1^1 is the class of decision problems that can be expressed by a formula of the form $(\forall\alpha)(\exists y)P(\alpha, x, y)$ with P recursive.

A standard Π_1^1-complete program is the following variant of the halting problem:

$$TERM = \{M \mid (\forall\alpha)M^\alpha \downarrow\},$$

where M^α ranges over counter machines that do not receive any input, except for access to α. As Π_1^1 strictly contains the whole arithmetic hierarchy, Π_1^1-completeness represents a degree of unsolvability far higher than Σ_1^0 or Π_2^0 completeness.

We will prove a Π_1^1-completeness result for the halting (or termination) problem of ILC loops using a single arbitrary irrational constant—the model addressed in Theorem 4.4. However, we have first to remove a minor obstacle, the irrational constant in the constraint system. Classifying a decision problem in a computability class presumes that problem instances are finite objects. So, how is an irrational constant represented? (Perhaps the reader has already wondered about this earlier.) Our assumption is that the representation is such that inequalities involving $ar \leq b$, with $a, b \in \mathbb{Z}$, can be effectively verified, and such that at least one irrational number can be represented. Beyond that, we impose no constraints. For example, the LEDA reals [Mehlhorn and Schirra 2001], a data type that supports exact comparison of algebraic numbers, would do well.

THEOREM 7.5. *The halting problem of ILC loops where the coefficients are from $\mathbb{Z} \cup \{r\}$, for a single arbitrary irrational constant r, is Π_1^1-complete.*

PROOF. Inclusion in Π^1_1 follows from a reduction *to* $TERM$. To this end, an ILC loop is encoded as a program that queries the oracle for all values of new (tagged) variables and then verifies the constraints (which by our assumptions is an effective procedure).

Π^1_1-hardness follows by reduction from $TERM$. The idea is to simulate an oracle machine by a constraint program. First, we note that any oracle machine may be patched, if necessary, to record the history of all its oracle queries, and thereby avoid making the same query twice. We can assume that the machines in $TERM$ are so standardized. The outcome is that the oracle behaves like a completely arbitrary stream of positive integers. Thus, the value of X_j when $query(X_j)$ is performed is insignificant, and we can further patch the machine so that it actually resets X_j to zero before performing any query. This is useful for the reduction below, which is based on our translation of counter programs to ILC loops with one irrational coefficient (Section 4.3).

We need a short recap of this reduction. In Section 3, we translated a counter program to an IPL loop. For each variable X_j, representing a counter, this loop might include several assignments to X_j, specifically assignments of the form $X_j := X_j \pm A_k$. In Section 4, we assumed that these assignments are translated to constraints via a single-assignment form. Thus, for every assignment of this kind, a unique variable X_j^k is generated and the assignment is represented by the constraint $X_j^k = X_j \pm A_k$. If another assignment to X_j, say $X_j := X_j \pm A_\ell$, is found, it will be represented by $X_j^\ell = X_j^k \pm A_k$. Finally, if X_j^t is the last-occurring variable of this kind, we add $X'_j = X_j^t$.

The reduction to ILC loops only adds a simulation of the oracle to what we have done in sections 3–4. Consider a query instruction $I_q \equiv query(X_j)$. Like the assignments to X_j described above, we translate this instruction into a constraint that "sets" the variable X_j^q. As above, the constraint will equate X_j^q with a previously-defined variable, say X_j^k, plus some additive term that represents the effect of this instruction (or 0 if the instruction is not selected).

We use dedicated variables A_q, B_q, X_q^*, Y_q^* and generate the following set of constraints:

$$\begin{aligned}
\Psi_1 &\equiv\ (0 \le A_q \le 1) \wedge (A_q \le X_q^*)\\
\Psi_2 &\equiv\ (rX_q^* \le B_q) \wedge (rY_q^* \le A_q) \wedge (-Y_q^* \le X_q^*) \wedge (A_q + B_q \le A_q)\\
\Psi_3 &\equiv\ X_j^q = X_j^k + X_q^*
\end{aligned}$$

Explanation: as in Section 4.3, the constraints Ψ_1, Ψ_2 ensure that if $A_q = 0$, also X_q^* must be 0, while if $A_q = 1$, X_q^* may be any positive integer. Thus, the effect of Ψ_3 is to set the value of X_j^q to that of X_j^k plus a value which is zero if $A_q = 0$ (namely, if the current instruction is not I_q) but may be any positive value if the current instruction *is* A_q. This correctly simulates I_q. □

For termination (on all inputs), we have to use a more complex argument, since an initial state of the ILC loop does not necessarily represent an initial state of the counter program, or even a valid state. We will build on the reduction of halting to mortality by Blondel et al. [2001].

THEOREM 7.6. *The termination of ILC loops, where the coefficients are from $\mathbb{Z} \cup \{r\}$, for a single arbitrary irrational constant r, is Π^1_1-complete.*

PROOF. Π^1_1-hardness follows by reduction from $TERM$.

Suppose that we are given an input-free counter program M with n counters $R_1, \ldots, R_n$, using an oracle α, so that we are to determine if it halts for all α when computing from the standard initial state $(1, \langle 0, \ldots, 0\rangle)$. We construct a program M' with $n + 3$ counters $R_1, \ldots, R_n, V, W, A$.

The program M' is obtained from M in two stages. First, the program is modified to record all the queries it made to the oracle in the variable A. Whenever an oracle query is to be made, the machine will first check whether a query on the same argument has already been recorded, and in this case, use this result. The details of the encoding of this query history are not important, as long as any contents of A can be processed by the procedures for retrieving a query or recording a new query, so that there is no danger that a "corrupt" register would cause non-termination.

The next feature of M' is that it has a special "reset" state q_0. Each time M' enters q_0, it executes a sequence of instructions whose effect is to reset $R_1, \ldots, R_n$ to zero, store $2 * \max(1, V)$ in W and 0 in V. After having done that, it moves into state 1 (the initial state of M).

The operation of M' in the states taken from M is such that it simulates M while also performing the following operations: for every step, it increments V and decrements W. It only performs the next instruction of M if $W > 0$. If $W = 0$, it returns to the reset state.

The reader may want to reflect on why this ensures mortality (for all oracles) if and only if M halts for all oracles (or turn to [Blondel et al. 2001] for explanations).

Now M' is further translated into an ILC loop, as in the previous proof. Every initial state of the loop, that satisfies its guard, represents some configuration of M', and therefore universal termination of the loop is equivalent to mortality of M'. This completes the reduction from $TERM$ to ILC loop universal termination.

Inclusion of the problem in Π^1_1 follows from translating the ILC loop to an input-free counter program, so that universal termination of the loop is equivalent to termination of the counter program. Specifically, initial states of the ILC loop only differ on the values of the variables. So the program, which is input-free, can create the initial state by querying the oracle for values. It then proceeds with simulating the ILC loop, with the help of the oracle, as in the previous proof.

This way, the universal termination of our class of ILC loops has been reduced to $TERM$. □

We conclude this section by noting that these results confirm, after all, Braverman's supposition that the non-determinism of constraint loops should make their analysis more difficult than that of loops with deterministic updates; at least, as long one cares about degrees of unsolvability! Indeed, if we consider a class of ILC loops where the update is deterministic (that is, for any state $\vec{x}$, exactly one successor state $\vec{x}'$ is determined by the loop body), the problem falls back to the classes considered in Section 7.1.

8. RELATED WORK

Termination of integer loops has received considerable attention recently, both from theoretical (e.g., decidability, complexity), and practical (e.g., developing tools) perspectives. Research has addressed straight-line while loops as well as loops in a constraint setting, possibly with multiple paths.

For straight-line while loops, the most remarkable results are those of Tiwari [2004] and Braverman [2006]. Tiwari proved that the problem is decidable for linear deterministic updates when the domain of the variables is $\mathbb{R}$. Braverman proved that this holds also for $\mathbb{Q}$, and for the homogeneous case it holds for $\mathbb{Z}$ (see Section 1). Both considered universal termination, the termination for a given input left open.

Decidability and complexity of termination of single and multiple-path ILC loops has been intensively studied for different classes of constraints. Lee et al. [2001] proved that termination of a multiple-path ILC loop, when the constraints are restricted to size-change constraints (i.e., constraints of the form $X_i > X'_j$ or $X_i \geq X'_j$ over $\mathbb{N}$), is PSPACE-complete. Ben-Amram and Lee [2007] and Ben-Amram and Codish [2008]

identified sub-classes of such loops for which the termination can be decided in, respectively, PTIME and NPTIME. Ben-Amram [2010] extended the types of constraints allowed to *monotonicity constraints* of the form $X_i > Y$, $X_i \geq Y$, where Y can be a primed or unprimed variable. Termination for such loops is, again, PSPACE-complete. All the above results involving size-change or monotonicity constraints apply to an arbitrary well-founded domain, although the hardness results only assume $\mathbb{N}$. Monotonicity constraints over $\mathbb{Z}$ were considered by Codish et al. [2005] and by Ben-Amram [2011], concluding that this termination problem too is PSPACE-complete. Recently, Bozzelli and Pinchinat [2012] proved that it is still PSPACE-complete for gap-constraints, which are constraints of the form $X - Y \geq c$ where $c \in \mathbb{N}$. In a similar vein, Ben-Amram [2008] proved that for general difference constraints over the integers, i.e., constraints of the form $X_i - X'_j \geq c$ where $c \in \mathbb{Z}$, the termination problem becomes undecidable. However for a subclass in which each target (primed) variable might be constrained only once (in each path of a multiple-path loop) the problem is PSPACE-complete.

All the above work concerns multiple-path loops. Recently, Bozga et al. [2012] showed that (universal) termination of a single ILC loop with octagonal relations is decidable. Petri nets and various extensions, such as *reset and transfer nets*, can also be seen as multiple-path ILC loops. The termination (for a given input) of place/transition Petri nets and certain extensions is known to be decidable [Rackoff 1978; Dufourd et al. 1999].

A related topic that received much attention is the synthesis of ranking functions for such loops, as a means of proving termination. Sohn and Gelder [1991] proposed a method for the synthesis of linear ranking functions for (single path) ILC loops over $\mathbb{N}$. Later, their method was extended by Mesnard and Serebrenik [2008] to $\mathbb{Q}$ and to multiple-path loops, and completeness has also been proved. The method relies on the duality theorem of linear programming. Podelski and Rybalchenko [2004] also proposed a method for synthesizing linear ranking function for ILC loops. Their method is based on Farkas' lemma, which has been used also by Colón and Sipma [2001] for synthesizing linear ranking functions. It is important to note that these methods are complete with respect to synthesizing linear ranking functions when the variables range over $\mathbb{R}$ or $\mathbb{Q}$, but not $\mathbb{Z}$. Recently, Bagnara et al. [2012] proved that the methods of Mesnard and Serebrenik [2008] and Podelski and Rybalchenko [2004] are actually equivalent, in the sense that they compute the same set of ranking functions, and that the method of Podelski and Rybalchenko can, potentially, be more efficient since it requires solving rational constraints systems with fewer variables and constraints. Bradley et al. [2005] presented an algorithm for computing linear ranking functions for straight-line integer while loops with integer division.

Piecewise affine functions have been long used to describe the step of a discrete time dynamical system. Blondel et al. [2001] considered systems of the form $x(t+1) = f(x(t))$ where f is a piecewise affine function over $\mathbb{R}^n$ (defined by rational coefficients). They show that some problems are undecidable for $n \geq 2$, in particular, whether all trajectories go through 0 (the mortality problem). This can be seen as termination of the loop `while x` $\neq$ `0 do x := f(x)`.

9. CONCLUSION

Motivated by the increasing interest in the termination of integer loops, we have studied the hardness of termination proofs for several variants of such loops. In particular, we have considered straight-line while loops, and integer linear-constraint loops. The latter are very common in the context of program analysis.

For straight-line while loops, we have proved that if the underlying instruction set allows the implementation of a simple piecewise linear function, namely the step func-

tion, the termination problem is undecidable. For integer linear-constraint loops, we have shown that allowing the constraints to include a single arbitrary irrational number makes the termination problem undecidable. For the case of integer constraint loops with rational coefficients only, we could simulate a Petri net. This result provides interesting lower bounds on the complexity of the termination, and other related problems, of ILC loops. For example, since marking equivalence (equality of the sets of reachable states) is undecidable for Petri nets [Hack 1976; Esparza and Nielsen 1994; Jançar 1995], it follows that equivalence (in terms of the reachable states) of two ILC loops with given initial states is also undecidable, which in turn implies that the reachable states of an ILC loop are not expressible in a logic where equivalence is decidable.

We hope that our results shed some light on the termination problem of simple integer loops and perhaps will inspire further progress on the open problems.

Acknowledgments

We thank Pierre Ganty for discussions on Petri nets. We also thank William Gasarch for motivating us to add Section 7.

REFERENCES

ALBERT, E., ARENAS, P., GENAIM, S., PUEBLA, G., AND ZANARDINI, D. 2007. Costa: Design and implementation of a cost and termination analyzer for java bytecode. In *Formal Methods for Components and Objects, FMCO'07*, F. S. de Boer, M. M. Bonsangue, S. Graf, and W. P. de Roever, Eds. Lecture Notes in Computer Science Series, vol. 5382. Springer, 113–132.

BAGNARA, R., MESNARD, F., PESCETTI, A., AND ZAFFANELLA, E. 2012. A new look at the automatic synthesis of linear ranking functions. *Inf. Comput. 215*, 47–67.

BEN-AMRAM, A. M. 2008. Size-change termination with difference constraints. *ACM Trans. Program. Lang. Syst. 30,* 3.

BEN-AMRAM, A. M. 2010. Size-change termination, monotonicity constraints and ranking functions. *Logical Methods in Computer Science 6,* 3.

BEN-AMRAM, A. M. 2011. Monotonicity constraints for termination in the integer domain. *Logical Methods in Computer Science 7,* 3.

BEN-AMRAM, A. M. AND CODISH, M. 2008. A SAT-based approach to size change termination with global ranking functions. In *Tools and Algorithms for the Construction and Analysis of Systems, TACAS'08*, C. Ramakrishnan and J. Rehof, Eds. Lecture Notes in Computer Science Series, vol. 5028. Springer, 46–55.

BEN-AMRAM, A. M., GENAIM, S., AND MASUD, A. N. 2012. On the termination of integer loops. In *Verification, Model Checking, and Abstract Interpretation, VMCAI'12*, V. Kuncak and A. Rybalchenko, Eds. Lecture Notes in Computer Science Series, vol. 7148. Springer, 72–87.

BEN-AMRAM, A. M. AND LEE, C. S. 2007. Program termination analysis in polynomial time. *ACM Trans. Program. Lang. Syst. 29,* 1.

BLONDEL, V. D., BOURNEZ, O., KOIRAN, P., PAPADIMITRIOU, C. H., AND TSITSIKLIS, J. N. 2001. Deciding stability and mortality of piecewise affine dynamical systems. *Theor. Comput. Sci. 255,* 1-2, 687–696.

BOZGA, M., IOSIF, R., AND KONECNÝ, F. 2012. Deciding conditional termination. In *Tools and Algorithms for the Construction and Analysis of Systems, TACAS'12*, C. Flanagan and B. König, Eds. Lecture Notes in Computer Science Series, vol. 7214. Springer, 252–266.

BOZZELLI, L. AND PINCHINAT, S. 2012. Verification of gap-order constraint abstractions of counter systems. In *Verification, Model Checking, and Abstract Interpretation, VMCAI'12*, V. Kuncak and A. Rybalchenko, Eds. Lecture Notes in Computer Science Series, vol. 7148. Springer, 88–103.

BRADLEY, A. R., MANNA, Z., AND SIPMA, H. B. 2005. Termination analysis of integer linear loops. In *Concurrency Theory, CONCUR 2005*, M. Abadi and L. de Alfaro, Eds. Lecture Notes in Computer Science Series, vol. 3653. Springer, 488–502.

BRAVERMAN, M. 2006. Termination of integer linear programs. In *Computer Aided Verification, CAV'06*, T. Ball and R. B. Jones, Eds. Lecture Notes in Computer Science Series, vol. 4144. Springer, 372–385.

BRUYNOOGHE, M., CODISH, M., GALLAGHER, J. P., GENAIM, S., AND VANHOOF, W. 2007. Termination analysis of logic programs through combination of type-based norms. *ACM Trans. Program. Lang. Syst. 29,* 2.

CODISH, M., LAGOON, V., AND STUCKEY, P. J. 2005. Testing for termination with monotonicity constraints. In *International Conference on Logic Programming, ICLP'05*, M. Gabbrielli and G. Gupta, Eds. Lecture Notes in Computer Science Series, vol. 3668. Springer, 326–340.

COLÓN, M. AND SIPMA, H. 2001. Synthesis of linear ranking functions. In *Tools and Algorithms for the Construction and Analysis of Systems, TACAS'01*, T. Margaria and W. Yi, Eds. Lecture Notes in Computer Science Series, vol. 2031. Springer, 67–81.

COOK, B., PODELSKI, A., AND RYBALCHENKO, A. 2006. Termination proofs for systems code. In *Programming Language Design and Implementation, PLDI'06*, M. I. Schwartzbach and T. Ball, Eds. ACM, 415–426.

DUFOURD, C., JANCAR, P., AND SCHNOEBELEN, P. 1999. Boundedness of reset p/t nets. In *International Colloquium on Automata, Languages and Programming, ICALP'99*, J. Wiedermann, P. van Emde Boas, and M. Nielsen, Eds. Lecture Notes in Computer Science Series, vol. 1644. Springer, 301–310.

ESPARZA, J. 1998. Decidability and complexity of Petri net problems—an introduction. In *Lectures on Petri Nets, Vol. I: Basic Models*, W. Reisig and G. Rozenberg, Eds. Lecture Notes in Computer Science Series, vol. 1491 (Volume I). Springer-Verlag (New York), Dagstuhl, Germany, 374–428.

ESPARZA, J. AND NIELSEN, M. 1994. Decidability issues for petri nets. Tech. Rep. RS-94-8, BRICS, Department of Computer Science, University of Aarhus.

GIESL, J., RAFFELSIEPER, M., SCHNEIDER-KAMP, P., SWIDERSKI, S., AND THIEMANN, R. 2011. Automated termination proofs for haskell by term rewriting. *ACM Trans. Program. Lang. Syst. 33,* 2, 7.

GIESL, J., THIEMANN, R., SCHNEIDER-KAMP, P., AND FALKE, S. 2004. Automated termination proofs with aprove. In *Rewriting Techniques and Applications, RTA'04*, V. van Oostrom, Ed. Lecture Notes in Computer Science Series, vol. 3091. Springer, 210–220.

HACK, M. 1976. Decidability questions for Petri nets. Technical Report MIT/LCS/TR-161, Massachusetts Institute of Technology. June.

JANÇAR. 1995. Undecidability of bisimilarity for Petri nets and some related problems. *Theoretical Computer Science 148,* 2, 281–301. Selected Papers of the Eleventh Symposium on Theoretical Aspects of Computer Science.

KURTZ, S. A. AND SIMON, J. 2007. The undecidability of the generalized Collatz problem. In *Theory and Applications of Models of Computation, TAMC'07*, J.-Y. Cai, S. B. Cooper, and H. Zhu, Eds. Lecture Notes in Computer Science Series, vol. 4484. Springer, 542–553.

LEE, C. S., JONES, N. D., AND BEN-AMRAM, A. M. 2001. The size-change principle for program termination. In *Symposium on Principles of Programming Languages, POPL'01*, C. Hankin and D. Schmidt, Eds. ACM, 81–92.

LINDENSTRAUSS, N. AND SAGIV, Y. 1997. Automatic termination analysis of Prolog programs. In *International Conference on Logic Programming, ICLP'97*, L. Naish, Ed. MIT Press, 64–77.

LINDENSTRAUSS, N., SAGIV, Y., AND SEREBRENIK, A. 1997. Termilog: A system for checking termination of queries to logic programs. In *Computer Aided Verification, CAV'97*, O. Grumberg, Ed. Lecture Notes in Computer Science Series, vol. 1254. Springer, 444–447.

LIPTON, R. J. 1976. The reachability problem requires exponential space. Tech. Rep. 63, Yale University.

MATIYASEVICH. 2000. Hilbert's tenth problem: What was done and what is to be done. In *Hilbert's Tenth Problem: Relations with Arithmetic and Algebraic Geometry, AMS, 2000*, Denef, Lipshitz, Pheidas, and V. Geel, Eds.

MEHLHORN, K. AND SCHIRRA, S. 2001. Exact computation with `leda_real`—theory and geometric applications. *Symbolic Algebraic Methods and Verification Methods 379*, 163–172.

MEMMI, G. AND ROUCAIROL, G. 1980. Linear algebra in net theory. In *Net Theory and Applications*, W. Brauer, Ed. Lecture Notes in Computer Science Series, vol. 84. Springer Berlin / Heidelberg, 213–223.

MESNARD, F. AND SEREBRENIK, A. 2008. Recurrence with affine level mappings is p-time decidable for clp(r). *TPLP 8,* 1, 111–119.

MEYER, R. R. 1975. Integer and mixed-integer programming models: General properties. *Journal of Optimization Theory and Applications 16*, 191–206.

MINSKY, M. L. 1967. *Computation: finite and infinite machines*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA.

PODELSKI, A. AND RYBALCHENKO, A. 2004. A complete method for the synthesis of linear ranking functions. In *Verification, Model Checking, and Abstract Interpretation, VMCAI'04*, B. Steffen and G. Levi, Eds. Lecture Notes in Computer Science Series, vol. 2937. Springer, 239–251.

RACKOFF, C. 1978. The covering and boundedness problems for vector addition systems. *Theoretical Computer Science 6,* 2, 223–231.

REISIG, W. 1985. *Petri Nets: An Introduction*. EATCS Monographs on Theoretical Computer Science. Springer-Verlag, Berlin, Germany.

SHOENFIELD, J. R. 1993. *Recursion Theory*. Springer-Verlag, Berlin.

SOHN, K. AND GELDER, A. V. 1991. Termination detection in logic programs using argument sizes. In *Symposium on Principles of Database Systems, PODS'91*, D. J. Rosenkrantz, Ed. ACM Press, 216–226.

SPOTO, F., MESNARD, F., AND PAYET, É. 2010. A termination analyzer for java bytecode based on path-length. *ACM Trans. Program. Lang. Syst. 32,* 3.

TIWARI, A. 2004. Termination of linear programs. In *Computer Aided Verification, CAV'04*, R. Alur and D. Peled, Eds. Lecture Notes in Computer Science Series, vol. 3114. Springer, 387–390.