Disproving termination with overapproximation

Byron Cook*†, Carsten Fuhs†, Kaustubh Nimkar† and Peter O'Hearn†
*Microsoft Research
†University College London

***Abstract*—When disproving termination using known techniques (*e.g.* recurrence sets), abstractions that overapproximate the program's transition relation are unsound. In this paper we introduce *live abstractions*, a natural class of abstractions that can be combined with the recent concept of *closed recurrence sets* to soundly disprove termination. To demonstrate the practical usefulness of this new approach we show how programs with nonlinear, nondeterministic, and heap-based commands can be shown nonterminating using linear overapproximations.**

1. Introduction

A program is terminating *iff* its transition relation (when restricted to reachable states) is well-founded. Because every subrelation of a well-founded relation is itself well-founded, if we prove an abstraction that overapproximates the program to be terminating, then we have proved the concrete program terminating. The reverse, unfortunately, is not true: the existence of a nonterminating overapproximating abstraction does not imply that the original concrete program is nonterminating. Thus, when proving nontermination, we currently cannot make use of the many techniques from program analysis that overapproximate programs.

In this paper we revisit a recently introduced concept called a *closed recurrence set* [8]. The existence of a closed recurrence set for a program implies that the program does not terminate. Curiously, the existence of a closed recurrence set for an overapproximating abstraction (meeting certain restrictions, which we formalize as *live abstractions*) also implies nontermination of the original concrete program. Thus, when combined with our technique, we can now use overapproximating abstractions when attempting to prove nontermination.

To demonstrate the usefulness of our approach we describe an experimental evaluation where nonlinear, nondeterministic, and heap-based programs are proved to be nonterminating using off-the-shelf overapproximating linear abstractions.

Limitations. As discussed in detail in the paper: not all overapproximating abstractions are compatible with our approach. We address this problem by describing the conditions on abstractions that make the abstraction sound for our approach, as the notion of *live abstractions*. Many of the known abstractions indeed meet these conditions. Additionally, closed recurrence sets are not complete, *i.e.* in some cases a closed recurrence set will not exist for nonterminating programs. In these situations our approach can still help in combination with previous techniques to disprove termination (*e.g.* underapproximation) in cases where existing techniques alone could not.

In our automation, counterexamples to termination are expressed as *simple while loops*, *a.k.a.* lasso paths, which are used extensively in the termination and nontermination proving literature. Unfortunately, not all counterexamples to termination can be expressed as lassos (see *e.g.* [8, Section 4] for a program where only *aperiodic* nonterminating runs exist). Furthermore, as done in TNT [18], when disproving termination of real programs with complex control-flow graphs, we must first search for candidate lassos before applying our approach. Like TNT, our tool also exhaustively searches program's control flow graph for candidate lassos. Alternatively, candidate lassos can be obtained from a termination prover when it fails to prove termination. Thus our technique can be efficiently combined with a termination prover.

Related work. Termination proving tools are now well-known, *e.g.* [5], [6], [11], [12], [14], [15], [22], *etc.* The difference here is that we are *dis*proving, rather than proving termination. While in some trivial cases termination provers can easily disprove termination (*e.g.* when variables are not modified in an infinite loop), in practice this is not the focus for these tools. Failure to find a termination proof does not imply a proof of nontermination. Thus dedicated techniques for nontermination proving are essential.

Since termination is not a safety property, its falsification cannot always be witnessed by a finite trace; thus testing cannot reliably be used to identify termination bugs.

In recent work, Chen *et al.* [8] introduce the notion of closed recurrence sets, upon which we build in this paper. Chen *et al.* combine closed recurrence sets with counterexample-guided underapproximation to harness safety provers for proving nontermination. The method hinges on the availability of suitable safety provers for the regarded class of programs, which currently makes an application of their method to nonlinear or heap-based programs difficult. We go beyond this limitation.

Closed recurrence sets were inspired by TNT [18], which uses a characterization of nontermination by (open) recurrence sets. Note that closed recurrence sets are stronger than recurrence sets: a recurrence set exists *iff* a program is nonterminating, whereas closed recurrence sets only *imply* nontermination; that is why they are useful for approximation. We show that additional techniques can be used to mitigate the relative strength of the condition. In contrast to us, TNT is restricted to programs using linear arithmetic. Our approach supports unbounded nondeterminism in the program's transition relation, whereas TNT is restricted to deterministic commands. As discussed later, this is due to a happy interaction between the definition of closed recurrence sets and Farkas' lemma.

Larraz *et al.* [23] prove non-termination via Max-SMT solving. The method explores all strongly connected subgraphs

ISBN: 978-0-9835678-4-4.

of a program's CFG and thus can find witnesses for nontermination that need not be lasso paths. However this approach is limited to linear arithmetic as well.

Termination analysis tools for constraint-logic programs (*e.g.* [30]) can in cases be used to prove nontermination of imperative programs (*e.g.* JULIA [31] can show nontermination for Java bytecode programs if the abstraction to constraint-logic programs is exact, but provides no witness like a recurrence set to the user). The main difficulty here is in the application of the tools to imperative programs, as overapproximating abstractions are typically used for converting languages such as Java and C to constraint-logic programs. These abstractions are in general unsound for directly proving nontermination. Our work may in fact have application in this domain.

APROVE [15] uses SMT solving to prove nontermination of Java programs [7]. First nontermination of a loop regardless of its context is proved, then reachability of the loop with suitable values. The drawback of their technique is that they require either that (after program slicing to the variables that influence the loop control flow) the values of the program variables are always the same at the loop header or that the loop conditions themselves must be loop invariants.

The tool INVEL [34] analyzes nontermination of Java programs using a combination of theorem proving and invariant generation. Like Brockschmidt *et al.* [7], we were unable to obtain a working version of INVEL. Note that in the empirical evaluation by Brockschmidt *et al.* [7], the APROVE tool (which we have compared against) subsumed INVEL on INVEL's data set. Finally, INVEL is only applicable to deterministic integer programs, whereas our approach allows nondeterminism and heap-based data structures as well.

Gulwani *et al.* [17] can prove nontermination in some cases by proving the exit points of the program unreachable, but use a restriction to linear arithmetic. Their technique is fairly imprecise in the presence of nondeterminism in the input.

Atig *et al.* [2] reduce nontermination of multithreaded programs to nontermination reasoning for sequential programs. Our work complements Atig *et al.*, as we improve the underlying sequential tools that future multithreaded tools can use.

Previous works (*e.g.* [10], [19], [32]) describe techniques for proving properties expressed in branching-time temporal logic of infinite-state programs. Nontermination can be encoded in these logics (*e.g.* in CTL, nontermination is $\mathsf{EG}\ pc \neq \mathsf{END}$). Our work complements these previous works. Here we facilitate the use of overapproximation.

Finally, several automatic tools exist for proving nontermination of term rewrite systems (*e.g.* [13], [16], [29]). However, in nontermination analysis for term rewriting the *entire* state space is considered as legitimate initial states for a (possibly infinite) evaluation sequence, whereas our setting also factors in reachability from the initial states.

2. ILLUSTRATING EXAMPLE

Before formally introducing our approach, we first describe the idea informally using an example. Imagine that we want to show nontermination of the toy program in Fig. 1(a). Here we are using an **assume** statement [27], which does not allow executions to pass unless the condition is valid.[1]

```
assume(j ≥ 1 and k ≥ 1);
while i ≥ 0 do
    i := j × k;
    j := j + 1;
    k := k + 1;
    skip; // location ℓ
done
```

(a)

```
assume(j ≥ 1 and k ≥ 1);
while i ≥ 0 do
    i := nondet();
    j := j + 1;
    k := k + 1;
    assume(i ≥ 1);
done
```

(b)

Fig. 1. Nonlinear program **(a)**, and its linear abstraction **(b)**. The command **assume** [27] only allows executions to continue when the condition holds, **nondet** represents nondeterministic choice.

We are looking to find initial values for i, j and k from which an infinite run is possible. Indeed, such a run is possible: from the state $(\mathsf{i} = 1, \mathsf{j} = 1, \mathsf{k} = 1)$ the program can perform a sequence of loop iterations via the states $(\mathsf{i} = 1, \mathsf{j} = 2, \mathsf{k} = 2)$, $(\mathsf{i} = 4, \mathsf{j} = 3, \mathsf{k} = 3)$, $(\mathsf{i} = 9, \mathsf{j} = 4, \mathsf{k} = 4)$, ... leading to an infinite run. This set of states $\mathcal{G} = \{(\mathsf{i} = 1, \mathsf{j} = 1, \mathsf{k} = 1), (\mathsf{i} = 1, \mathsf{j} = 2, \mathsf{k} = 2)\}, (\mathsf{i} = 4, \mathsf{j} = 3, \mathsf{k} = 3), (\mathsf{i} = 9, \mathsf{j} = 4, \mathsf{k} = 4), \ldots\}$ meets a criterion that defines a recurrence set [18]: during the execution of the **while** loop the program can get into the set of states $\mathcal{G}$, and when in $\mathcal{G}$ it is possible to stay in $\mathcal{G}$ during an iteration of the loop. Finding a valid recurrence set such as this is a complete method of proving nontermination.

Now the question is, how can we *automatically* find such a proof of nontermination? The difficulty here is the nonlinear assignment $\mathsf{i} := \mathsf{j} \times \mathsf{k}$: most automatic formal verification techniques struggle to support nonlinear arithmetic in a scalable fashion. An arbitrary overapproximation of this program will not help in this context. The problem is that if we prove nontermination of the overapproximation we still have not proved nontermination of the original concrete program. The reason is that—due to the nature of overapproximation—a nonterminating execution in the overapproximation need not correspond to any execution in the concrete program.

To avoid this problem we can use an overapproximating abstraction of our program such that the abstraction satisfies certain conditions. We call such an abstraction a live abstraction. See Section 3. Such an abstraction is shown in Fig. 1(b). This abstraction uses nondeterministic choice (*i.e.* **nondet**) to abstract away the nonlinear command and also uses a linear location invariant at location ℓ from the original program ($\mathsf{i} \geq 1$). Note that in Fig. 1(b) we do not alter the loop condition from the original program but only overapproximate the transitions that can take place inside the loop. This abstraction is a live abstraction and is thus a safe abstraction for our approach. Later in Section 3 we give the necessary conditions for an abstraction to be a live abstraction. Most of the abstractions used in the termination literature satisfy the properties of a live abstraction.

Our approach is based on the following insight: if we can

[1] For termination, we can encode **assume**$(e) \equiv$ **if** $\neg e$ **then exit**(); **fi**

```
assume(j ≥ 1 and k ≥ 1);
while i ≥ 0 and m ≥ 0 do
    i := j × k;
    j := j + 1;
    k := k + 1;
    m := nondet();
done
```

(a)

```
assume(j ≥ 1 and k ≥ 1);
while i ≥ 0 and m ≥ 0 do
    i := j × k;
    j := j + 1;
    k := k + 1;
    m := nondet();
    assume(m ≥ 0);
done
```

(b)

```
assume(j ≥ 1 and k ≥ 1);
while i ≥ 0 and m ≥ 0 do
    i := nondet();
    j := j + 1;
    k := k + 1;
    m := nondet();
    assume(m ≥ 0);
    assume(i ≥ 1);
done
```

(c)

Fig. 2. Nonlinear program **(a)**, its underapproximation **(b)**, and the resulting linear abstraction **(c)**.

prove existence of a set of states $\mathcal{G}$ at the loop head in the live abstraction meeting the following conditions then we know that both the abstraction and the original concrete program are nonterminating: **a)** $\mathcal{G}$ is nonempty and at least one state in $\mathcal{G}$ is reachable, **b)** every state in $\mathcal{G}$ has at least one transition, and **c)** all transitions from $\mathcal{G}$ in the abstraction only lead to $\mathcal{G}$. If these conditions hold then $\mathcal{G}$ is a closed recurrence set. This now allows us to use tools on the overapproximating abstraction rather than the original program to establish nontermination. Here such a set could be given by $\mathcal{G} = \{s \mid s \models \mathsf{i} \geq 1\}$.

Combining over- and underapproximation. Sometimes closed recurrence sets are alone not enough: we may still require the use of underapproximation. However, even then, our approach facilitates the *mixture* of over- and underapproximation to make more powerful nontermination proving tools.

Consider the program in Fig. 2(a). Here it is difficult to find a useful linear overapproximation directly because of the nondeterministic assignment to the variable m. However if an underapproximation of a program is nonterminating, then the original program itself is nonterminating as well. Here we can use known techniques to automatically find an underapproximation that rules out the unwanted transitions. Consider the program in Fig. 2(b), an underapproximation of the program in Fig. 2(a) restricting the choice for nondeterministic assignment to the variable m. Using our approach we can now easily find a useful linear overapproximation that is a live abstraction for this program. The program in Fig. 2(c) is a linear *over*approximation of the *under*approximation in Fig. 2(b). Here, we can find a closed recurrence set $\mathcal{G} = \{s \mid s \models \mathsf{i} \geq 1 \wedge \mathsf{m} \geq 0\}$ for the program in Fig. 2(c), which proves nontermination of the program in Fig. 2(b), which in turn proves nontermination of the program in Fig. 2(a). Note that it is unsound to first overapproximate and then underapproximate: as in this example we must first underapproximate and then overapproximate. Also note that for overapproximations we only consider live abstractions.

3. Closed recurrence sets and overapproximation

In this section we discuss *closed recurrence sets* and their relationship to overapproximation.

Transition Systems. A *transition system* (S, R, I, F) is defined by a set of states S, a *transition relation* $R \subseteq S \times S$, a set of *initial states* $I \subseteq S$ and a set of *final states* $F \subseteq S$. For a state s with $R(s, s')$, we say that s' is a *post-state* of s and that s is a *pre-state* of s'. We also call s' a *successor of* s *under* R. Execution of a transition system can only halt in a final state, so every state $s \notin F$ must have a successor under R, and any final state $f \in F$ has no successors under R.

Example. Consider the example in Fig. 1(a). We can describe the loop and its initial condition as a transition system (S, R, I, F) where any state s is basically a tuple $(\mathsf{i}, \mathsf{j}, \mathsf{k})$ of values of variables and $S = \mathbb{Z}^3$, $R = \{(s, s') \mid s, s' \vDash \mathsf{i} \geq 0 \wedge \mathsf{i}' = \mathsf{j} \times \mathsf{k} \wedge \mathsf{j}' = \mathsf{j}+1 \wedge \mathsf{k}' = \mathsf{k}+1\}$, $I = \{s \mid s \vDash \mathsf{j} \geq 1 \wedge \mathsf{k} \geq 1\}$, $F = \{s \mid s \vDash \mathsf{i} < 0\}$.

A. Closed recurrence sets.

A transition system (S, R, I, F) is nonterminating *iff* there exists an infinite transition sequence $s_0 \xrightarrow{R} s_1 \xrightarrow{R} s_2 \xrightarrow{R} \ldots$ with $s_0 \in I$. Gupta *et al.* [18] characterize nontermination of a relation R by the existence of a *recurrence set*, *viz.* a nonempty set of states $\mathcal{G}$ such that for each $s \in \mathcal{G}$ there exists a transition to some $s' \in \mathcal{G}$. Here we extend the notion of a recurrence set to transition systems. A transition system (S, R, I, F) has a *recurrence set* (or *open recurrence set*) of states $\mathcal{G}(s)$ *iff*

$$\exists s.\mathcal{G}(s) \wedge I(s), \tag{1}$$

$$\forall s \exists s'.\mathcal{G}(s) \rightarrow R(s, s') \wedge \mathcal{G}(s'). \tag{2}$$

A transition system (S, R, I, F) is nonterminating *iff* it has a recurrence set of states.

Quantifier alternation as in Condition (2) can be a headache for automation. To avoid this problem Gupta *et al.* [18] restrict the transition relation to deterministic programs only. In this case we can represent the post-state s' using a unique expression in terms of the pre-state s. Thus the existential quantifier can be eliminated by instantiating it with this expression.

Example. For the loop from Fig. 1(a), we can have a recurrence set $\mathcal{G} = \{(\mathsf{i} = 1, \mathsf{j} = 1, \mathsf{k} = 1), (\mathsf{i} = 1, \mathsf{j} = 2, \mathsf{k} = 2), (\mathsf{i} = 4, \mathsf{j} = 3, \mathsf{k} = 3), (\mathsf{i} = 9, \mathsf{j} = 4, \mathsf{k} = 4), \ \ldots\}$.

Definition (Closed Recurrence Set [8])
A set $\mathcal{G}$ *is a* closed recurrence set *for a transition system* (S, R, I, F) iff *the following three conditions hold:*

$$\exists s.\mathcal{G}(s) \wedge I(s) \tag{3}$$

$$\forall s \exists s'.\mathcal{G}(s) \rightarrow R(s, s') \tag{4}$$

$$\forall s \forall s'.\mathcal{G}(s) \wedge R(s, s') \rightarrow \mathcal{G}(s') \tag{5}$$

In contrast to standard (open) recurrence sets, we now require a purely universal property: for each $s \in \mathcal{G}$ and *for each* of its successors s', also s' must be in the recurrence set (Condition (5)). So instead of requiring that we *can* stay in the recurrence set, we now demand that we *must* stay in the recurrence set. This has several advantages. First, without quantifier alternation, Farkas' lemma can now be applied directly. This now helps us to incorporate nondeterministic transition systems too. Secondly, the interaction with overapproximation is improved. The downside is that the condition can be too strong.

There is an additional problem: what if a state s in our recurrence set $\mathcal{G}$ has no successor s' at all? This would bring our alleged infinite transition sequence to a sudden halt, yet our universal formula would trivially hold. To deal with this issue, we must impose that each $s \in \mathcal{G}$ has *some* successor s' (Condition (4)). But this existential statement need not mention that s' must be in $\mathcal{G}$ again—our previous *universal* statement already takes care of this. In this way, we have gained something: the existential quantifier in Condition (4) refers only to the (known) transition relation R and, as we shall see in the Section 5 on automation, the condition can be easily automated in spite of quantifier alternation when we search for a closed recurrence set $\mathcal{G}$.

Example. For the loop from Fig. 1(b), we can have a closed recurrence set $\mathcal{G} = \{s \mid s \vDash \mathsf{i} \geq 1\}$. $\mathcal{G}$ satisfies all the conditions of a closed recurrence set.

Theorem (Closed Recurrence Sets are Recurrence Sets [8]) *Let $\mathcal{G}$ be a closed recurrence set for (S, R, I, F). Then $\mathcal{G}$ is also a standard (open) recurrence set for (S, R, I, F).*

If our transition system is *deterministic*, every recurrence set is also a closed recurrence set. In particular, closed recurrence sets characterize nontermination in the setting of Gupta *et al.* [18], which assumes deterministic programs.

Corollary (Recurrence Sets are Closed Recurrence Sets for Deterministic Transition Systems)
Let $\mathcal{G}$ be a recurrence set for (S, R, I, F) such that for every state s there exists at most one state s' with $R(s, s')$. Then $\mathcal{G}$ is also a closed recurrence set for (S, R, I, F).

B. Live abstractions

We now describe generic conditions on abstractions that are sufficient to establish soundness for nontermination proving using our approach, in the form of *live abstractions*.

Live Abstractions. We assume that an abstraction of $T = (S, R, I, F)$ is a system $T^\alpha = (S^\alpha, R^\alpha, I^\alpha, F^\alpha)$, with a concretion (or meaning) function $[\![\cdot]\!] : S^\alpha \to \mathcal{P}(S)$.

Definition (Live Abstraction)
An abstraction $T^\alpha = (S^\alpha, R^\alpha, I^\alpha, F^\alpha)$ is live *iff*

$\forall s \forall s' \forall a.\, R(s, s') \wedge s \in [\![a]\!] \to \exists a'.R^\alpha(a, a') \wedge s' \in [\![a']\!]$
(Simulation)

$\forall f \forall g.\, f \in F \wedge f \in [\![g]\!] \to g \in F^\alpha$
(Upward Termination)

The *Simulation* (or, 'up simulation') condition is a standard one for overapproximation: it says that any steps you can take in the concrete transition system can be overapproximated in the abstract transition system. The *Upward Termination* condition says that for every final state in the concrete transition system, any corresponding abstract state is also a final state in the abstract transition system. Together *Simulation* and *Upward Termination* imply that for every terminating run in the concrete transition system, also any corresponding run in the abstract transition system is terminating.

The connection of these conditions to disproving termination then is: if there is an initial state a_0 from which all computations in the abstract program are nonterminating and there is an initial state s_0 in the concrete program such that $s_0 \in [\![a_0]\!]$, then all computations in the concrete program starting from s_0 are nonterminating (*i.e.*, for live abstractions, *closed* recurrence carries over from the abstract to the concrete).

Theorem (Soundness)
Consider a live abstraction $(S^\alpha, R^\alpha, I^\alpha, F^\alpha)$ for a transition system (S, R, I, F). Suppose $\mathcal{G}^\alpha$ is a closed *recurrence set for $(S^\alpha, R^\alpha, I^\alpha, F^\alpha)$ and for some a_0 we have $\mathcal{G}^\alpha(a_0) \wedge I^\alpha(a_0) \wedge \exists s_0.(\, s_0 \in [\![a_0]\!] \wedge I(s_0)\,)$. Then there also exists a closed recurrence set $\mathcal{G} = \{s \mid \exists a.\, \mathcal{G}^\alpha(a) \wedge s \in [\![a]\!]\}$ for (S, R, I, F).*

Proof. We need to prove Conditions (3), (4), and (5) for $\mathcal{G}$.

For Condition (3) for $\mathcal{G}$: We have for some a_0, $\mathcal{G}^\alpha(a_0) \wedge I^\alpha(a_0) \wedge \exists s_0.(\, s_0 \in [\![a_0]\!] \wedge I(s_0)\,)$. Thus for such s_0 we have $I(s_0)$ and the definition of $\mathcal{G}$ implies $\mathcal{G}(s_0)$. Thus we have Condition (3) for $\mathcal{G}$.

For Condition (4) for $\mathcal{G}$: Let s such that $\mathcal{G}(s)$. We now prove that $s \notin F$ by contradiction. Suppose $s \in F$. The definition of $\mathcal{G}$ implies $\exists a.s \in [\![a]\!] \wedge \mathcal{G}^\alpha(a)$. Condition (4) for $\mathcal{G}^\alpha$ implies $\exists a'.R^\alpha(a, a')$. However *Upward Termination* implies $a \in F^\alpha$, which implies $\neg \exists a' R^\alpha(a, a')$. Thus we have a contradiction. Thus we must have $s \notin F$. This gives Condition (4) for $\mathcal{G}$.

For Condition (5) for $\mathcal{G}$: Let s, s' such that $\mathcal{G}(s) \wedge R(s, s')$. The definition of $\mathcal{G}$ implies $\exists a.s \in [\![a]\!] \wedge \mathcal{G}^\alpha(a)$. Moreover, the *Simulation* condition gives $\exists a'.R^\alpha(a, a') \wedge s' \in [\![a']\!]$. Condition (5) for $\mathcal{G}^\alpha$ implies $\mathcal{G}^\alpha(a')$. The definition of $\mathcal{G}$ gives $\mathcal{G}(s')$ and thus we have Condition (5) for $\mathcal{G}$. □

Note that similar to what many abstractions do, a live abstraction can overapproximate the concrete initial states. For a live abstraction to be useful for proving nontermination using closed recurrence sets, we only need $a_0 \in S^\alpha$ and $s_0 \in S$ that satisfy the conditions of the soundness theorem.

Example. Recall Fig. 1(a) and its abstraction in Fig. 1(b). We can represent the abstraction as a transition system:

$$I^\alpha = \{a \mid a \vDash \mathsf{j} \geq 1 \wedge \mathsf{k} \geq 1\} \qquad F^\alpha = \{a \mid a \vDash \mathsf{i} < 0\}$$
$$S^\alpha = \mathbb{Z}^3 \qquad R^\alpha = \{(a, a') \mid (a, a') \vDash \mathsf{i} \geq 0 \wedge \mathsf{i}' \geq 1 \wedge \mathsf{j}' = \mathsf{j} + 1 \wedge \mathsf{k}' = \mathsf{k} + 1\}$$

The abstraction contains $\mathsf{i}' \geq 1$ in the transition relation of the loop instead of the nonlinear update $\mathsf{i}' = \mathsf{j} \times \mathsf{k}$. Here the abstraction has not changed the state space, the set of initial states and the set of final states, but it has weakened

the transition relation of the loop. Note that this abstraction fulfills all criteria for a live abstraction.

Example. Consider again the examples from Fig. 1(a) and (b). Here we have the closed recurrence set $\mathcal{G}^\alpha = \{s \mid s \vDash \mathsf{i} \geq 1\}$ for the loop in our abstraction in Fig. 1(b). This implies existence of a closed recurrence set $\mathcal{G}$ for the loop in the concrete program in Fig. 1(a) and hence its nontermination.

Example. To see why we need the *Upward Termination* condition for the abstraction, consider the following transition system (S, R, I, F) and its abstraction $(S^\alpha, R^\alpha, I^\alpha, F^\alpha)$:

$$S = \{s_0, s_1, s_2, s_3\} \quad I = \{s_0\} \quad F = \{s_1\}$$

$$R = \{(s_0, s_1), (s_2, s_3), (s_3, s_0)\}$$

$$S^\alpha = \{\{s_0\}, \{s_1, s_2\}, \{s_3\}\} \quad I^\alpha = \{\{s_0\}\} \quad F^\alpha = \emptyset$$

$$R^\alpha = \{(\{s_0\}, \{s_1, s_2\}), (\{s_1, s_2\}, \{s_3\}), (\{s_3\}, \{s_0\})\}$$

Here an abstract state is a subset of the set of all concrete states, where we have "merged" the states s_1 and s_2 to a single state. The abstraction satisfies the *Simulation* condition but not *Upward Termination* because s_1 is a final state in the concrete transition system, but the corresponding abstract state $\{s_1, s_2\}$ is not a final state in the abstract transition system. The abstraction has a closed recurrence set $\{\{s_0\}, \{s_1, s_2\}, \{s_3\}\}$, but the concrete transition system has no recurrence set.

4. Classes of Live Abstractions for Automation

As mentioned earlier, in our automation we focus on program fragments of a special shape: lassos.

Definition (Lasso) A *lasso* is a program fragment that contains a sequence of commands called a *stem* followed by a *simple loop* with guarded updates. The *guard* of a simple loop is a conjunction of atomic conditions. Formally a lasso L is a transition system $(S, R_{loop}, I_{loop}, F_{loop})$ where S is the set of states in the domain, R_{loop} is the transition relation of the loop, and I_{loop} is the set of initial states for the loop. I_{loop} represents the strongest postcondition after execution of the stem. F_{loop} is a set of final states for the loop such that for every final state there is no transition inside the loop.

Abstracting nonlinear commands. We describe the abstraction that our tool uses to abstract nonlinear commands present in the lassos. In our abstraction nonlinear assignment commands are abstracted, but loop guards are kept unchanged.

Towards the purpose of abstracting assignments we first compute a linear location invariant at the end of the loop (using Apron's [20] octagon abstract domain [26] in our implementation). We then replace the nonlinear update command with a nondeterministic choice and add an assume statement with the invariant at the end of the loop. Instead of octagons, here also dedicated disjunctive analyses for nonlinearity (*e.g.* the technique by Alonso *et al.* [1]) can be used to increase precision of the overapproximation. However, as our experiments show, here we can already get quite far using standard octagons.

Consider the nonlinear lasso in Fig. 1(a) and its linear abstraction in Fig. 3 that our tool computes. Here, $\mathsf{i} - 1 \geq 0 \wedge \mathsf{i} + \mathsf{j} - 3 \geq 0 \wedge \mathsf{i} - \mathsf{j} + 1 \geq 0 \wedge \mathsf{i} + \mathsf{k} - 3 \geq 0$ is the invariant computed at location ℓ of the original lasso from Fig. 1(a) by the Apron library using the octagon abstract domain.

```
assume(j ≥ 1 and k ≥ 1);
while i ≥ 0 do
    i := nondet();
    j := j + 1;
    k := k + 1;
    assume(i − 1 ≥ 0 and i + j − 3 ≥ 0 and
        i − j + 1 ≥ 0 and i + k − 3 ≥ 0); // location ℓ
done
```

Fig. 3. Linear overapproximation of the program in Fig. 1(a) computed by our tool using Apron [20]

```
assume(...);                    assume(...);
assume(...);                    assume(...);
while i × j ≥ 0 do              v := i × j;
    i := ...                    while v ≥ 0 do
    j := ...                        i := ...
    .....                           j := ...
done                                .....
                                    v := i × j;
                                done
        (a)                             (b)
```

Fig. 4. Lasso **(a)** with nonlinear guards and equivalent lasso **(b)** with auxiliary variable with linear guards

Mapping nonlinear assignments to nondeterministic assignments is clearly an overapproximation. This abstraction of assignments satisfies the *Simulation* condition of live abstraction because it adds extra abstract transitions only when a concrete transition (the assignment) is already possible. Since we do not alter loop guards, *Upward Termination* holds as well because all the final states of the original lasso are final states in the abstract lasso too. Clearly this abstraction satisfies the conditions of a live abstraction. Formally for a concrete lasso with a transition system $(S, R_{loop}, I_{loop}, F_{loop})$ our tool computes an abstract lasso with a transition system $(S^\alpha, R^\alpha_{loop}, I^\alpha_{loop}, F^\alpha_{loop})$ where $S^\alpha = S, R_{loop} \subseteq R^\alpha_{loop}, I^\alpha_{loop} = I_{loop}, F^\alpha_{loop} = F_{loop}$ and the concretion function is essentially the identity, *i.e.*, $\forall a \in S^\alpha.\ [\![a]\!] = \{a\}$.

Dealing with nonlinear guards. We use a simple trick to get rid of nonlinearity out of guards. Consider Fig. 4. We remove nonlinearity present in the guards by adding an auxiliary variable v. The rest of the analysis proceeds as before.

This approach yields nonlinear commands in the stem of our lassos. The stem commands enter our constraints only existentially (as we will see in Section 5). Thus constraint solvers can deal with such constraints efficiently.

Abstracting heap-based commands. Magill *et al.* [25] propose an overapproximating abstraction from programs operating on the heap to purely arithmetic programs. The abstraction is obtained by instrumenting a memory safety proof for the program. Since in general memory safety only holds under certain preconditions, the user can specify the shape of the

```
while p ≠ null do
    p := p→next;
done
```

(a)

```
while k ≥ 1 do
    assume(k > 1);
    l := nondet();
    assume(l ≥ 1 and k = l + 1);
    m := nondet();
    assume(m = l + 1);
    n := nondet();
    assume(n = l + 1);
    k := n;
done
```

(b)

Fig. 5. Heap-based program **(a)** with precondition that p points to a nonempty cyclic list and linear overapproximation **(b)** computed by THOR [25]

heap data structures by user-defined predicates in separation logic [28]. We can use Magill's tool THOR [25] to abstract heap-based C programs into linear arithmetic programs operating over the integers. This is exemplified in Fig. 5. In the arithmetic program the variable k tracks the length of the list segment from p to null, and the other variables are temporaries used in the update of k.

Magill's PhD thesis [24, Def. 29] describes the notion of stuttering simulation and proves (in his Thm. 18) that the abstraction satisfies the properties of stuttering simulation. In stuttering simulation for a transition in the concrete system, the corresponding transition in the abstract system may contain a sequence of steps and vice versa. An abstraction satisfying stuttering simulation obeys standard simulation condition and additionally for stuttering simulation to hold, the *Upward Termination* condition is needed. Thus Magill's abstraction satisfies the properties of a live abstraction and thus is safe for our approach of nontermination proving.

We could also abstract linked-list programs via the results connecting lists and counter automata [4]. These results are in fact stronger, a bisimulation rather than a simulation, for lists.

Combining over- and underapproximation. As previously mentioned, closed recurrence sets must in some cases be used in conjunction with underapproximation. Here we can use existing techniques for underapproximation in combination with our own. Note that closed recurrence sets form a complete method when combined with underapproximation, in the sense that every nonterminating program also has an underapproximation with a closed recurrence set.

Underapproximation. We call a transition system (S, R', I', F') an underapproximation of a transition system (S, R, I, F) *iff* $R' \subseteq R$, $I' \subseteq I$, $F \subseteq F'$.

Theorem (Open Recurrence Sets Always Contain Closed Recurrence Sets [8]) *There exists a recurrence set $\mathcal{G}$ for (S, R, I, F)* iff *there exist an underapproximation (S, R', I', F') of (S, R, I, F) and $\mathcal{G}' \subseteq \mathcal{G}$ such that $\mathcal{G}'$ is a closed recurrence set for (S, R', I', F').*

5. FINDING CLOSED RECURRENCE SETS

In the previous section we showed how it is possible to prove nontermination of a program by proving the existence of a closed recurrence set for an abstraction of the program. Here we address the problem of how to *find* a closed recurrence set for the abstracted program, *i.e.*, a program over linear integer arithmetic. We will search for a closed recurrence set $\mathcal{G}$ described by a conjunction of linear inequalities $Q\boldsymbol{x} \leq q$.

We adapt the Farkas-based approach used in TNT to find closed recurrence sets rather than recurrence sets. In our application the restriction to deterministic relations from TNT can be lifted. This is particularly important when working with abstractions of programs, which can introduce nondeterminism even when the concrete program is deterministic. It is also essential for treating the heap, because of the nondeterminism inherent in `malloc`.

In this section it will be convenient to phrase our discussion in terms of lassos expressed in linear arithmetic, as such lassos are convenient for automation. In the domain of linear arithmetic, a state s is just a vector $\boldsymbol{x}$ that represents the valuation of program variables. A lasso L in linear arithmetic can be expressed as a transition system $(S, R_{loop}(\boldsymbol{x}, \boldsymbol{x'}), I_{loop}(\boldsymbol{x}), F_{loop}(\boldsymbol{x}))$. In terms of programs, $I_{loop}(\boldsymbol{x})$ represents the strongest postcondition of a path leading to the loop body, with precondition 'true' from which the program starts, and $R_{loop}(\boldsymbol{x}, \boldsymbol{x'})$ is the transition relation corresponding to the composition of a sequence of (possibly nondeterministic) assignment statements in the loop body, guarded by a condition. $F_{loop}(\boldsymbol{x})$ represents the set of final states such that no loop transition can take place from any final state. As we are working in linear arithmetic, we can represent the transition relation of the loop by systems of inequalities

$$R_{loop}(\boldsymbol{x}, \boldsymbol{x'}) \triangleq G\boldsymbol{x} \leq g \wedge U\boldsymbol{x} + U'\boldsymbol{x'} \leq u$$

where $G\boldsymbol{x} \leq g$ describes the guards and $U\boldsymbol{x} + U'\boldsymbol{x'} \leq u$ the updates. Here G, U and U' are matrices, g and u are vectors. We make the following assumption:

$$\forall \boldsymbol{x} \exists \boldsymbol{x'}. G\boldsymbol{x} \leq g \rightarrow U\boldsymbol{x} + U'\boldsymbol{x'} \leq u. \tag{6}$$

The assumption says that whenever the guards of a lasso can be satisfied we are guaranteed to have a next state given by the updates. This holds in a lasso with a satisfiable transition system when every row in U' contains a non-zero coefficient, which corresponds to an update of the variables.

We are in search of a predicate $\mathcal{G}$ expressed as a system of inequalities using coefficients, *i.e.* $\mathcal{G} \equiv Q\boldsymbol{x} \leq q$, where Q is a matrix and q a vector of existentially quantified variables. The number of rows in Q and q then corresponds to the number of inequalities which we use.

We wish to employ a constraint solver (*e.g.* Z3 [21]) to find the coefficients Q and q. A difficulty in doing so is that these conditions contain mixtures of existential and universal quantifiers: Q and q are existentially quantified at the top-level, and both (4) and (5) use universals. Many constraint solvers struggle to solve problems such as these. The standard approach (*e.g.* in invariant generation [9], rank function synthesis [5] and recurrence set synthesis [18]) is to apply Farkas' lemma to convert the problem into a purely existential one that is easier for existing solvers.

In the remainder of this section we describe a Farkas-based reduction to automate the search for closed recurrence sets. To find a closed recurrence set for $(S, R_{loop}(\boldsymbol{x}, \boldsymbol{x'}), I_{loop}(\boldsymbol{x}), F_{loop}(\boldsymbol{x}))$ we must find Q and q such that the following conditions are satisfied (here we have substituted $Q\boldsymbol{x} \leq q$ for $\mathcal{G}$ in Conditions (3), (4), and (5)):

$$\exists \boldsymbol{x}.Q\boldsymbol{x} \leq q \wedge I_{loop}(\boldsymbol{x}) \tag{7}$$

$$\forall \boldsymbol{x} \exists \boldsymbol{x'}.Q\boldsymbol{x} \leq q \rightarrow R_{loop}(\boldsymbol{x}, \boldsymbol{x'}) \tag{8}$$

$$\forall \boldsymbol{x} \forall \boldsymbol{x'}.Q\boldsymbol{x} \leq q \wedge R_{loop}(\boldsymbol{x}, \boldsymbol{x'}) \rightarrow Q\boldsymbol{x'} \leq q \tag{9}$$

In order to apply Farkas' lemma we must eliminate the $\forall\exists$ alternation in Condition (8).[2] Assumption (6) lets us remove the existential quantifier in (8),[3] which now becomes:

$$\forall \boldsymbol{x}.Q\boldsymbol{x} \leq q \rightarrow G\boldsymbol{x} \leq g \tag{10}$$

Next, although it is not essential, because of (10) we can drop $G\boldsymbol{x} \leq g$ from $R_{loop}(\boldsymbol{x}, \boldsymbol{x'})$ in (9), thus giving us a simpler constraint to solve:

$$\forall \boldsymbol{x} \forall \boldsymbol{x'}.Q\boldsymbol{x} \leq q \wedge U\boldsymbol{x} + U'\boldsymbol{x'} \leq u \rightarrow Q\boldsymbol{x'} \leq q \tag{11}$$

Conditions (7), (10), and (11) are sufficient constraints for finding a closed recurrence set. Furthermore, (10) and (11) are now in a form which facilitates applications of Farkas' lemma to eliminate the universal quantifiers, and we obtain:

$$\exists \Lambda_1 \geq 0.\Lambda_1 Q = G \wedge \Lambda_1 q \leq g \tag{12}$$

and

$$\exists \Lambda_2 \geq 0.\Lambda_2 \begin{pmatrix} Q \\ U \end{pmatrix} = 0 \wedge \Lambda_2 \begin{pmatrix} 0 \\ U' \end{pmatrix} = Q \wedge \Lambda_2 \begin{pmatrix} q \\ u \end{pmatrix} \leq q \tag{13}$$

The constraints that we finally generate are (7), (12), and (13). These conditions are readily solved by off-the-shelf constraint solving tools. A satisfying assignment for these constraints gives us values of coefficients in Q and q, thus giving us the closed recurrence set.

Note that if the constraints are unsatisfiable, like Gupta *et al.* [18] we can use Q and q with increasingly many rows (and hence inequalities) in $Q\boldsymbol{x} \leq q$. In this way, we can increase the precision of our method further.

6. Implementation and Experiments

In order to assess the practicality of our approach we have developed a prototype implementation called ANANT. Given a program's CFG, ANANT exhaustively searches for candidate lassos.[4] For every lasso the tool applies our method, using Z3 as the constraint solver for the constraints from Section 5 together with abstractions for heap and nonlinear commands described in Section 3. If a lasso under consideration contains a loop variable with a nondeterministic update that also appears in the loop guard, before applying the abstraction the tool first applies an underapproximation strategy. To obtain the desired underapproximation the tool adds an **assume**-statement at the end of the loop body that enforces the loop guard (as done for variable m in Fig. 2(b)).

We make ANANT available for download along with its source code at the following URL:

http://www0.cs.ucl.ac.uk/staff/K.Nimkar/live-abstraction

We compared ANANT experimentally to several other tools. As a benchmark set (also available at the above URL), we have gathered 33 example programs containing nonlinear, nondeterministic and heap-based commands from various sources.

Since nontermination usually indicates a bug, some of our benchmarks implement functions computing factorial, logarithm, *etc.*, with typical programming mistakes that lead to nontermination. The set also includes the nonterminating examples from Berdine *et al.* [3], in particular the bug in a Windows device driver discussed in this paper. While Berdine *et al.* report that their analysis uncovers this bug by absence of a successful termination proof, we can now go a step further and actually *prove* nontermination of such heap programs.

We compared ANANT to the following tools:

- APROVE [15], using the Java bytecode frontend with the nontermination analysis by Brockschmidt *et al.* [7].
- JULIA [33], implementing a reduction to constraint logic programming described by Payet and Spoto [31].

Like Brockschmidt *et al.* [7], we were unable to obtain a working version of the tool INVEL [34]. Note that the other nontermination provers (*e.g.* TNT [18], T2 [8] and CPPINV [23]) are not applicable, as they do not support programs with nonlinear or heap-based commands.

Fig. 6 shows the results of our experiments with ANANT, APROVE, and JULIA. We ran ANANT and APROVE on an Intel i7-2640M CPU clocked at 2.8 GHz under Linux. For JULIA, an unknown cloud-based configuration was used. All tools were run with 600 s timeout. As Fig. 6 shows, ANANT succeeded on 29 of 33 benchmarks, whereas APROVE and JULIA succeeded on only 2 and 4 benchmarks, respectively. This difference is not surprising since overapproximation was thus far not applicable to *dis*proving termination for nonlinear and heap-based programs. In contrast, as our experiments show, we can now disprove termination in many such cases.

It is worth highlighting that *e.g.* on benchmark 9, ANANT took over 4 min to disprove termination, *vs.* JULIA's <7 s. This difference may partly be due to different machine configurations. However, note that a combined prover for termination and nontermination (like APROVE or JULIA) can discard parts of the program proved terminating and only analyze the rest for nontermination. This can lead to a more focused search for a nontermination proof than ANANT's approach of enumerating arbitrary lassos (whose termination might be easy to prove). Thus, ideally, our contributions for disproving termination should be combined with a termination prover.

[2] When Gupta *et al.* [18] search for recurrence sets, they also need to eliminate the $\forall\exists$ alternation in their constraints for automation. They do so by instantiating the existential variable explicitly with the value of the update. The price for this is that the update must be deterministic. We do not have this restriction.

[3] The statements (6) $\wedge$ (8) and (6) $\wedge$ (10) are equivalent.

[4] ANANT uses the same syntax for transition systems as the termination prover T2 [6]. For heap-based programs in C syntax, the lasso extraction is currently conducted manually.

	ANANT		APROVE		JULIA	
Benchmark	Res	Runtime	Res	Runtime	Res	Runtime
1	✓	0.50 s	×	*timeout*	×	7.01 s
2	✓	0.55 s	×	*timeout*	×	7.80 s
2a	✓	0.82 s	×	*timeout*	×	12.01 s
3	✓	0.56 s	×	*timeout*	×	7.74 s
4	✓	125.66 s	×	*timeout*	×	12.85 s
5	✓	0.45 s	×	18.59 s	×	7.24 s
6	✓	0.48 s	×	235.79 s	✓	7.70 s
7	✓	0.59 s	×	23.51 s	✓	11.83 s
8	✓	0.26 s	×	3.15 s	✓	5.08 s
9	✓	243.00 s	×	5.10 s	✓	6.72 s
10	✓	246.83 s	×	27.42 s	×	11.29 s
11	✓	0.63 s	×	*timeout*	×	8.69 s
12	×	2.35 s	×	*timeout*	×	10.67 s
13	×	1.40 s	×	108.61 s	×	8.54 s
14	✓	121.69 s	×	147.54 s	×	7.33 s
15	✓	131.80 s	×	*timeout*	×	8.45 s
16	✓	57.41 s	×	18.81 s	×	7.07 s
17	✓	0.54 s	×	24.18 s	×	7.06 s
18	×	0.66 s	×	28.03 s	×	6.92 s
19	✓	0.44 s	×	*timeout*	×	7.27 s
20	×	0.74 s	×	*timeout*	×	6.95 s
factorial	✓	0.38 s	×	*timeout*	×	7.57 s
log	✓	0.46 s	×	3.17 s	×	8.59 s
log_by_mul	✓	0.63 s	×	*timeout*	×	7.68 s
lasso_ex1	✓	0.45 s	×	*timeout*	×	7.03 s
lasso_ex2	✓	1.21 s	×	72.25 s	×	8.79 s
lasso_ex3	✓	0.48 s	×	*timeout*	×	7.28 s
nCr_combi	✓	0.70 s	×	10.45 s	×	17.26 s
power	✓	0.43 s	×	*timeout*	×	7.03 s
Create	✓	3.47 s	✓	1.75 s	×	4.94 s
Insert	✓	177.69 s	×	16.86 s	×	7.77 s
Traverse	✓	1.23 s	✓	2.12 s	×	50.28 s
WindowsBug	✓	21.69 s	×	14.46 s	×	50.92 s

Fig. 6. Results ("Res") and runtimes of ANANT, APROVE, and JULIA on 29 benchmarks with nonlinear arithmetic and 4 heap-based benchmarks from Berdine *et al.* [3]. Here ✓ denotes that the tool proved nontermination, × means that the tool returned without a definite answer, and *timeout* means that the run was terminated externally after 600 s.

7. CONCLUSION

Overapproximation is the workhorse of program analysis. Unfortunately, overapproximation can invalidate conventional techniques for disproving termination. In this paper we have introduced the notion of a *live abstraction* to show how overapproximation can *help*, not hinder nontermination proving. The idea is to prove the existence of a *closed recurrence set* rather than simply a recurrence set. This modification in strategy allows us to use off-the-shelf overapproximating abstractions, leading to a new set of methods for disproving termination of real programs.

Acknowledgments. We thank the anonymous reviewers for helpful suggestions and Fabian Emmes and Fausto Spoto for help with the experiments.

REFERENCES

[1] Diego Esteban Alonso, Puri Arenas, and Samir Genaim. Handling non-linear operations in the value analysis of COSTA. In *Proc. BYTECODE '11*.

[2] Mohamed Faouzi Atig, Ahmed Bouajjani, Michael Emmi, and Akash Lal. Detecting fair non-termination in multithreaded programs. In *Proc. CAV '12*.

[3] Josh Berdine, Byron Cook, Dino Distefano, and Peter O'Hearn. Automatic termination proofs for programs with shape-shifting heaps. In *Proc. CAV '06*.

[4] Ahmed Bouajjani, Marius Bozga, Peter Habermehl, Radu Iosif, Pierre Moro, and Thomas Vojnar. Programs with lists are counter automata. In *Proc. CAV '06*.

[5] Aaron R. Bradley, Zohar Manna, and Henny B. Sipma. Linear ranking with reachability. In *Proc. CAV '05*.

[6] Marc Brockschmidt, Byron Cook, and Carsten Fuhs. Better termination proving through cooperation. In *Proc. CAV '13*.

[7] Marc Brockschmidt, Thomas Ströder, Carsten Otto, and Jürgen Giesl. Automated detection of non-termination and `NullPointerExceptions` for Java Bytecode. In *Proc. FoVeOOS '11*.

[8] Hong-Yi Chen, Byron Cook, Carsten Fuhs, Kaustubh Nimkar, and Peter O'Hearn. Proving nontermination via safety. In *Proc. TACAS '14*.

[9] Michael A. Colón, Sriram Sankaranarayanan, and Henny B. Sipma. Linear invariant generation using non-linear constraint solving. In *Proc. CAV '03*.

[10] Byron Cook and Eric Koskinen. Reasoning about nondeterminism in programs. In *Proc. PLDI '13*.

[11] Byron Cook, Andreas Podelski, and Andrey Rybalchenko. Termination proofs for systems code. In *Proc. PLDI '06*.

[12] Nachum Dershowitz, Naomi Lindenstrauss, Yehoshua Sagiv, and Alexander Serebrenik. A general framework for automatic termination analysis of logic programs. *AAECC*, 12(1-2), 2001.

[13] Fabian Emmes, Tim Enger, and Jürgen Giesl. Proving non-looping nontermination automatically. In *Proc. IJCAR '12*.

[14] Samir Genaim, Michael Codish, John P. Gallagher, and Vitaly Lagoon. Combining norms to prove termination. In *Proc. VMCAI '02*.

[15] Jürgen Giesl, Marc Brockschmidt, Fabian Emmes, Florian Frohn, Carsten Fuhs, Carsten Otto, Martin Plücker, Peter Schneider-Kamp, Thomas Ströder, Stephanie Swiderski, and René Thiemann. Proving termination of programs automatically with AProVE. In *Proc. IJCAR '14*.

[16] Jürgen Giesl, René Thiemann, and Peter Schneider-Kamp. Proving and disproving termination of higher-order functions. In *Proc. FroCoS '05*.

[17] Sumit Gulwani, Saurabh Srivastava, and Ramarathnam Venkatesan. Program analysis as constraint solving. In *Proc. PLDI '08*.

[18] Ashutosh Gupta, Thomas A. Henzinger, Rupak Majumdar, Andrey Rybalchenko, and Ru-Gang Xu. Proving non-termination. In *Proc. POPL '08*.

[19] Arie Gurfinkel, Ou Wei, and Marsha Chechik. Yasm: A software model-checker for verification and refutation. In *Proc. CAV '06*.

[20] Bertrand Jeannet and Antoine Miné. Apron: A library of numerical abstract domains for static analysis. In *Proc. CAV '09*.

[21] Dejan Jovanovic and Leonardo de Moura. Solving non-linear arithmetic. In *Proc. IJCAR '12*.

[22] Daniel Kroening, Natasha Sharygina, Aliaksei Tsitovich, and Christoph M. Wintersteiger. Termination analysis with compositional transition invariants. In *Proc. CAV '10*.

[23] Daniel Larraz, Kaustubh Nimkar, Albert Oliveras, Enric Rodríguez-Carbonell, and Albert Rubio. Proving non-termination using Max-SMT. In *Proc. CAV '14*.

[24] Stephen Magill. *Instrumentation Analysis: An Automated Method for Producing Numeric Abstractions of Heap-Manipulating Programs*. PhD thesis, Carnegie Mellon University, Pittsburgh, PA, USA, 2010.

[25] Stephen Magill, Ming-Hsien Tsai, Peter Lee, and Yih-Kuen Tsay. Automatic numeric abstractions for heap-manipulating programs. In *Proc. POPL '10*.

[26] Antoine Miné. The octagon abstract domain. *Higher-Order and Symbolic Computation*, 19(1), 2006.

[27] Greg Nelson. A generalization of Dijkstra's calculus. *TOPLAS*, 11(4), 1989.

[28] Peter O'Hearn, John Reynolds, and Hongseok Yang. Local reasoning about programs that alter data structures. In *Proc. CSL '01*.

[29] Étienne Payet. Loop detection in term rewriting using the eliminating unfoldings. *Theor. Comput. Sci.*, 403(2-3), 2008.

[30] Étienne Payet and Frédéric Mesnard. A non-termination criterion for binary constraint logic programs. *TPLP*, 9(2), 2009.

[31] Étienne Payet and Fausto Spoto. Experiments with non-termination analysis for Java Bytecode. In *Proc. BYTECODE '09*.

[32] Fu Song and Tayssir Touili. Pushdown model checking for malware detection. In *Proc. TACAS '12*.

[33] Fausto Spoto, Fred Mesnard, and Étienne Payet. A termination analyzer for Java bytecode based on path-length. *TOPLAS*, 32(3), 2010.

[34] Helga Velroyen and Philipp Rümmer. Non-termination checking for imperative programs. In *Proc. TAP '08*.