A Forward Analysis for Recurrent Sets

Alexey Bakhirkin[1], Josh Berdine[2(✉)], and Nir Piterman[1]

[1] Department of Computer Science, University of Leicester, Leicester, UK
{ab643,nir.piterman}@le.ac.uk
[2] Microsoft Research, Cambridge, UK
jjb@microsoft.com

Abstract. Non-termination of structured imperative programs is primarily due to infinite loops. An important class of non-terminating loop behaviors can be characterized using the notion of recurrent sets. A recurrent set is a set of states from which execution of the loop cannot or might not escape. Existing analyses that infer recurrent sets to our knowledge rely on one of: the combination of forward and backward analyses, quantifier elimination, or SMT-solvers. We propose a purely forward abstract interpretation–based analysis that can be used together with a possibly complicated abstract domain where none of the above is readily available. The analysis searches for a recurrent set of every individual loop in a program by building a graph of abstract states and analyzing it in a novel way. The graph is searched for a witness of a recurrent set that takes the form of what we call a recurrent component which is somewhat similar to the notion of an end component in a Markov decision process.

1 Introduction

Termination is a fundamental property of software routines. The majority of code is required to terminate, e.g., dispatch routines of device drivers or other event-driven code, GPU programs – and the existence of non-terminating behaviors is a severe bug that might freeze a device, an entire system, or cause a multi-region cloud service disruption [1]. The problem of proving *termination* has seen much attention lately [15,16,27] but the techniques are sound and hence necessarily incomplete. That is, failure to prove termination does not imply the existence of non-terminating behaviors. Therefore, proving *non-termination* is an interesting complementary problem.

Several modern analyses [11,13,14] characterize non-terminating behaviors of programs or fragments of programs by a notion of *recurrent set*, i.e., a set of input states from which execution of the program or fragment cannot or might not escape (there are different flavors of recurrent sets). The analyses that can infer recurrent sets to our knowledge rely on one of: the combination of forward and backward analyses [13], quantifier elimination [11,14], or SMT-solvers [12]. We propose a purely forward abstract interpretation–based analysis that can be used with a potentially complicated abstract domain where none of the above is readily available. In our approach, we consider structured imperative programs

S. Blazy and T. Jensen (Eds.): SAS 2015, LNCS 9291, pp. 293–311, 2015.
DOI: 10.1007/978-3-662-48288-9_17

without recursion where loops are the only source of non-termination. Our analysis searches for what we call a *universal* recurrent set (that cannot be escaped) of every individual loop in a program by building and analyzing a graph of its abstract states. The main challenge of a forward approach is that while recurrent sets can be characterized by greatest fixed points of *backward transformers* (and this gives an intuition into the success of the approach [13] combining forward and backward analyses), we are not aware of a way to characterize them in terms of forward transformers. Instead, we produce a *condition* for a set of states to be recurrent and systematically explore the state space of a program searching for satisfying sets of abstract states. Our approach is similar to the one of Brockschmidt et al. [12], but the analysis of the state graph that we employ is novel. The graph is searched for a witness of a recurrent set that takes the form of what we call a *recurrent component* which is somewhat similar to the notion of an *end component* in a Markov decision process [8].

Note that finding a recurrent set is a *sub-problem* of proving non-termination. To prove non-termination, we would need to show that a recurrent set is reachable from the program entry. Also, some divergent behaviors do not fit the form discussed in this paper, and a non-terminating loop need not necessarily have a universal recurrent set.

2 Background

We define the analysis for a simple structured language without procedures. For a set of *atomic statements* A ranged over by a, statements C of the language are built as follows:

$C ::= a$	atomic statement
$\mid C_1 \,;\, C_2$	sequential composition: executes C_1 and then C_2
$\mid C_1 + C_2$	branch: non-deterministically branches to either C_1 or C_2
$\mid C^*$	loop: iterated sequential composition of ≥ 0 copies of C

We assume that A contains the passive statement *skip* and an assumption statement $[\theta]$ for each state formula θ, and that the language of state formulas is closed under negation. Informally, assumption statements work by filtering out the violating executions. Standard conditionals $\texttt{if}(\theta)\ C_1\ \texttt{else}\ C_2$ can be expressed by $([\theta]; C_1) + ([\neg\theta]; C_2)$. Similarly, loops $\texttt{while}(\theta)\ C$ can be expressed by $([\theta]; C)^*\ ; [\neg\theta]$.

2.1 Concrete Semantics

We use 1 and 0 to mean logical truth and falsity respectively. For a set S, we use Δ_S to mean the diagonal relation $\Delta_S = \{(s, s) \mid s \in S\}$. For a relation T, we use $T(s, s')$ to mean $(s, s') \in T$. We use $\circ$ for right composition of relations: $T_2 \circ T_1 = \{(s, s'') \mid \exists s'.\ (s, s') \in T_1 \wedge (s', s'') \in T_2\}$. For a function F, we use lfp F to mean its least fixed point. We use Kleene's 3-valued logic [21] to represent truth values of state formulas in abstract, and sets of concrete, states. It uses a

set of three values $\mathcal{K} = \{1, 0, {}^1/_2\}$ meaning *true*, *false*, and *maybe* respectively. $\mathcal{K}$ is arranged in partial *information order* $\sqsubseteq_{\mathcal{K}}$, s.t. 1 and 0 are incomparable, $1 \sqsubseteq_{\mathcal{K}} {}^1/_2$, and $0 \sqsubseteq_{\mathcal{K}} {}^1/_2$. For $k_1, k_2 \in \mathcal{K}$ the least upper bound $\sqcup_{\mathcal{K}}$ is defined s.t. $k_1 \sqcup_{\mathcal{K}} k_2 = k_1$ if $k_1 = k_2$, and ${}^1/_2$ otherwise.

Let $\boldsymbol{\mathcal{U}}$ be the set of all *memory* states. The concrete domain of the analysis is the powerset $\mathcal{P}(\boldsymbol{\mathcal{U}})$ with least element $\varnothing$, greatest element $\boldsymbol{\mathcal{U}}$, partial order $\subseteq$, and join $\cup$. This particular concrete domain is used for clarity of presentation, and another domain can be used if needed. A state formula θ denotes a set of states $[\![\theta]\!] \subseteq \boldsymbol{\mathcal{U}}$. We say that a state s *satisfies* θ if $s \in [\![\theta]\!]$. For a state formula θ and a set of states S, the *value* of θ over S is defined by: $\mathit{eval}(\theta, S) = 1$ if $S \subseteq [\![\theta]\!]$; $\mathit{eval}(\theta, S) = 0$ if $S \cap [\![\theta]\!] = \varnothing$; $\mathit{eval}(\theta, S) = {}^1/_2$ otherwise. That is, a formula evaluates to 1 in a *set* of states, if all states in the set satisfy the formula, to 0 if none satisfy the formula, and to ${}^1/_2$ if some of the states satisfy the formula and some do not.

The semantics of a statement C is a relation $[\![C]\!] \subseteq \boldsymbol{\mathcal{U}} \times \boldsymbol{\mathcal{U}}$. For a state s, $[\![C]\!](s, s')$ holds for every state s' that it is possible to reach by executing C from s. For an atomic statement a, we assume that $[\![a]\!]$ is pre-defined. Then $[\![C]\!]$ is defined as follows:

$$
\begin{aligned}
[\![\mathit{skip}]\!] &= \Delta_{\boldsymbol{\mathcal{U}}} & [\![C_1 \,;\, C_2]\!] &= [\![C_2]\!] \circ [\![C_1]\!] \\
[\![[\theta]]\!] &= \{(s, s) \mid s \in [\![\theta]\!]\} & [\![C_1 + C_2]\!] &= [\![C_1]\!] \cup [\![C_2]\!] \\
& & [\![C^*]\!] &= \mathrm{lfp}\, \lambda\, X.\, \Delta_{\boldsymbol{\mathcal{U}}} \cup (X \circ [\![C]\!])
\end{aligned}
$$

If for a state s, there exists no state s' s.t. $[\![C]\!](s, s')$, we say that the execution of C *diverges* from s. For "normal" programs, this definition agrees with the common one based on a small-step semantics: all traces starting from s are infinite, and there exists at least one. That is, if assumption statements appear only at the start of a branch or at the entry or exit of a loop (they cannot be used as normal atomic statements):

$$C ::= a \mid C_1 \,;\, C_2 \mid ([\varphi] \,;\, C_1) + ([\psi] \,;\, C_2) \mid ([\psi] \,;\, C)^* \,;\, [\varphi]$$

and branch and loop guard assumptions are exhaustive: $\varphi \vee \psi = 1$, then the only way for an execution to diverge is to get stuck in an infinite loop.

As standard, we define a *state transformer*, *post*, that for a statement C and a set of states S, gives the states a program might reach after executing C from a state in S: $\mathit{post}(C, S) = \{s' \mid \exists s \in S.\, [\![C]\!](s, s')\}$.

In what follows, we focus on the loop statement:

$$C_{\mathrm{loop}} = ([\psi_{\mathrm{ent}}] \,;\, C_{\mathrm{body}})^* \,;\, [\varphi_{\mathrm{exit}}] \tag{1}$$

where C_{body} is the *loop body*; if ψ_{ent} holds the execution may enter the loop body; if φ_{exit} holds the execution may exit the loop; and $\psi_{\mathrm{ent}} \vee \varphi_{\mathrm{exit}} = 1$. What is important for us is that this form of loop has a single point serving as both the entry and the exit. As currently formulated, our analysis relies on this property, although we anticipate that more complicated control flow graphs can be analyzed in a similar way.

For a loop as in (1), a *universal recurrent set* is a set $R_\forall$, s.t.,

$$R_\forall \subseteq [\![\neg\varphi_{\text{exit}}]\!] \qquad \forall s \in R_\forall.\, \big(\forall s' \in \boldsymbol{\mathcal{U}}.\, [\![C_{\text{body}}]\!](s, s') \Rightarrow s' \in R_\forall\big)$$

These are states that *must* cause non-termination, i.e., must cause the computation to stay inside the loop forever. Chen et al. [13] call a similar notion *closed recurrence set*. There is also a related notion of an *existential, or open, recurrent set*, i.e., a set of states that *may* cause non-termination, but it is not discussed here. Thus, in what follows, by just *recurrent set* we mean universal recurrent set.

Lemma 1. *For a loop as in* (1)*, the set* $R \subseteq \boldsymbol{\mathcal{U}}$ *is universally recurrent iff* $\mathit{eval}(\neg\varphi_{\text{exit}}, R) = 1$ *and* $\mathit{post}(C_{\text{body}}, R) \subseteq R$.

Proof. Follows from the definitions of *eval*, *post*, and universal recurrent set. □

2.2 Recurrent Sets in the Abstract

It is standard for forward program analyses to introduce an abstract domain $\mathcal{D}$ with least element $\bot_{\mathcal{D}}$, greatest element $\top_{\mathcal{D}}$, partial order $\sqsubseteq_{\mathcal{D}}$, and join $\sqcup_{\mathcal{D}}$. Every *element* of the abstract domain $d \in \mathcal{D}$ represents the set of concrete states $\gamma(d) \subseteq \boldsymbol{\mathcal{U}}$. Then, over-approximate versions of *post* and *eval*, are introduced, s.t. for a statement C, state formula θ and abstract element d,

$$\gamma(\mathit{post}^{\mathcal{D}}(C, d)) \supseteq \mathit{post}(C, \gamma(d)) \qquad \mathit{eval}^{\mathcal{D}}(\theta, d) \sqsupseteq_{\mathcal{K}} \mathit{eval}(\theta, \gamma(d))$$

We require that $\mathit{eval}^{\mathcal{D}}$ is homomorphic: for a formula θ and $d_1, d_2 \in \mathcal{D}$, $d_1 \sqsubseteq d_2 \Rightarrow \mathit{eval}^{\mathcal{D}}(\theta, d_1) \sqsubseteq_{\mathcal{K}} \mathit{eval}^{\mathcal{D}}(\theta, d_2)$. Normally, $\mathit{eval}^{\mathcal{D}}$ is given for atomic statements, and for arbitrary formulas it is defined by induction over the formula structure, using 3-valued logical operators, possibly over-approximate with respect to $\sqsubseteq_{\mathcal{K}}$.

Theorem 1. *For a loop as in* (1)*, an abstract domain* $\mathcal{D}$*, and an element* $d \in \mathcal{D}$*, if* $\mathit{eval}^{\mathcal{D}}(\neg\varphi_{\text{exit}}, d) = 1$ *and* $\mathit{post}^{\mathcal{D}}(C_{\text{body}}, d) \sqsubseteq_{\mathcal{D}} d$*, then* $\gamma(d)$ *is universally recurrent.*

For proofs, please, see the companion technical report [9].

Note that in Theorem 1, the post-condition is taken with respect to the loop body *without* the preceding assumption statement.

3 Finding a Universal Recurrent Set

We define our analysis for a finite powerset domain $\mathcal{P}(\mathcal{L})$, where the underlying set $\mathcal{L}$ of abstract elements is partially ordered by $\sqsubseteq_{\mathcal{L}}$ with least element $\bot_{\mathcal{L}}$. For example, in a numeric analysis, $\mathcal{L}$ may be the domain of intervals or polyhedra [17]. We call the elements of $\mathcal{L}$ *abstract states*. We assume that $\mathcal{P}(\mathcal{L})$ uses the Hoare order, and that concretization is defined as shown below. For $L, L_1, L_2 \subseteq \mathcal{L}$,

$$\gamma(L) = \bigcup \{\gamma(l) \mid l \in L\} \qquad L_1 \sqsubseteq_{\mathcal{P}(\mathcal{L})} L_2 \text{ iff } \forall l_1 \in L_1.\, \exists l_2 \in L_2.\, l_1 \sqsubseteq_{\mathcal{L}} l_2$$

We assume that evaluation function $eval^{\mathcal{P}(\mathcal{L})}$ and forward transformers $post^{\mathcal{P}(\mathcal{L})}$ for all statements (e.g. C_{body} in (1)) are given. We assume that $\bot_{\mathcal{L}}$ represents unreachability, and is transformed and evaluated precisely: $\gamma(\bot_{\mathcal{L}}) = \varnothing$, $post^{\mathcal{P}(\mathcal{L})}(C, \{\bot_{\mathcal{L}}\}) = \varnothing$, and $eval^{\mathcal{P}(\mathcal{L})}(\theta, \{\bot_{\mathcal{L}}\}) = 1$. Then, we define *pointwise* transformers $eval^\sharp$ and $post^\sharp$ as follows. For $L \subseteq \mathcal{L}$, statement C, and state formula θ,

$$post^\sharp(C, L) = \bigcup_{l \in L} post^{\mathcal{P}(\mathcal{L})}(C, \{l\}) \qquad eval^\sharp(\theta, L) = \bigsqcup_{l \in L}{}_{\mathcal{K}}\, eval^{\mathcal{P}(\mathcal{L})}(\theta, \{l\})$$

Note that $post^\sharp$ and $eval^\sharp$ are sound over-approximations of concrete *post* and *eval*. Also, if $post^{\mathcal{P}(\mathcal{L})}$ and $eval^{\mathcal{P}(\mathcal{L})}$ distribute over set union, then $post^\sharp = post^{\mathcal{P}(\mathcal{L})}$ and $eval^\sharp = eval^{\mathcal{P}(\mathcal{L})}$. For a single state $l \in \mathcal{L}$, we overload $post^\sharp(C, l)$ to mean $post^\sharp(C, \{l\})$ and $eval^\sharp(\theta, l)$ to mean $eval^\sharp(\theta, \{l\})$. We use $[\theta, l]^\sharp$ and $[\theta, L]^\sharp$ to mean $post^\sharp([\theta], l)$ and $post^\sharp([\theta], L)$ respectively.

We use a powerset domain for the following reason. Only a subset of the loop invariant belongs to a recurrent set, so there needs to be a mechanism in the abstract domain to partition the "interesting" and "not interesting" states. Therefore, we search for a recurrent set in the form of a *set of* abstract elements. We use Theorem 1 to show soundness: $\mathcal{P}(\mathcal{L})$ is $\mathcal{D}$ for its purposes; $post^\sharp$ and $eval^\sharp$ are $post^{\mathcal{D}}$ and $eval^{\mathcal{D}}$.

3.1 Idea of the Algorithm

```
1 | while  x ≥ 1 :
2 |   if  x = 60 :  x ← 50
3 |   x ← x + 1
4 |   if  x = 100 :  x ← 0
```

Fig. 1. Program for Example 1.

For a loop as in (1), if we find $X \subseteq \mathcal{L}$, s.t. $eval^\sharp(\neg\varphi_{\text{exit}}, X) = 1$ and $post^\sharp(C_{\text{body}}, X) \sqsubseteq X$, then $\gamma(X)$ is *definitely* a recurrent set. The idea is to explore the state space of the program with forward analysis until such an X is found. We proceed as follows. Separately for every loop, we build a graph where vertices are abstract elements, or *states*, from $\mathcal{L}$, all representing sets of concrete states at the loop head. We initialize the graph with some set of states $I \subseteq \mathcal{L}$ and then repeatedly apply the transformer for the whole loop body, $post^\sharp(C_{\text{body}}, \cdot)$, to the vertices and add the elements of the resulting set to the graph as successors. Our experiments suggest that in many cases a subset X of vertices satisfying the conditions of Theorem 1 will emerge as a result. To be able to efficiently find such a subset, we remember which elements are related w.r.t. abstract order $\sqsubseteq$, as a second kind of edges in the graph. Note that in case of nested loops, we analyze inner and outer loops separately; when analyzing the outer one, the effect of the inner needs to be summarized in an over-approximating way.

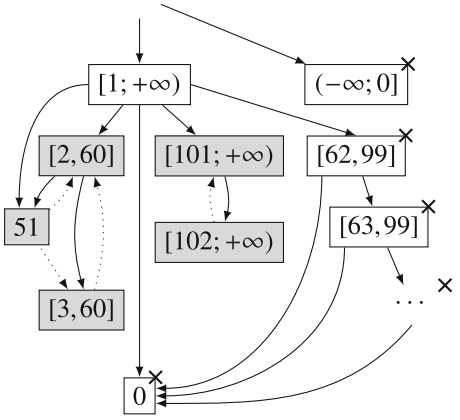

Fig. 2. Graph of the states of the program in Fig. 1.

We use a number of heuristics to help the analysis. First, we try to distinguish states that took different paths through the loop body. Currently, we take a simplistic approach: when possible, we prefer power-set domains where join is set union, s.t. states produced by different branches are not joined, and $post^\sharp(C_1 + C_2, l) = post^\sharp(C_1, l) \cup post^\sharp(C_2, l)$. If needed, a more involved trace partitioning [24] could be introduced instead. Second, with a similar intent, we compute the post-conditions with respect to a modified loop body $C_{\text{body}}{}' = C_{\text{body}}\ ; ([\psi_{\text{ent}}]+[\neg\psi_{\text{ent}}])$. This is sound since in the concrete case, for every set $S \subseteq \boldsymbol{\mathcal{U}}$, $post(C_{\text{body}}{}', S) = post(C_{\text{body}}, S)$. Also, for a set of initial states I, we initialize the graph with a set $I' = [\psi_{\text{ent}}, I]^\sharp \cup [\neg\psi_{\text{ent}}, I]^\sharp$. This is helpful when (as is often the case) there is a specific path through the loop body that infinite traces take. The heuristics introduce control-flow distinctions and enable states taking such path to be partitioned from others. But these heuristics may not be helpful when additional distinguishing power is needed for the data in states, e.g., when certain kinds of non-determinism are present, when non-termination depends on the properties of mathematical functions that the program implements, or when the abstract domain is not expressive enough to capture the states that take the interesting control paths.

Example 1. Consider the loop shown in pseudocode in Fig. 1. The loop does not terminate for some inputs, and the maximal recurrent set is $(1 \leq x \leq 60) \vee (x \geq 100)$. Let us informally demonstrate how the algorithm that we propose works, assuming that x ranges over integers and using intervals to represent its values. Since we do not know the initial value of x, we start with a graph consisting of a pair of states: $\{(-\infty; 0], [1; +\infty)\}$ – one represents the loop condition and another represents its complement. We then start adding new states to the graph by computing $post^\sharp$ as described above, s.t. paths through the loop body are represented in a post-condition of a state by different disjuncts. For example, let us see what happens to $[1; +\infty)$ when it enters the loop. In line 2, we consider three cases. If $x < 60$, then the conditional body in line 2 is skipped, x is incremented at line 3, the conditional body in line 4 is skipped, and the output state is $[2, 60]$. If $x = 60$, the conditional body in line 2 sets x to 50, at line 3 x is incremented, the conditional body in line 4 is skipped, and the output state is 51. If $x > 60$, the conditional body at line 2 is skipped and at line 3 x is incremented to $[62; +\infty)$. Then, if $x < 100$, the conditional body at line 4 is skipped, and the output state is $[62, 99]$. If $x = 100$, the conditional body at line 4 sets x to 0, and the output state is 0. If $x > 100$, the

conditional body at line 4 is skipped, and the output state is $[101; +\infty)$. Thus, $post^\sharp(C_{\text{body}}, [1; +\infty)) = \{[3, 60],\ 51,\ [62, 99],\ 0,\ [101; +\infty)\}$. We add these states to the graph and continue the exploration. Figure 2 shows a state graph that could be produced this way after a number of steps. In the graph, boxes represent states, and solid edges represent post-conditions. Note that in the graph, there exists a subset of states $X = \{[2, 60], [101; +\infty)\}$ has the desired property: $eval^\sharp(\neg\varphi_{\text{exit}}, X) = 1$ and $post^\sharp(C_{\text{body}}, X) \sqsubseteq X$, thus $\gamma(X)$ is a recurrent set. In what follows, we discuss how to efficiently find such subset of states if it exists. We revisit this example in Sect. 4.

For some domains (e.g., for shape analysis with 3-valued logic [26]), the analysis benefits from case splits that $post^\sharp$ naturally performs. For example, when a program traverses a potentially cyclic list, $post^\sharp$ would consider a definitely cyclic list as a separate case. If the abstraction is expressive enough, the cyclic list case will appear as a separate vertex and become part of a recurrent set.

Finally, the choice of the set of initial states I may matter. When the abstract domain is finite (and no widening is required) and the loop is not nested, we initialize the graph with the states that reach the loop via the rest of the program, i.e., produced by the standard forward analysis of the preceding part of the program. In this case, the analysis will explore all the states reachable at the head of the loop, and the success relies only on how refined the resulting graph is. When the abstract domain is infinite (e.g., for intervals or polyhedra) or for inner nested loops, we normally initialize the graph with a pre-fixpoint of $post^\sharp$. That is, we assume that initially, a standard forward analysis is run to produce a pre-fixpoint for every loop. Starting with a state below (w.r.t. $\sqsubseteq$) a pre-fixpoint makes it less likely that the analysis terminates, as our procedure does not include widening. Starting with a state above a pre-fixpoint is more likely to drive the search towards the states unreachable from the program entry. Note that it is *sound* to start with any set of states, and we sometimes use $\top$.

Our procedure is sound (by Theorems 1 and 2), but incomplete: if we do not find a recurrent set after a number of steps, we do not know the reason: whether the loop does not have a universal recurrent set; or the abstraction and $post^\sharp$ are not expressive enough; or we did not explore enough states. And for an infinite domain, the procedure might not terminate. So, we perform the exploration incrementally: we proceed breadth-first until some recurrent set is found. Then, we may decide to stop or to continue the search for a larger recurrent set.

3.2 Abstract State Graph

For a loop as in (1), an *abstract state graph* is a graph $G = \langle V, E_p, E_c \rangle$, s.t.,

- V is finite non-empty set of vertices which are abstract elements, or *states*: $V \subseteq \mathcal{L}$. All states belong to the loop entry location.
- There are two independent sets of edges: $E_c, E_p \subseteq V \times V$.
- E_p is a set of *post-edges*. For every state $l \in V$, *one* of the following holds:
  (i) there are no outgoing post-edges: $(\{l\} \times V) \cap E_p = \varnothing$; or

(ii) ψ_{ent} may hold in l, $eval^\sharp(\psi_{\text{ent}}, l) \neq 0$; post-condition of l with respect to the loop body is not empty, $post^\sharp(C_{\text{body}}, l) \neq \varnothing$; the whole post-condition is in the graph, $post^\sharp(C_{\text{body}}, l) \subseteq V$, and connected to l by post-edges, $(\{l\} \times V) \cap E_p = \{l\} \times post^\sharp(C_{\text{body}}, l)$; or

(iii) ψ_{ent} may hold in l, $eval^\sharp(\psi_{\text{ent}}, l) \neq 0$; post-condition of l is empty, $post^\sharp(C_{\text{body}}, l) = \varnothing$; l has $\bot_{\mathcal{L}}$ as the only post-successor, $\{l\} \times V \cap E_p = \{(l, \bot_{\mathcal{L}})\}$; and $\bot_{\mathcal{L}}$ has a post-self-loop $(\bot_{\mathcal{L}}, \bot_{\mathcal{L}}) \in E_p$.

– E_c is a set of *containment-edges*. For $l_1, l_2 \in V$, $(l_1, l_2) \in E_c \Leftrightarrow (l_1 \neq l_2 \wedge l_1 \sqsubseteq l_2)$.

This forbids self-loops. Due to properties of $\sqsubseteq$, G may not have containment cycles.

Note that this is similar to the notion of *termination graph* of [12]. For a loop as in (1), a state graph $G = \langle V, E_p, E_c \rangle$, a state $l \in V$, and a set of states $L \subseteq V$, let

$$post^G(l) = \{l' \in V \mid (l, l') \in E_p\} \qquad post^G(L) = \{l' \in V \mid \exists l \in L.\ (l, l') \in E_p\}$$

For a loop as in (1) and a graph $G = \langle V, E_p, E_c \rangle$, a *recurrent component* is a set of states $R \subseteq V$, s.t. for every state $l \in R$, l cannot exit the loop, $eval^\sharp(\neg\varphi_{\text{exit}}, l) = 1$, l has at least one outgoing edge, $\exists l' \in V.\ (l, l') \in E_p \cup E_c$, and *at least one* is true:

(i) l has a containment-edge into R, $\exists l' \in R.\ (l, l') \in E_c$; or

(ii) the outgoing post-edges of l lead exclusively into R, $post^G(l) \neq \varnothing \wedge post^G(l) \subseteq R$.

Lemma 2. *The union of two recurrent components is a recurrent component.*

Lemma 3. *In a state graph G, there exists a unique maximal (possibly, empty) recurrent component.*

Proof. Lemma 2 follows from the definition of recurrent component. Lemma 3 follows from Lemma 2 and finiteness of G. □

Theorem 2. *For a loop as in (1) and a state graph $G = \langle V, E_p, E_c \rangle$ we say $X \subseteq V$ is* fully closed *if $eval^\sharp(\neg\varphi_{\text{exit}}, X) = 1$, $\forall l \in X.\ post^G(l) \neq \varnothing$, and $post^\sharp(C_{\text{body}}, X) \sqsubseteq X$. Note that in this case, $\gamma(X)$ is a recurrent set. Then, for every state graph G:*

(i) For a recurrent component R, there exists a fully closed $X \subseteq R$ s.t. $\gamma(X) = \gamma(R)$.

(ii) For a fully closed X, there exists a recurrent component $R \supseteq X$, s.t. $\gamma(R) = \gamma(X)$.

3.3 The Algorithm

The algorithm, whose main body is shown in pseudocode in Fig. 3, is applied individually to every loop in a program. Initially, we call *FindFirst* giving it the set of elements $I \subseteq \mathcal{L}$ to start the search from (normally, a loop invariant). After

performing initialization, *FindFirst* calls *FindNext* once. *FindNext* contains a loop in which we build the state graph $G = \langle V, E_p, E_c \rangle$. In every iteration, proceeding in breadth-first order, we pick from the worklist F a state without post-edges and add its successors to the graph, together with relevant post- and containment-edges. This happens in lines 12–17 of Fig. 3; new states and post-edges are created by *MakeStates* shown in Fig. 4. We choose not to explore the successors of a state belonging to a recurrent component (line 13) even though when $post^\sharp$ is non-monotonic, they might lie outside the recurrent component. Similarly, we do not explore the successors of a must-exiting state, even if ψ_{ent} may hold in it. If adding new states and edges could create a larger recurrent component, we call *FindRecComp* to search for it (lines 20–21). If a new recurrent component is found, we return 1, and *Rec* contains those states of the component found so far that have no outgoing containment-edges (lines 22–27). If we wish to find a larger recurrent component, we can call *FindNext* again to resume the search. If the search terminates and no new recurrent component can be found, the procedure returns 0.

For every abstract state $l \in V$, we maintain the *status* as follows.

The state $l \in V$ *must exit*, $mustE(l) = 1$, if all executions starting in it exit the loop, i.e., if it is definitely the case that for every concrete state $s \in \gamma(l)$ the loop eventually terminates. We mark l as must-exiting if

(i) $eval^\sharp(\psi_{\text{ent}}, l) = 0$; or if
(ii) All post-successors of l are already must-exiting; or if
(iii) There exists a larger (w.r.t. $\sqsubseteq$) state that is already must-exiting.

The state $l \in V$ *may exit*, $mayE(l) = 1$, if we know that it cannot be part of a recurrent component. We mark l as may-exiting if

(i) it is must-exiting or if $eval^\sharp(\neg\varphi_{\text{exit}}, l) \neq 1$; or if
(ii) $post^\sharp$ is monotonic and l has a post-successor that is already may-exiting; or if
(iii) $post^\sharp$ is monotonic, and there exists a smaller (w.r.t. $\sqsubseteq$) already may-exiting state.

The state $l \in V$ is *recurrent*, $rec(l) = 1$, if it is a part of a recurrent component. If $post^\sharp$ is monotonic, we also mark as recurrent all successors of a recurrent state. Note that here, the term *recurrent* is overloaded. For a recurrent *state* $l \in V$, $\gamma(l)$ is in general not a recurrent *set* itself, but is included in some recurrent set.

Otherwise, the state $l \in V$ is *unknown*, $unk(l) = 1$, i.e., $unk(l) \Rightarrow (\neg mayE(l) \wedge \neg rec(l))$. This is the case if $eval^\sharp(\neg\varphi_{\text{exit}}, l) = 1$, and the state may potentially be a part of a recurrent component, but is not part of the recurrent component found so far.

Lemma 4. *May-exiting states cannot be part of a recurrent component.*

When searching for a recurrent component, it is only necessary to consider unknown and recurrent states, therefore every step of the algorithm only creates new containment-edges between unknown states or from an unknown to a recurrent state.

```
1  | while  G = ⟨V, E_p, E_c⟩, F, Rec
2  |
3  | proc FindFirst(I):
4  |   for l ∈ ℒ:
5  |     mayE(l) ← mustE(l) ← rec(l) ← 0;  unk(l) ← 1
6  |   MakeStates(I, nil)
7  |   F ← {l ∈ V | ¬mustE(l) ∧ l ≠ ⊥_ℒ}
8  |   FindNext()
9  |
10 | proc FindNext():
11 |   while F ≠ ∅:
12 |     l ← first(F);  F ← F \ {l}
13 |     if mustE(l) ∨ rec(l): continue
14 |     newPost ← MakeStates(post♯(C_body, l), l)
15 |     E_c^+ ←{(l', l'') ∈ V × V | unk(l') ∧ (unk(l'') ∨ rec(l'')) ∧ l' ⊑ l'' ∧
   |           (l'' ∈ newPost ∨ l' ∈ newPost)}
16 |     E_c ← E_c ∪ E_c^+
17 |     F ← F ∪ {l' ∈ newPost | ¬mustE(l') ∧ l' ≠ ⊥_ℒ}
18 |     PropagateStatus()
19 |     R ← ∅
20 |     if (newPost = ∅ ∧ (∀l' ∈ post^G(l). unk(l') ∨ rec(l'))) ∨ E_c^+ ≠ ∅:
21 |       R ← FindRecComp()
22 |     if R ≠ ∅:
23 |       for l ∈ R:  rec(l) ← 1;  unk(l) ← 0
24 |       PropagateStatus()
25 |       Rec' ← Rec
26 |       Rec ← {l' ∈ V | rec(l') ∧ ({(l', l'') | l'' ∈ V ∧ rec(l'')} ∩ E_c = ∅)}
27 |       if (Rec ≠ Rec'): return 1
28 |   return 0
```

Fig. 3. Main algorithm

Note that when new states or edges are added to the graph, or the status of an existing state changes, we make a call to *PropagateStatus*. For brevity, we do not show the pseudocode, and only informally describe its effect. *PropagateStatus* propagates the statuses through the edges of the graph according to the following rules. For a state l:

1. if $post^G(l) \neq \varnothing \wedge \forall l' \in post^G(l).\ mustE(l')$, then $mustE(l)$
2. if $mustE(l)$, then $\forall l'.\ (l', l) \in E_c \Rightarrow mustE(l')$
3. if $post^G(l) \neq \varnothing \wedge \forall l' \in post^G(l).\ rec(l')$, then $rec(l)$
4. if $rec(l)$, then $\forall l'.\ (l', l) \in E_c \Rightarrow rec(l')$

Additionally, if $post^\sharp$ is monotonic:

5. if $\exists l' \in post^G(l).\ mayE(l')$, then $mayE(l)$
6. if $mayE(l)$, then $\forall l'.\ (l, l') \in E_c \Rightarrow mayE(l')$
7. if $rec(l)$, then $\forall l' \in post^G(l).\ rec(l')$
8. if $mustE(l)$, then $\forall l' \in post^G(l).\ mustE(l')$

```
1  proc MakeStates(L, l_p):
2    N ← ∅
3    if L = ∅: L' ← {⊥_L}
4    else: L' ← [ψ_ent, L]^♯ ∪ [¬ψ_ent, L]^♯
5    for l ∈ L':
6      if l_p ≠ nil: E_p ← E_p ∪ (l_p, l)
7      if l ∉ V:
8        if eval^♯(ψ_ent, l) = 0:
9          unk(l) ← 0
10         mayE(l) ← mustE(l) ← 1
11       elif eval^♯(¬φ_exit, l) ≠ 1:
12         unk(l) ← 0
13         mayE(l) ← 1
14       V ← V ∪ l
15       N ← N ∪ l
16       if l = ⊥_L:
17         E_p ← E_p ∪ {(l, l)}
18   return N
```

Fig. 4. Adding new states. New states are unknown unless marked otherwise.

```
1  proc FindRecComp():
2    C ← {l ∈ V | unk(l)}
3    R ← {l ∈ V | rec(l)}
4    While 1:
5      C⁻ ← {l ∈ C |
           {(l, l') | l' ∈ C ∪ R} ∩ E_c = ∅ ∧
           (post^G(l) = ∅ ∨ post^G(l) ⊈ C ∪ R)}
6      if C⁻ = ∅: break
7      C ← C \ C⁻
8    return C
```

Fig. 5. Finding a recurrent component

Rules 1 and 2 are derived from the definition of must-exiting state. Rules 3 and 4 mark as recurrent those states that would be included in a recurrent component next time *FindRecComp* is called. Rules 5 and 6 are derived from the definition of may-exiting states. Rule 7 is for the case when for some l, first its post-condition is computed, and later, l is marked as recurrent by rule 4. If $post^\sharp$ is monotonic, the successors of l would eventually become part of a recurrent component. Similarly, rule 8 is for the case when for some l, first its post-condition is computed, and later, l is marked as must-exiting by rule 2. If $post^\sharp$ is monotonic, the successors of l would eventually be marked as must-exiting. This is not necessary for the correctness: every state that *PropagateStatus* marks as may- or must-exiting, cannot be part of a recurrent component, and every state that it marks as recurrent would eventually become a part of a recurrent component. But this allows to eliminate unknown states earlier, create fewer containment-edges, and search for recurrent component in a smaller portion of the graph.

Figure 4 shows the procedure *MakeStates* that adds new states to the graph. Given a set of abstract elements $L \subseteq \mathcal{L}$ and a predecessor state $l_p \in V$, it adds abstract states corresponding to L to the graph and creates post-edges from l_p to them. Every $l \in L$ is split into a pair of states with $[\cdot]^\sharp$, then is possibly marked as may- or must- exiting depending on the values of φ_{exit} and ψ_{ent} and added to the graph together with a post-edge from l_p. The procedure returns the set N of new states produced from L that were not present in the graph before.

Figure 5 shows the procedure *FindRecComp* that finds a recurrent component among the unknown states. It is called from *FindNext* when a new containment-edge is created or a state is discovered such that all its outgoing post-edges lead to existing unknown or recurrent states (i.e., when a larger recurrent component

could emerge). It starts the search with the whole set of unknowns as the candidate C and iteratively removes the states C^- that make the candidate violate the definition of recurrent component. Note that *FindRecComp* works incrementally: assuming that R is a set of states that are currently marked as recurrent (i.e., R is the recurrent component found so far), the procedure produces a set C, s.t. $C \cup R$ is a recurrent component. In general, C might not be a recurrent component by itself.

Theorem 3. *For an abstract state graph* $G = \langle V, E_p, E_c \rangle$ *and* some *recurrent component* $R \subseteq V$*, FindRecComp produces* $C \subseteq V$ *such that* $C \cup R$ *is the maximal recurrent component of* G*.*

4 Examples

In this section, we demonstrate how our analysis can be successfully applied to numeric and heap-manipulating programs. Examples 1 and 2 present **Numeric Programs**. Program variables range over integers, and we use intervals to represent their values.

Example 1 (Continued). Let us revisit Fig. 2. The figure displays a state graph of the program in Fig. 1 at a stage when the algorithm cannot find a larger recurrent component, and *FindNext* returns 0. The recurrent component is shown grayed, post-edges are solid, containment edges are dotted, and for clarity, containment-edges to and from may-exiting states are not displayed. The state $[1; +\infty)$ is may-exiting, and must-exiting states are marked with a cross. The resulting recurrent set is $\{[2, 60], [101; +\infty)\}$. Note that the states $x = 1$ and $x = 100$ are lost compared to the maximal recurrent set, and the discovered recurrent set is closed under application of the forward transformer, but not the backward transformer. This can be the case for some other tools based on forward semantics. For example, E-HSF [11] when presented with this example, may report the recurrent set to be $\{[4, 60], [100; +\infty)\}$. Also, note the set of must-exiting states (on the right in Fig. 2). While our algorithm often succeeds in proving that a recurrent set exists, it behaves badly when no recurrent set can be found. For example, in this case, it had to enumerate all states of the form $[62, 99], [63, 99], [64, 99]$, and so on. Finally, note that our procedure did terminate, although the domain is infinite and no measures were taken to guarantee termination.

Example 2. Figure 6 demonstrates a bug in the software of Zune players that on 31 Dec 2008 caused many devices to freeze [2]. The example is extracted from a procedure that was used to calculate the year based on the number of days passed since 1 Jan 1980. The loop repeatedly subtracts 365 or 366 from the number of days depending on whether the year is leap and increases the year by 1. Due to a logical error, if the year is leap and the number of days is 366, the variables are not updated, and the program goes into an infinite loop. We presented this program to our tool with the starting state being the loop

invariant: $year \geq 1980 \wedge days \geq 0$. Every call to *FindNext* extends the recurrent set with a single state: $year = 1980 \wedge days = 366$, $year = 1984 \wedge days = 366$, $year = 1988 \wedge days = 366$, and so on. The abstract domain was not strong enough to infer that every leap year causes non-termination. Also, because the analysis is forward-only, it did not explore the predecessors of those states: e.g., from the state $year = 1983 \wedge days = 731$, the loop also diverges, but this was not discovered by the tool. Still, we count this result as success: our tool does expose the bug even if it does not find all inputs for which the bug manifests.

```
1  | days ← a number ≥ 0
2  | year ← 1980
3  |  while  days > 365 :
4  |    if  leap(year) :
5  |       if  days > 366 :
6  |          days ← days − 366
7  |          year ← year + 1
8  |    else :
9  |       days ← days − 365
10 |       year ← year + 1
```

Fig. 6. A potentially non-terminating loop in Zune software (simplified).

Shape Analysis. Examples 3 and 4 present heap-manipulating programs. We use 3-valued logic [26] to represent heaps, and build the analysis on top of TVLA [3,23]. For more information on shape analysis with 3-valued logic, please refer to Sagiv et al. [26] and related papers [7,23,25]. In this framework, abstract heaps are represented by *3-valued structures*, i.e., models of 3-valued first-order logic with transitive closure. Every individual represents either a single heap cell or a set of heap cells that share some properties. Pointer variables are represented by unary predicates: the predicate is true for the cell where the variable points. Pointer fields are represented by binary predicates: the predicate is true for those pairs of cells where the corresponding field of one cell points to another. The analysis also maintains in the form of predicates additional information about the heap: whether the cells are reachable from each other, whether some condition is true of the cells, and so on. Three-valued structures can be displayed as *shape graphs*, and an example is shown in Fig. 7. The graph represents an acyclic singly-linked list with two or more elements and is read as follows. Left node represents a single cell which is the head of the list and is pointed to by pointer variables x and y. The text $c = {}^1/_2$ means that some condition c might or might not be true for the head – we do not know. The right node displayed with double border represents a finite non-empty set of cells that are the tail of the list. The dotted edge annotated by n between the head and the tail means that the pointer field n of the head points to some node of the tail, but not to all of them. The analysis is usually instructed that predicate n induces a function, but this is not reflected in the shape graph. The analysis also keeps track of reachability between cells with the predicate t_n. Solid t_n-edge between the head and the tail means that all cells of the tail are reachable from the head by traversing the n-pointers. Dotted n- and t_n-loops on the tail mean that there are pointers and reachability between some pairs of cells in the tail but not between all of them. Absence of n- and t_n edges from the tail to the head means that no cell in the tail points to or can reach head. In this case, the analysis is also instructed that there are no *shared* cells, i.e., every cell is pointed to by at most one cell. The above is sufficient for Fig. 7 to represent exactly the set of acyclic

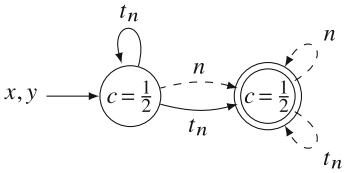

Fig. 7. Acyclic list with 2+ elements.

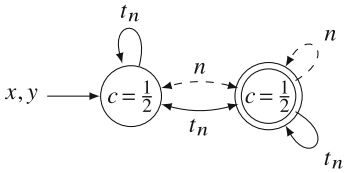

Fig. 8. Cyclic list with 2+ elements.

```
1 | y ← x
2 | while  y ≠ nil ∧ ¬c(y) :
3 |   y ← (y → n)
```

Fig. 9. Search in a list.

```
1 | y ← x
2 | while  y ≠ nil :
3 |   y ← new  struct
4 |   if  y ≠ nil :
5 |     (y → n) ← x
6 |     x ← y
```

Fig. 10. Prepending to a non-empty list.

singly-linked lists with two or more elements. Similarly, Fig. 8 represents a set of cyclic lists with two or more elements.

Example 3. One source of non-termination in heap-manipulating programs is incorrect traversal of cyclic data structures. The companion technical report [9] discusses such non-termination bug in a device driver that was found by a termination prover [10]. Figure 9 shows a procedure that searches a list pointed to by x for an element y s.t. the condition $c(y)$ holds. The search terminates when such y is found or when the end of the list is reached, and it does not handle cyclic lists correctly. In this and the next example, the initial statement: $y \leftarrow x$ – is disregarded by the analysis and only emphasizes that when the loop is reached for the first time, both x and y point to the head of the list. Due to canonical abstraction [26], the set of 3-valued structures that we can explore is finite, and there is no need to perform pre-analysis for the loop invariant. Thus, we analyze the loop starting with the set of states containing cyclic and acyclic lists with both x and y pointing to the head and with unknown value of c for all the cells: the structures shown in Figs. 7 and 8, plus structures to represent single-element lists and an empty list. This way, our tool reports as the recurrent set all the heaps that cause non-termination of the loop, i.e., the cyclic lists where the condition c is false for all the elements. One of such lists (with three or more elements, y pointing into the list) is shown in Fig. 11.

Example 4. Another interesting class of bugs in heap-manipulating programs is related to heap allocation. Sometimes, models of programs do not take into account that heap allocation can fail. For example, in a real program, an infinite loop performing allocation would usually lead to an out-of-memory error and may consume much time and system resources. But in a model of the program

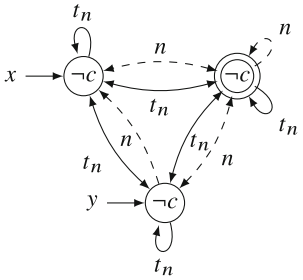

Fig. 11. Example of a cyclic list where c is false for all elements.

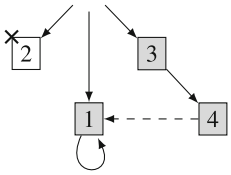

Fig. 12. State graph for the program in Fig. 10. State 1 is shown in Fig. 7. Grayed are recurrent states and must-exiting state is marked with a cross.

this may appear as potential non-termination. Figure 10 shows a program that repeatedly prepends a newly allocated element to a (non-empty) list. The loop is supposed to terminate if the allocation fails, but this is not possible in our TVLA model. The state space for the example is shown in Fig. 12. The initial states are: a list with two or more elements (state 1, as shown in Fig. 7), an empty heap (2), and a single-element list (3). The empty heap is must-exiting, and the states 1, 3, and 4 (list with exactly two elements) form the recurrent set. State 4 does not have an outgoing post-edge as the algorithm finishes before the post-condition of the state is computed. Note the post-loop on state 1. Because of canonical abstraction [26], the post-condition of a list with two or more elements is again a list with two or more elements, i.e., the analysis loses track of the length of the list.

5 Experiments

We implemented our technique in a prototype tool that supports numeric and 3-valued programs. The analysis of 3-valued programs is based on TVLA [23], and for numeric programs, we use interval domain with ad hoc support for *modulo* operation: we perform some artificial case splits when modulo operation is invoked. We applied our analysis to the test set [4] of Invel [28], and to the non-terminating programs from the Ultimate Büchi Automizer [20] test set [5]. *For detailed test results, please, see the companion technical report* [9]. Out of 52 non-terminating Invel programs, our tool was able to find recurrent sets in 39. For the remaining 13, the analysis either terminates without producing a result or diverges. We attribute 8 cases of failure to the lack of expressiveness in the abstract domain. In those programs, successful analysis would require relational reasoning, e.g., with polyhedra [17]. Another two cases of failure come from the limitations of our prototype tool that does not support nested loops (while the approach does). In one case, the program uses a `break` statement which

is not currently supported by our technique. Finally, two cases of failure seem problematic for our general approach. Those programs implement mathematical functions (least common multiplier and k-th Fibonacci number respectively) and their termination depends on the relation between the properties of those functions and program input, e.g., whether there exists such k that k-th Fibonacci number is equal to the argument of the function. As a result, we fail to isolate the path through the program that is taken by non-terminating traces. We may speculate that for a forward analysis to succeed, it needs to perform some artificial case splits, but we are not aware of a possible heuristic at this point. While specialized numeric tools (e.g., AProVE with SMT backend [12]) handle more of the Invel test programs [6], they do not subsume our tool. We believe, our approach can complement existing numeric tools in cases when the underlying linear solvers struggle.

Out of 18 Automizer programs that we considered, our tool handles 10 successfully. Among the remaining 8 programs, five use unsupported features (arrays, `break` statements, recursion), one would require additional case splits that our tool does not perform, and two have non-terminating behaviors, but do not have universal recurrent sets (non-termination relies on making a specific series of non-deterministic choices in the loop body). The latter points to a limitation of universal recurrent sets. Though a non-termination bug may cause the program to have one, it may be hard to build an abstraction that preserves it and does not introduce spurious terminating traces from every interesting state.

In some of the test programs, the main loop was preceded by a loop-free stem that performed initialization of the variables. We observed that in all cases (where our tool was able to find a recurrent set) this initial state had non-empty intersection with the recurrent set produced by the tool. For example, the program 'GCD' from the Invel test set, has two integer valuables: a an b – and the stem sets up the initial state $a \geq b$. The recurrent set that our tool finds is of the form $(a \geq 1 \wedge b \leq -1) \vee (a \leq -1 \wedge b \geq 1)$. The fact that this recurrent set has non-empty intersection with the initial state can be checked using the operations of the polyhedral abstract domain. This result is specific to the tests programs and the choice of abstract domain. In general, it might not be possible to check the recurrent sets for concrete reachability using standard forward analysis techniques.

6 Related Work

The approach [12] implemented in AProVE [18] is similar to ours in that it builds and analyzes an abstract state graph (*termination graph*, in their terms). However, they are interested in proving the existence of at least one non-terminating trace (which is dual to the notion of universal recurrent set) and they analyze the graph differently. They relate cycles in the graph to loops in the program and either try to prove that some loop does not modify the variables affecting termination, or employ SMT-based analysis (available when non-termination relies on integer arithmetic) to show that for some loop, at least one path through

it is always enabled. In contrast, we introduce a notion of recurrent component which witnesses a recurrent set and search the graph for those.

Cook et al. [14] analyze *linear* over-approximations of programs and use Farkas' lemma to find universal recurrent sets. Their soundness result is similar to ours and is more general: they state it for arbitrary transition systems and require a property of *upward termination* (for every concrete final state, the corresponding abstract state is also final) which for us implicitly holds. Note that linear abstractions have not yet demonstrated to be very effective for analyzing heap-manipulating programs.

The analysis of Chen et al. [13] combines a forward model checker and backwards analysis of single traces to modify the original program and turn it into a non-terminating one, by adding assumption statements. On the low level their approach is dual to ours, as they work with under-approximations of programs and try to prove the existence of at least one infinite trace.

The above analyses are predated by that of Gupta et al. [19] where existential recurrent sets are produced from lasso-shaped symbolic executions using Farkas' lemma.

Velroyen and Rümmer developed their analysis [28] independently of Gupta et al. [19]. They propose a template and a refinement scheme to infer invariants proving that terminating states of a program are unreachable.

Larraz et al. [22] use the notion of an edge-closed quasi-invariant (a set of states that, one reached, cannot be escaped) as a generalization of recurrent set. They encode the search for such set as a max-SMT problem.

We note that the above analyses focus on *proving non-termination*, while we consider a sub-problem of finding a recurrent set. To prove non-termination of a program we would need to show that a recurrent set is reachable from the program entry.

The analysis implemented in E-HSF [11] allows to specify the semantics of programs and express verified properties in the form of $\forall\exists$ quantified Horn clauses extended with well-foundedness conditions. In particular, the input language allows to query for the existence of universal and existential recurrent sets. The *implementation* is to our knowledge targeted at linear programs and relies on Farkas' lemma.

7 Conclusion and Future Work

We have described a forward technique for finding recurrent sets in imperative programs where loops of a specific form are the source of non-termination. The recurrent sets that we produce are genuine, but may not be reachable from the program entry. We applied our analysis to numeric and heap-manipulating programs and were successful if (i) we were able to capture the paths through the program that infinite traces take, and (ii) we were able to perform enough case splits to isolate the recurrent set into a separate set of abstract states. The latter point can benefit from heuristics in some cases.

Our analysis only admits structured programs without `goto` statements and a restricted form of loops: `while`-loops without statements that affect control flow (`break`, `continue`, etc.). One direction for future work is to enable the analysis of a larger class of loops: either by introducing relevant program transformations and studying their effect on the outcome of the analysis or by extending the technique to handle more complicated control flow graphs. Another direction is to solidify the analysis: eliminate the need for a separate forward pre-analysis by weaving it into the main algorithm, introduce a proper trace partitioning, etc.

Acknowledgements. We thank Mooly Sagiv and Roman Manevich for the source code of TVLA. A. Bakhirkin is supported by a Microsoft Research PhD Scholarship.

References

1. http://azure.microsoft.com/blog/2014/11/19/update-on-azure-storage-service-interruption. Accessed March 2015
2. http://www.zuneboards.com/forums/showthread.php?t=38143. Accessed March 2015
3. http://www.cs.tau.ac.il/~tvla/. Accessed March 2015
4. http://www.key-project.org/nonTermination/. Accessed March 2015
5. http://cl2-informatik.uibk.ac.at/mercurial.cgi/TPDB/file/d085ae59ef47/C/Ultimate. Accessed March 2015
6. http://aprove.informatik.rwth-aachen.de/eval/JBC-Nonterm/. Accessed June 2015
7. Arnold, G., Manevich, R., Sagiv, M., Shaham, R.: Combining shape analyses by intersecting abstractions. In: Emerson, E.A., Namjoshi, K.S. (eds.) VMCAI 2006. LNCS, vol. 3855, pp. 33–48. Springer, Heidelberg (2006)
8. Baier, C., Katoen, J.: Principles of Model Checking. MIT Press, Cambridge (2008)
9. Bakhirkin, A., Berdine, J., Piterman, N.: A forward analysis for recurrent sets. Technical report CS-15-001, University of Leicester (2015)
10. Berdine, J., Cook, B., Distefano, D., O'Hearn, P.W.: Automatic termination proofs for programs with shape-shifting heaps. In: Ball, T., Jones, R.B. (eds.) CAV 2006. LNCS, vol. 4144, pp. 386–400. Springer, Heidelberg (2006)
11. Beyene, T.A., Popeea, C., Rybalchenko, A.: Solving existentially quantified horn clauses. In: Sharygina, N., Veith, H. (eds.) CAV 2013. LNCS, vol. 8044, pp. 869–882. Springer, Heidelberg (2013)
12. Brockschmidt, M., Ströder, T., Otto, C., Giesl, J.: Automated detection of non-termination and NullPointerExceptions for Java Bytecode. In: Beckert, B., Damiani, F., Gurov, D. (eds.) FoVeOOS 2011. LNCS, vol. 7421, pp. 123–141. Springer, Heidelberg (2012)
13. Chen, H.-Y., Cook, B., Fuhs, C., Nimkar, K., O'Hearn, P.: Proving nontermination via safety. In: Ábrahám, E., Havelund, K. (eds.) TACAS 2014 (ETAPS). LNCS, vol. 8413, pp. 156–171. Springer, Heidelberg (2014)
14. Cook, B., Fuhs, C., Nimkar, K., O'Hearn, P.W.: Disproving termination with overapproximation. In: FMCAD, pp. 67–74. IEEE (2014)
15. Cook, B., Podelski, A., Rybalchenko, A.: Proving program termination. Commun. ACM **54**(5), 88–98 (2011)

16. Cook, B., See, A., Zuleger, F.: Ramsey vs. Lexicographic termination proving. In: Piterman, N., Smolka, S.A. (eds.) TACAS 2013 (ETAPS 2013). LNCS, vol. 7795, pp. 47–61. Springer, Heidelberg (2013)
17. Cousot, P., Halbwachs, N.: Automatic discovery of linear restraints among variables of a program. In: Aho, A.V., Zilles, S.N., Szymanski, T.G. (eds.) POPL, pp. 84–96. ACM Press (1978)
18. Giesl, J., Brockschmidt, M., Emmes, F., Frohn, F., Fuhs, C., Otto, C., Plücker, M., Schneider-Kamp, P., Ströder, T., Swiderski, S., Thiemann, R.: Proving termination of programs automatically with AProVE. In: Demri, S., Kapur, D., Weidenbach, C. (eds.) IJCAR 2014. LNCS, vol. 8562, pp. 184–191. Springer, Heidelberg (2014)
19. Gupta, A., Henzinger, T.A., Majumdar, R., Rybalchenko, A., Xu, R.G.: Proving non-termination. In: Necula, G.C., Wadler, P. (eds.) POPL, pp. 147–158. ACM (2008)
20. Heizmann, M., Hoenicke, J., Leike, J., Podelski, A.: Linear ranking for linear lasso programs. In: Van Hung, D., Ogawa, M. (eds.) ATVA 2013. LNCS, vol. 8172, pp. 365–380. Springer, Heidelberg (2013)
21. Kleene, S.: Introduction to Metamathematics, 2nd edn. Literary Licensing, LLC, Amsterdam (1987)
22. Larraz, D., Nimkar, K., Oliveras, A., Rodríguez-Carbonell, E., Rubio, A.: Proving non-termination using Max-SMT. In: Biere, A., Bloem, R. (eds.) CAV 2014. LNCS, vol. 8559, pp. 779–796. Springer, Heidelberg (2014)
23. Lev-Ami, T., Manevich, R., Sagiv, S.: TVLA: a system for generating abstract interpreters. In: Jacquart, R. (ed.) IFIP 2004. IFIP, vol. 156, pp. 367–375. Springer, Heidelberg (2004)
24. Mauborgne, L., Rival, X.: Trace partitioning in abstract interpretation based static analyzers. In: Sagiv, M. (ed.) ESOP 2005. LNCS, vol. 3444, pp. 5–20. Springer, Heidelberg (2005)
25. Reps, T., Sagiv, M., Loginov, A.: Finite differencing of logical formulas for static analysis. In: Degano, P. (ed.) ESOP 2003. LNCS, vol. 2618, pp. 380–398. Springer, Heidelberg (2003)
26. Sagiv, S., Reps, T.W., Wilhelm, R.: Parametric shape analysis via 3-valued logic. ACM Trans. Program. Lang. Syst. **24**(3), 217–298 (2002)
27. Urban, C., Miné, A.: A decision tree abstract domain for proving conditional termination. In: Müller-Olm, M., Seidl, H. (eds.) Static Analysis. LNCS, vol. 8723, pp. 302–318. Springer, Heidelberg (2014)
28. Velroyen, H., Rümmer, P.: Non-termination checking for imperative programs. In: Beckert, B., Hähnle, R. (eds.) TAP 2008. LNCS, vol. 4966, pp. 154–170. Springer, Heidelberg (2008)