Finding Recurrent Sets with Backward Analysis and Trace Partitioning

Alexey Bakhirkin   Nir Piterman

University of Leicester, Department of Computer Science

Why (Non-)Termination

A non-termination bug in the below code (simplified) made many Zune devices freeze on 31 Dec 2008.

```
days =// days since 1 Jan 1980
year = 1980;
while (days > 365) {
    if (leap(year))
        if (days > 366) {
            days = days − 366;
            year = year + 1;  }
    else {
        days = days − 365;
        year = year + 1;  }
}
```

The official response was, “Wait until battery dies”.

Why (Non-)Termination

- Many programs are supposed to terminate.
- Non-termination bugs have big impact, but are caused by simple errors.
- People are bad at finding (non-)termination bugs.
- We want automated analyses for:
  - validation (prove termination);
  - debugging (explain non-termination).
- Other analyses may rely on (non-)termination results.

Termination and Non-Termination

- A family of undecidable problems.
- Sound analyses are incomplete.

Find a set of states, such that from every state:

Every trace is finite (prove termination)	There exists an infinite trace (prove non-termination)
There exists a finite trace	Every trace is infinite

Recurrent Set

We search for a recurrent set which is a *sub-problem* of showing non-termination.

- Recurrent set is a set of states s.t. a program *may stay* in it forever (after it reaches the recurrent set).
- To prove non-termination, we need to show reachability of a recurrent set. *We do not do it*.

Abstract Interpretation

Search for a recurrent set fits into abstract interpretation.

Abstract Interpretation

Search for a recurrent set fits into abstract interpretation.

How to Use Abstract Interpretation

First, characterise the interesting property as a fixed point of some function.

- Example 1 (Invariant)

$$Inv = \mathsf{lfp}\, \lambda X.\, Init \cup \mathsf{post}(X)$$

Smallest set
that includes
initial states
and all its **own successors**

How to Use Abstract Interpretation

First, we characterize the interesting property as a fixed point.

- Example 1 (Invariant)

$$Inv = \mathrm{lfp}\, \lambda X.\, X \cup Init \cup \mathrm{post}(X)$$

- **Example 2** (set where a program may stay forever):

$$R_e = \mathrm{gfp}\, \lambda X.\, (\neg Final) \cap \mathrm{pre}(X)$$

$\uparrow$May lead into X

How to Use Abstract Interpretation

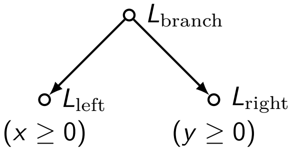
- **Second**, compute an approximation of the fixed point.
- The approximation will be in a certain form (called *abstract domain*, e.g., polyhedra, separation logic, etc).
- We find a stable limit of a chain:

$$
\begin{aligned}
& e_0 = \top \\
& e_1 = e_0 \sqcap (\neg \mathit{Final})^\flat \sqcap \mathrm{pre}^\flat(e_0) \\
& e_2 = e_1 \sqcap (\neg \mathit{Final})^\flat \sqcap \mathrm{pre}^\flat(e_1) \\
& \ldots \text{eventually} \\
& e_{k+1} = e_k
\end{aligned}
$$

- If the chain is infinite, use acceleration (*widening*).

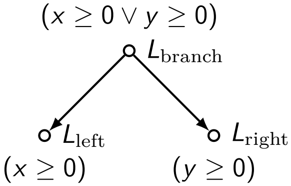
In Practice

- Recurrent set needs to be under-approximated.
- Under-approximation is difficult.
  - Transfer functions may have hidden disjunctions and recurrent sets may not be convex.



In Practice

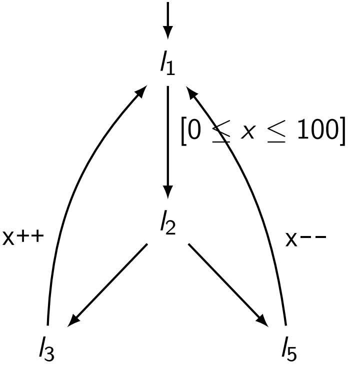
- Recurrent set needs to be under-approximated.
- Under-approximation is difficult.
  - Transfer functions may have hidden disjunctions and recurrent sets may not be convex.



In Practice

- Recurrent set needs to be under-approximated.
- Under-approximation is difficult.
  - Transfer functions may have hidden disjunctions and recurrent sets may not be convex.
- Have to come up with workarounds.
- Our workaround:
  - Allow some joins, guided by trace partitioning.
  - After computation, check for soundness.

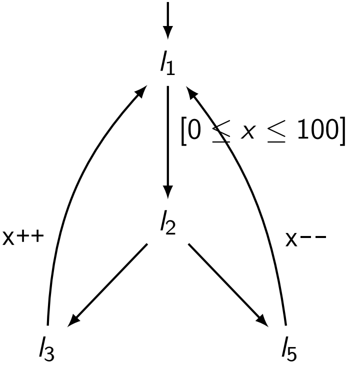
Recurrent Sets Via Compute-and-Check




```
while (0 ≤ x ≤ 100) {
  if (?)
    x++;
  else
    x--;
}
```

Recurrent Sets Via Compute-and-Check

First, approximate a fixed point.



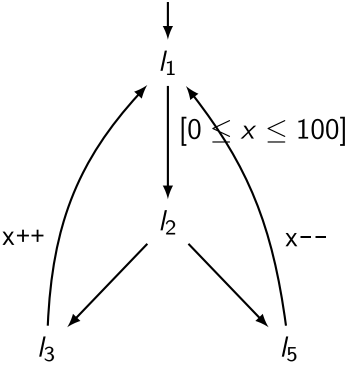
l_1 : $\top$

l_2 : $[0, 100]$

l_3 : $[0, 100]$

l_5 : $[0, 100]$

Recurrent Sets Via Compute-and-Check



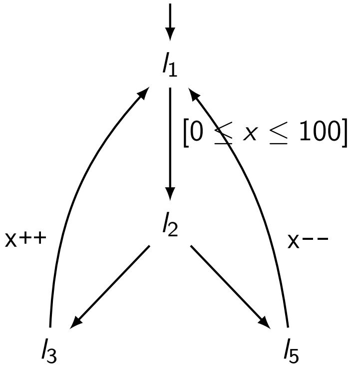
l_1 : $[0, 100]$

l_2 : $[0, 100]$

l_3 : $[0, 100]$

l_5 : $[0, 100]$

Recurrent Sets Via Compute-and-Check



l_1 : $[0, 100]$

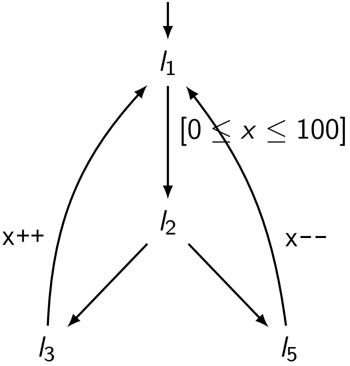
l_2 : $[0, 100]$

l_3 : $[0, 99]$

l_5 : $[1, 100]$

Recurrent Sets Via Compute-and-Check

Add path information to abstract states (*trace partitioning*).



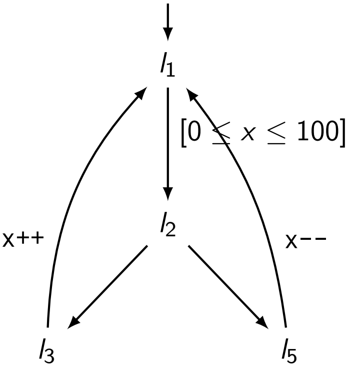
l_1 : $[0, 100]$

l_2 : $[0, 99]_{l_2 \to l_3}$, $[1, 100]_{l_2 \to l_5}$

l_3 : $[0, 99]$

l_5 : $[1, 100]$

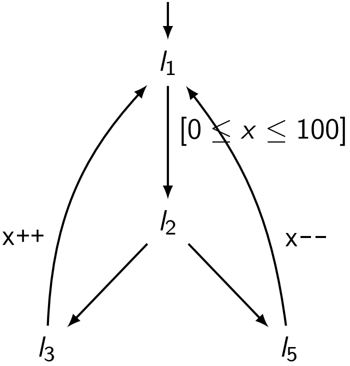
Recurrent Sets Via Compute-and-Check



l_1 :	$[0, 99]_{l_2 \to l_3}$,	$[1, 100]_{l_2 \to l_5}$
l_2 :	$[0, 99]_{l_2 \to l_3}$,	$[1, 100]_{l_2 \to l_5}$
l_3 :	$[0, 98]_{l_2 \to l_3}$,	$[0, 99]_{l_2 \to l_5}$
l_5 :	$[1, 100]_{l_2 \to l_3}$,	$[2, 100]_{l_2 \to l_5}$

Recurrent Sets Via Compute-and-Check

Allow some joins, guided by path domain.



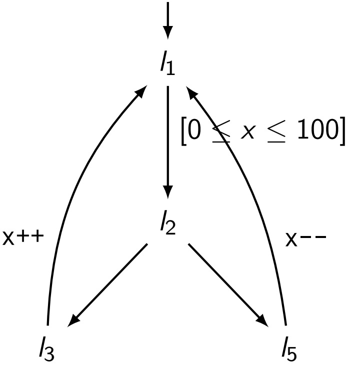
l_1 : $[0, 99]_{l_2 \to l_3}$, $[1, 100]_{l_2 \to l_5}$

l_2 : $[0, 99]_{l_2 \to l_3}$, $[1, 100]_{l_2 \to l_5}$

l_3 : $[0, 98]_{l_2 \to l_3}$, $[0, 99]_{l_2 \to l_5}$

l_5 : $[1, 100]_{l_2 \to l_3}$, $[2, 100]_{l_2 \to l_5}$

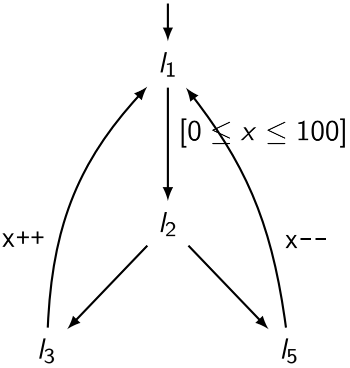
Recurrent Sets Via Compute-and-Check

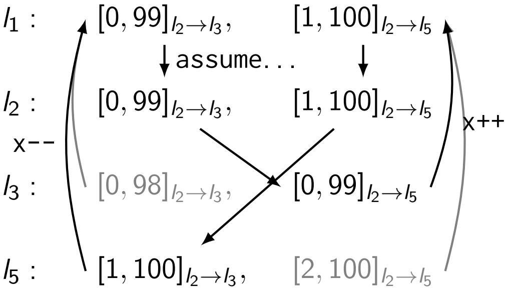


$$
\begin{array}{llll}
l_1 : & [0,99]_{l_2 \to l_3}, & [1,100]_{l_2 \to l_5} \\
l_2 : & [0,99]_{l_2 \to l_3}, & [1,100]_{l_2 \to l_5} \\
l_3 : & [0,98]_{l_2 \to l_3}, & [0,99]_{l_2 \to l_5} \\
l_5 : & [1,100]_{l_2 \to l_3}, & [2,100]_{l_2 \to l_5}
\end{array}
$$

Recurrent Sets Via Compute-and-Check

Then, ensure that it represents a recurrent set.





Recurrent Sets Via Compute-and-Check

- Implemented for individual loops of numeric programs.
- We believe, may be adapted for non-numeric programs.
- Precision depends whether path represenation can express the non-terminating paths.
- Compares well to other tools in benchmarks. We selected 44 non-terminating programs from SV-COMP'2015, and 3 other tools. All tools handle 30-40 programs well, with no tool subsuming the others.

We		Tool 1			Tool 2		Tool 3	
OK	X	OK	?	X	OK	X	OK	X
32	12	30(+6)	4	10(+6)	37(+11)	7(+6)	35(+7)	9(+4)

Recurrent Sets Via Compute-and-Check

- Implemented for individual loops of numeric programs.
- We believe, may be adapted for non-numeric programs.
- Precision depends whether path represenation can express the non-terminating paths.
- Compares well to other tools in benchmarks. We selected 44 non-terminating programs from SV-COMP'2015, and 3 other tools. All tools handle 30-40 programs well, with no tool subsuming the others.
- Many test programs have a single loop and a sequential stem $\rightarrow$ in principle, we can prove non-termination for them.

We		Tool 1			Tool 2		Tool 3	
OK	X	OK	?	X	OK	X	OK	X
32	12	30(+6)	4	10(+6)	37(+11)	7(+6)	35(+7)	9(+4)

Thanks

Related Work

- (Heizmann et al. 2015) Extract lasso-shaped subprograms (stem and branch-free loop) and analyze them separately.
- (Chen et al. 2014) Iteratively remove terminating behaviours from a program.
- (Beyene, Popeea, and Rybalchenko 2013) Encode problems as sets of quantified Horn clauses. Can express (non-)termination properties.
- (Brockschmidt et al. 2011) Implemented in AProVE, uses multiple techniques.
  - Build and analyze a graph of abstract states.
  - Produce an SMT problem.