Disproving Termination with Overapproximation

Byron Cook

Carsten Fuhs

Kaustubh Nimkar

Peter O'Hearn

University College London
Microsoft Research

FMCAD 2014, Lausanne, Switzerland, 24 October 2014

Proving Program Non-Termination

Given: program P
Goal: prove that P can have an **infinite run** for some input
$\rightarrow$ (usually) a bug

Note:
if termination proof attempt fails, this alone means nothing

- more sophisticated techniques might have proved termination ...
- ... or the program actually **is** non-terminating

$\Rightarrow$ **Need dedicated techniques to prove non-termination**

Proving Program Non-Termination

Given: program P
Goal: prove that P can have an **infinite run** for some input
$\rightarrow$ (usually) a bug

Note:
if termination proof attempt fails, this alone means nothing

- more sophisticated techniques might have proved termination ...
- ... or the program actually **is** non-terminating

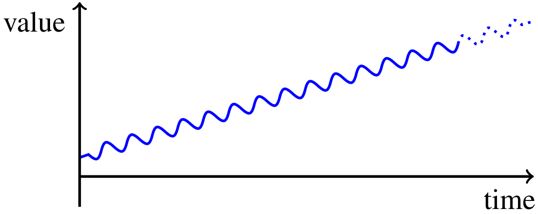
$\Rightarrow$ **Need dedicated techniques to prove non-termination**

This Talk in a Nutshell

Goal: show that for some input **there exists** an infinite run of program P

- compute (over-approximating) abstraction $\alpha(P)$ for P
- show that for some input **all** runs of $\alpha(P)$ are infinite

$\Rightarrow$ non-termination of P



concrete infinite run of P $=$ **some** abstract infinite run of $\alpha(P)$

Not all abstractions α are ok, but many are.

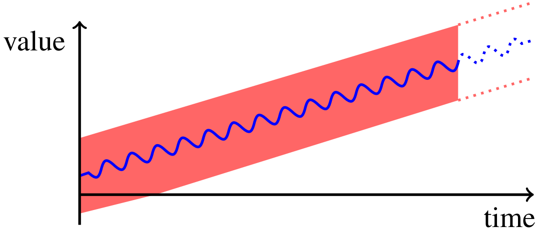
- new notion of **Live Abstractions** to prove non-termination
- e.g. for non-linear arithmetic, heap-based data structures, ...

This Talk in a Nutshell

Goal: show that for some input **there exists** an infinite run of program P

- compute (over-approximating) abstraction $\alpha(P)$ for P
- show that for some input **all** runs of $\alpha(P)$ are infinite

⇒ non-termination of P



concrete infinite run of P = **some** abstract infinite run of $\alpha(P)$

Not all abstractions α are ok, but many are.

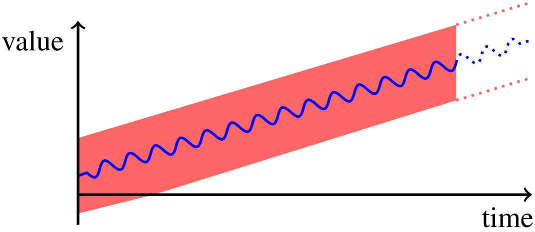
- new notion of **Live Abstractions** to prove non-termination
- e.g. for non-linear arithmetic, heap-based data structures, ...

This Talk in a Nutshell

Goal: show that for some input **there exists** an infinite run of program P

- compute (over-approximating) abstraction $\alpha(P)$ for P
- show that for some input **all** runs of $\alpha(P)$ are infinite

$\Rightarrow$ non-termination of P



concrete infinite run of P $=$ **some** abstract infinite run of $\alpha(P)$

Not all abstractions α are ok, but many are.

- new notion of **Live Abstractions** to prove non-termination
- e.g. for non-linear arithmetic, heap-based data structures, ...

Outline

Recurrence Set [Gupta *et al.*, POPL '08]

- set $\mathcal{G}$ of states: you can start in $\mathcal{G}$, and then you **can** stay in $\mathcal{G}$
- program P with transition relation R, initial states I
- $\mathcal{G}$ is recurrence set for P *iff*

($\mathcal{G}$ has an initial state) $\exists s.\, \mathcal{G}(s) \wedge I(s)$

(some transition **can** stay in $\mathcal{G}$) $\forall s\, \exists s'.\, \mathcal{G}(s) \rightarrow R(s, s') \wedge \mathcal{G}(s')$

Theorem (Gupta, Henzinger, Majumdar, Rybalchenko, Xu, *POPL '08*)

Program P non-terminating iff *P has a recurrence set $\mathcal{G}$.*

Automation

- by under-approximation to "lassos" and constraint solving
- restricted to **deterministic** programs on **linear integer arithmetic**

Recurrence Set [Gupta *et al.*, POPL '08]

- set $\mathcal{G}$ of states: you can start in $\mathcal{G}$, and then you **can** stay in $\mathcal{G}$
- program *P* with transition relation *R*, initial states *I*
- $\mathcal{G}$ is recurrence set for *P* *iff*

($\mathcal{G}$ has an initial state) $\exists s.\, \mathcal{G}(s) \wedge I(s)$

(some transition **can** stay in $\mathcal{G}$) $\forall s\, \exists s'.\, \mathcal{G}(s) \rightarrow R(s, s') \wedge \mathcal{G}(s')$

Theorem (Gupta, Henzinger, Majumdar, Rybalchenko, Xu, *POPL '08*)

Program P non-terminating iff *P has a recurrence set* $\mathcal{G}$.

Automation

- by under-approximation to "lassos" and constraint solving
- restricted to **deterministic** programs on **linear integer arithmetic**

Closed Recurrence Set [Chen *et al.*, TACAS '14]

- set $\mathcal{G}$ of states: you can start in $\mathcal{G}$, and then you **must** stay in $\mathcal{G}$
- program P with transition relation R, initial states I
- $\mathcal{G}$ is **closed** recurrence set for P *iff*

($\mathcal{G}$ has an initial state) $\exists s. \mathcal{G}(s) \wedge I(s)$

(all transitions **must** stay in $\mathcal{G}$) $\forall s \forall s'. \mathcal{G}(s) \wedge R(s, s') \rightarrow \mathcal{G}(s')$

(can make a transition from $\mathcal{G}$) $\forall s \exists s'. \mathcal{G}(s) \rightarrow R(s, s')$

- example

Closed Recurrence Set [Chen *et al.*, TACAS '14]

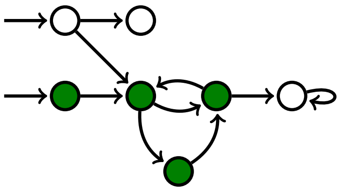
- set $\mathcal{G}$ of states: you can start in $\mathcal{G}$, and then you **must** stay in $\mathcal{G}$
- program P with transition relation R, initial states I
- $\mathcal{G}$ is **closed** recurrence set for P *iff*

($\mathcal{G}$ has an initial state) $\exists s.\, \mathcal{G}(s) \wedge I(s)$

(all transitions **must** stay in $\mathcal{G}$) $\forall s \forall s'.\, \mathcal{G}(s) \wedge R(s, s') \rightarrow \mathcal{G}(s')$

(can make a transition from $\mathcal{G}$) $\forall s \exists s'.\, \mathcal{G}(s) \rightarrow R(s, s')$

- example



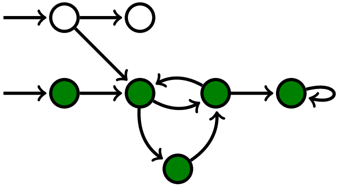
recurrence set $\mathcal{G}$

Closed Recurrence Set [Chen *et al.*, TACAS '14]

- set $\mathcal{G}$ of states: you can start in $\mathcal{G}$, and then you **must** stay in $\mathcal{G}$
- program P with transition relation R, initial states I
- $\mathcal{G}$ is **closed** recurrence set for P *iff*

($\mathcal{G}$ has an initial state)	$\exists s.\, \mathcal{G}(s) \wedge I(s)$
(all transitions **must** stay in $\mathcal{G}$)	$\forall s \forall s'.\, \mathcal{G}(s) \wedge R(s, s') \rightarrow \mathcal{G}(s')$
(can make a transition from $\mathcal{G}$)	$\forall s \exists s'.\, \mathcal{G}(s) \rightarrow R(s, s')$

- example



closed recurrence set $\mathcal{G}$

Beyond Linear Arithmetic

Programs can use more complex operations or data

- non-linear arithmetic

```
int x = z * z;
```

- dynamic data structures on the heap

```
list = list->next;
```

Standard solution: over-approximating abstractions
$\rightarrow$ fine for proving termination, but not for **non**-termination

Example (program and abstraction)

P :
```
while (x > 0) {
    x = x - z*z - 1;
}
```

$\Rightarrow$ terminating

$\alpha(P)$:
```
while (x > 0) {
    x = nondet();
}
```

$\Rightarrow$ becomes **non-terminating**

Abstraction $\alpha(P)$ non-terminating $\not\Rightarrow$ P non-terminating

Beyond Linear Arithmetic

Programs can use more complex operations or data

- non-linear arithmetic

```
int x = z * z;
```

- dynamic data structures on the heap

```
list = list->next;
```

Standard solution: over-approximating abstractions

$\to$ fine for proving termination, but not for **non**-termination

Example (program and abstraction)

P :

```
while (x > 0) {
    x = x - z*z - 1;
}
```

$\Rightarrow$ terminating

$\alpha(P)$:

```
while (x > 0) {
    x = nondet();
}
```

$\Rightarrow$ becomes **non-terminating**

Abstraction $\alpha(P)$ non-terminating $\not\Rightarrow$ P non-terminating

(Toy) Example for Non-Linear Arithmetic

program *P*

```
assume(j ≥ 1 ∧ k ≥ 1);
while (i ≥ 0) {
    i = j*k;
    j = j + 1;
    k = k + 1;
}
```

(Toy) Example for Non-Linear Arithmetic

program *P*

```
assume(j ≥ 1 ∧ k ≥ 1);
while (i ≥ 0) {
     i = j*k;
     j = j + 1;
     k = k + 1;
}
```

(initial states: $\mathbf{j} \geq 1 \wedge \mathbf{k} \geq 1$,
transition relation:

$$\mathbf{i} \geq 0 \wedge \mathbf{i}' = \mathbf{j} * \mathbf{k} \wedge$$
$$\mathbf{j}' = \mathbf{j} + 1 \wedge \mathbf{k}' = \mathbf{k} + 1)$$

(Toy) Example for Non-Linear Arithmetic

program *P*

```
assume(j ≥ 1 ∧ k ≥ 1);
while (i ≥ 0) {
     i = j*k;
     j = j + 1;
     k = k + 1;
}
```

(initial states: $\mathtt{j} \geq 1 \wedge \mathtt{k} \geq 1$,
transition relation:
$\mathtt{i} \geq 0 \wedge \mathtt{i'} = \mathtt{j} * \mathtt{k} \wedge$
$\mathtt{j'} = \mathtt{j} + 1 \wedge \mathtt{k'} = \mathtt{k} + 1$)

has (closed) recurrence set

$$\{(\mathtt{i} = 1, \mathtt{j} = 1, \mathtt{k} = 1),$$
$$(\mathtt{i} = 1, \mathtt{j} = 2, \mathtt{k} = 2),$$
$$(\mathtt{i} = 4, \mathtt{j} = 3, \mathtt{k} = 3),$$
$$(\mathtt{i} = 9, \mathtt{j} = 4, \mathtt{k} = 4), \quad \ldots\}$$

(Toy) Example for Non-Linear Arithmetic

program P

```
assume(j ≥ 1 ∧ k ≥ 1);
while (i ≥ 0) {
    i = j*k;
    j = j + 1;
    k = k + 1;
}
```

(initial states: $\mathtt{j} \geq 1 \wedge \mathtt{k} \geq 1$,
transition relation:
$\mathtt{i} \geq 0 \wedge \mathtt{i}' = \mathtt{j} * \mathtt{k} \wedge$
$\mathtt{j}' = \mathtt{j} + 1 \wedge \mathtt{k}' = \mathtt{k} + 1$)

has (closed) recurrence set
$\{(\mathtt{i} = 1, \mathtt{j} = 1, \mathtt{k} = 1),$
$(\mathtt{i} = 1, \mathtt{j} = 2, \mathtt{k} = 2),$
$(\mathtt{i} = 4, \mathtt{j} = 3, \mathtt{k} = 3),$
$(\mathtt{i} = 9, \mathtt{j} = 4, \mathtt{k} = 4), \quad \ldots\}$

abstract program $\alpha(P)$

```
assume(j ≥ 1 ∧ k ≥ 1);
while (i ≥ 0) {
    i = j*k;
    j = j + 1;
    k = k + 1;


}
```

(Toy) Example for Non-Linear Arithmetic

program *P*

```
assume(j ≥ 1 ∧ k ≥ 1);
while (i ≥ 0) {
     i = j*k;
     j = j + 1;
     k = k + 1;
}
```

(initial states: $\mathbf{j} \geq 1 \wedge \mathbf{k} \geq 1$,
transition relation:
$\mathbf{i} \geq 0 \wedge \mathbf{i}' = \mathbf{j} * \mathbf{k} \wedge$
$\mathbf{j}' = \mathbf{j} + 1 \wedge \mathbf{k}' = \mathbf{k} + 1$)

has (closed) recurrence set
$\{(\mathbf{i} = 1, \mathbf{j} = 1, \mathbf{k} = 1),$
$(\mathbf{i} = 1, \mathbf{j} = 2, \mathbf{k} = 2),$
$(\mathbf{i} = 4, \mathbf{j} = 3, \mathbf{k} = 3),$
$(\mathbf{i} = 9, \mathbf{j} = 4, \mathbf{k} = 4), \quad \ldots\}$

abstract program $\alpha(P)$

```
assume(j ≥ 1 ∧ k ≥ 1);
while (i ≥ 0) {
     i = j*k;
     j = j + 1;
     k = k + 1;
     assume(i ≥ 1);
     // linear invariant
}
```

(Toy) Example for Non-Linear Arithmetic

program *P*

```
assume(j ≥ 1 ∧ k ≥ 1);
while (i ≥ 0) {
    i = j*k;
    j = j + 1;
    k = k + 1;
}
```

(initial states: $\mathbf{j} \geq 1 \wedge \mathbf{k} \geq 1$,
transition relation:
$\mathbf{i} \geq 0 \wedge \mathbf{i}' = \mathbf{j} * \mathbf{k} \wedge$
$\mathbf{j}' = \mathbf{j} + 1 \wedge \mathbf{k}' = \mathbf{k} + 1$)

has (closed) recurrence set
$\{(\mathbf{i} = 1, \mathbf{j} = 1, \mathbf{k} = 1),$
$(\mathbf{i} = 1, \mathbf{j} = 2, \mathbf{k} = 2),$
$(\mathbf{i} = 4, \mathbf{j} = 3, \mathbf{k} = 3),$
$(\mathbf{i} = 9, \mathbf{j} = 4, \mathbf{k} = 4), \quad \ldots\}$

abstract program α(*P*)

```
assume(j ≥ 1 ∧ k ≥ 1);
while (i ≥ 0) {
    i = nondet();
    j = j + 1;
    k = k + 1;
    assume(i ≥ 1);
    // linear invariant
}
```

(Toy) Example for Non-Linear Arithmetic

program *P*

```
assume(j ≥ 1 ∧ k ≥ 1);
while (i ≥ 0) {
    i = j*k;
    j = j + 1;
    k = k + 1;
}
```

(initial states: $\mathtt{j} \geq 1 \land \mathtt{k} \geq 1$,
transition relation:

$$\mathtt{i} \geq 0 \land \mathtt{i}' = \mathtt{j} * \mathtt{k} \land$$
$$\mathtt{j}' = \mathtt{j} + 1 \land \mathtt{k}' = \mathtt{k} + 1)$$

has (closed) recurrence set

$$\{(\mathtt{i} = 1, \mathtt{j} = 1, \mathtt{k} = 1),$$
$$(\mathtt{i} = 1, \mathtt{j} = 2, \mathtt{k} = 2),$$
$$(\mathtt{i} = 4, \mathtt{j} = 3, \mathtt{k} = 3),$$
$$(\mathtt{i} = 9, \mathtt{j} = 4, \mathtt{k} = 4), \quad \ldots\}$$

abstract program $\alpha(P)$

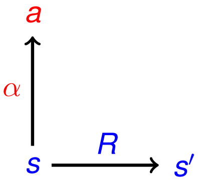
```
assume(j ≥ 1 ∧ k ≥ 1);
while (i ≥ 0) {
    i = nondet();
    j = j + 1;
    k = k + 1;
    assume(i ≥ 1);
    // linear invariant
}
```

has closed recurrence set

$$\{(\mathtt{i}, \mathtt{j}, \mathtt{k}) \mid \mathtt{i} \geq 1 \land \mathtt{j} \geq 1 \land \mathtt{k} \geq 1\}$$

Live Abstractions

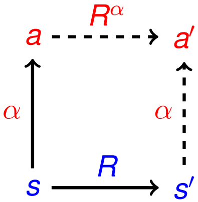
α is a **live abstraction** from $P = (R, I)$ to $\alpha(P) = (R^\alpha, I^\alpha)$ *iff*



(Simulation)

Live Abstractions

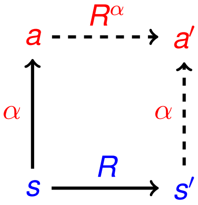
α is a **live abstraction** from $P = (R, I)$ to $\alpha(P) = (R^\alpha, I^\alpha)$ *iff*



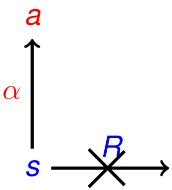
(Simulation)

Live Abstractions

α is a **live abstraction** from $P = (R, I)$ to $\alpha(P) = (R^\alpha, I^\alpha)$ *iff*



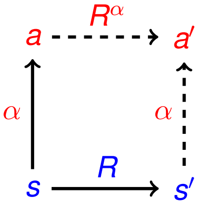
(Simulation)



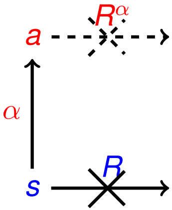
(Upward Termination)

Live Abstractions

α is a **live abstraction** from $P = (R, I)$ to $\alpha(P) = (R^\alpha, I^\alpha)$ *iff*



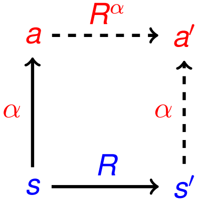
(Simulation)



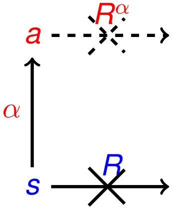
(Upward Termination)

Live Abstractions

α is a **live abstraction** from $P = (R, I)$ to $\alpha(P) = (R^\alpha, I^\alpha)$ *iff*



(Simulation)



(Upward Termination)

Theorem (Cook, Fuhs, Nimkar, O'Hearn, *FMCAD '14*)

Let α a live abstraction, let $\mathcal{G}^\alpha$ a closed recurrence set for $\alpha(P)$.
If there are a_0, s_0 with

$$a_0 \in I^\alpha \cap \mathcal{G}^\alpha$$
$$\alpha \uparrow$$
$$s_0 \in I$$

... then there is a closed recurrence set

$$\mathcal{G} = \{s \mid s \overset{\alpha}{\dashrightarrow} a \in \mathcal{G}^\alpha\} \quad \text{for } P$$

$\Rightarrow$ A closed recurrence set for $\alpha(P)$ also proves non-termination of P!

Live Abstractions in the Wild

Non-Linear Arithmetic

- find linear invariants (optional)
- then replace non-linear expressions in assignments by `nondet()`
- finally get linear arithmetic program

Heap-Based Programs

- programs with data structures on the heap: linked lists, trees, etc.
- abstraction to linear integer arithmetic program by THOR [Magill, Tsai, Lee, Tsay, *POPL '10*]
- THOR's abstraction is a live abstraction

. . .

Live Abstractions in the Wild

Non-Linear Arithmetic

- find linear invariants (optional)
- then replace non-linear expressions in assignments by `nondet()`
- finally get linear arithmetic program

Heap-Based Programs

- programs with data structures on the heap: linked lists, trees, etc.
- abstraction to linear integer arithmetic program by THOR [Magill, Tsai, Lee, Tsay, *POPL '10*]
- THOR's abstraction is a live abstraction

. . .

Automation and Implementation

Automation

- like [Gupta *et al.*, *POPL '08*] we only consider **lassos** → first **under-approximate** to lasso *L*, then abstract to $\alpha(L)$ in linear arithmetic
- use linear arithmetic **template** for closed recurrence set, find via Farkas' lemma + constraint solving (solution ⇒ values for template)
- can also deal with **`nondet()`**

Lasso-shaped programs

```
...
...
/* straight-line code */
while (... ∧ ...) {
    ...
    ...
    /* straight-line code */
}
```

Implementation in prototype tool ANANT

- extracts lasso from non-linear program
- uses APRON to find (octagon) invariants
- uses Z3 for constraint solving
- for heap-based C programs: abstraction by THOR

Automation and Implementation

Automation

- like [Gupta *et al.*, *POPL '08*]
  we only consider **lassos**
  → first **under-approximate** to
  lasso *L*, then abstract to $\alpha(L)$
  in linear arithmetic
- use linear arithmetic **template** for closed recurrence set, find via Farkas' lemma + constraint solving (solution ⇒ values for template)
- can also deal with `nondet()`

Lasso-shaped programs

```
...
...
/* straight-line code */
while (... ∧ ...) {
    ...
    ...
    /* straight-line code */
}
```

Implementation in prototype tool **ANANT**

- extracts lasso from non-linear program
- uses APRON to find (octagon) invariants
- uses Z3 for constraint solving
- for heap-based C programs: abstraction by THOR

Experiments

- collected benchmark set of 29 non-linear and 4 heap-based programs (literature, typical programming mistakes, ...)
- many tools only work on linear integer arithmetic programs
- experimented with ANANT, APROVE, JULIA
- timeout 600 s

Number of non-termination proofs found:

	Non-linear	Heap
ANANT	**25**	**4**
APROVE	0	2
JULIA	4	0

$\Rightarrow$ live abstractions open up more complex program domains for non-termination proving

Future Work

- lasso extraction in ANANT stand-alone
  $\rightarrow$ should be much more efficient in combination with a termination prover
- lift automation beyond lassos
- identify further classes of live abstractions

Conclusion

- new notion of live abstractions to disprove termination using **over**-approximation + **closed** recurrence sets
- allows to prove non-termination on complex data domains
  $\rightarrow$ non-linear arithmetic, heap, ...
- implementation in prototype tool ANANT
- tool and benchmark set available at

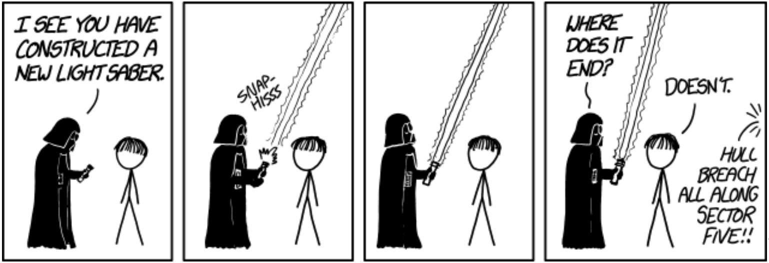
`http://www0.cs.ucl.ac.uk/staff/K.Nimkar/live-abstraction`

... **is *your* abstraction a live abstraction?**

Bonus Slide: Safety has the Same Issue, Right?

Analysis of safety (unreachability of "bad" states):
Check with **symbolic execution** if an abstract counterexample is legit

But: Counterexamples to termination are **infinite** . . .



[1]

. . . so their symbolic execution **does not terminate**

---

[1] http://xkcd.com/1433/