Monotone Program Analysis

Shaowei Zhu
With Zachary Kincaid and Nicolas Koh

Princeton University

IFIP 2.3, May 2024

Outline

Introduction

Background and Preliminaries

A Framework for Compositional and Monotone Termination Analysis

A Weak Theory of Nonlinear Arithmetics with Applications

Experimental Evaluation

Takeaways

Ensuring correctness of programs

Incorrect software has cost time, money, and human lives.[1]

- Non-terminating device driver code leads to non-responsive systems
- Software errors in the Ariane 5 rocket cost about $370M
- Therac-25 software errors caused deaths of cancer patients
- ...

.

---

[1] `https://www.cs.tau.ac.il/~nachumd/horror.html`

Why can't we solve the problem once and for all?

Undecidability of termination

There does not exist an algorithm that solves the halting problem for every program and input.

Why can't we solve the problem once and for all?

Undecidability of termination

There does not exist an algorithm that solves the halting problem for every program and input.

Rice's theorem

All non-trivial semantic properties of programs are undecidable.

Why can't we solve the problem once and for all?

Undecidability of termination

There does not exist an algorithm that solves the halting problem for every program and input.

Rice's theorem

All non-trivial semantic properties of programs are undecidable.

Thus

- Automated verification usually involves trade-offs between soundness, precision, resource consumption, etc.;
- Studying programs and loops with restricted forms is justified;
- Use of heuristics that only work *sometimes* can be justified.

A troubled user of our termination provers

	CPAChecker	UAutomizer	2LS

A troubled user of our termination provers

```
int n = 4096;
for (int i = 0; i < n; i++)
  for (int j = 0; j < n; j++)
    ; // skip
```

CPAChecker	UAutomizer	2LS
✗	✓	✓

A troubled user of our termination provers

```c
int n = 4096;
for (int i = 0; i < n; i++)
  for (int j = 0; j < n; j++)
    ; // skip
```

CPAChecker	UAutomizer	2LS
✗	✓	✓

```c
for (int i = 0; i < 4096; i++)
  for (int j = 0; j < 4096; j++)
    ; // skip
```

CPAChecker	UAutomizer	2LS
✗	✗	✓

A troubled user of our termination provers

	CPAChecker	UAutomizer	2LS

`int n = 4096;`
`for (int i = 0; i < n; i++)`
`  for (int j = 0; j < n; j++)`
`    ; // skip`
	✗	✓	✓

`for (int i = 0; i < 4096; i++)`
`  for (int j = 0; j < 4096; j++)`
`    ; // skip`
	✗	✗	✓

`for (int i = 0; i < 4096; i++)`
`  for (int j = 0; j < 4096; j++)`
`    i = i;`
	✗	✗	✗

A troubled user of our termination provers

```c
int n = 4096;
for (int i = 0; i < n; i++)
  for (int j = 0; j < n; j++)
    ; // skip
```

CPAChecker	UAutomizer	2LS
✗	✓	✓

```c
for (int i = 0; i < 4096; i++)
  for (int j = 0; j < 4096; j++)
    ; // skip
```

✗	✗	✓

```c
for (int i = 0; i < 4096; i++)
  for (int j = 0; j < 4096; j++)
    i = i;
```

✗	✗	✗

The unpredictability problem

Changes in the source program may have unpredictable effects on the analysis results.

The undesirable and the desirable

Program verification is currently unusable, because current tools require too much expertise from the user. Stated differently, tools can understand our programs, but we cannot understand our tools.
– Rustan and Michał

The undesirable and the desirable

Program verification is currently unusable, because current tools require too much expertise from the user. Stated differently, tools can understand our programs, but we cannot understand our tools.
– Rustan and Michał

I want to understand why Coverity findings disappear and re-appear...
– A user of static analysis

The undesirable and the desirable

Program verification is currently unusable, because current tools require too much expertise from the user. Stated differently, tools can understand our programs, but we cannot understand our tools.
– Rustan and Michał

I want to understand why Coverity findings disappear and reappear...
– A user of static analysis

We want to make verification part of the continuous integration pipeline...
– Peter

A troubled (expert) user

---

[1] `https://github.com/dafny-lang/dafny/issues/1426`

A troubled (expert) user

- Azadeh: Urgent help needed for teaching in class, Dafny verifying forever.

---

[1] `https://github.com/dafny-lang/dafny/issues/1426`

A troubled (expert) user

- Azadeh: Urgent help needed for teaching in class, Dafny verifying forever.

```
lemma SquareRoot2NotRational(p: nat, q: nat)
requires p > 0 && q > 0
ensures p * p !=  2 * q * q
{
if (p * p) ==  2 * (q * q) {
    calc == {
        (2 * q - p) * (2 * q - p);
        4 * q * q + p * p - 4 * p * q;
        {assert 2 * q * q == p * p;}
        2 * q * q + 2 * p * p - 4 * p * q;
        2 * (p - q) * (p - q);
    }
    SquareRoot2NotRational(2 * q - p, p - q);
}
}
```

---

[1] `https://github.com/dafny-lang/dafny/issues/1426`

A troubled (expert) user

- Azadeh: Urgent help needed for teaching in class, Dafny verifying forever.
- Rustan: I'm guessing the nonlinear arithmetic is causing the extreme slow-down ...

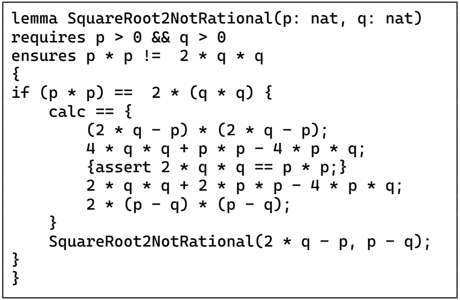
```
lemma SquareRoot2NotRational(p: nat, q: nat)
requires p > 0 && q > 0
ensures p * p !=  2 * q * q
{
if (p * p) ==  2 * (q * q) {
    calc == {
        (2 * q - p) * (2 * q - p);
        4 * q * q + p * p - 4 * p * q;
        {assert 2 * q * q == p * p;}
        2 * q * q + 2 * p * p - 4 * p * q;
        2 * (p - q) * (p - q);
    }
    SquareRoot2NotRational(2 * q - p, p - q);
}
}
```

---

[1] `https://github.com/dafny-lang/dafny/issues/1426`

A troubled (expert) user

- Azadeh: Urgent help needed for teaching in class, Dafny verifying forever.
- Rustan: I'm guessing the nonlinear arithmetic is causing the extreme slow-down ...
- Azadeh: But I used the same example in class last year.



```
lemma SquareRoot2NotRational(p: nat, q: nat)
requires p > 0 && q > 0
ensures p * p !=  2 * q * q
{
if (p * p) ==  2 * (q * q) {
    calc == {
        (2 * q - p) * (2 * q - p);
        4 * q * q + p * p - 4 * p * q;
        {assert 2 * q * q == p * p;}
        2 * q * q + 2 * p * p - 4 * p * q;
        2 * (p - q) * (p - q);
    }
    SquareRoot2NotRational(2 * q - p, p - q);
}
}
```

---

[1] `https://github.com/dafny-lang/dafny/issues/1426`

A troubled (expert) user

This story from the Github issue[1] illustrates

1. Users want behaviors of tools to be somehow predictable.
2. Nonlinear verification conditions can cause unpredictable behavior.

- Azadeh: Urgent help needed for teaching in class, Dafny verifying forever.
- Rustan: I'm guessing the nonlinear arithmetic is causing the extreme slow-down ...
- Azadeh: But I used the same example in class last year.

```
lemma SquareRoot2NotRational(p: nat, q: nat)
requires p > 0 && q > 0
ensures p * p !=  2 * q * q
{
if (p * p) ==  2 * (q * q) {
    calc == {
        (2 * q - p) * (2 * q - p);
        4 * q * q + p * p - 4 * p * q;
        {assert 2 * q * q == p * p;}
        2 * q * q + 2 * p * p - 4 * p * q;
        2 * (p - q) * (p - q);
    }
    SquareRoot2NotRational(2 * q - p, p - q);
}
}
```

---

[1] `https://github.com/dafny-lang/dafny/issues/1426`

Predictability via monotonicity

Current state-of-the-art tools exhibit unpredictable behavior.

Predictability via monotonicity

Current state-of-the-art tools exhibit unpredictable behavior.

We consider one property that makes the behavior of tools more predictable:

Predictability via monotonicity

Current state-of-the-art tools exhibit unpredictable behavior.

We consider one property that makes the behavior of tools more predictable:

Monotonicity

A monotone analysis improves or maintains the same result when given a problem that is “not more difficult” than the original.

Proposition: tool users will appreciate such properties!

Can you make it more concrete?

Example 1: reducing reachable states

- Before: a tool can prove a program P terminates unconditionally.
- After: can the tool prove that P terminates under precondition $x \geq 0$?

Can you make it more concrete?

Example 1: reducing reachable states

- Before: a tool can prove a program P terminates unconditionally.
- After: can the tool prove that P terminates under precondition $x \geq 0$?

Example 2: adding loop invariants

- Before: a tool can prove an assertion C in a program P.
- After: a user annotates P with an additional loop invariant I. Can the tool still prove the assertion?

Sources of non-monotonicity

1. Widening and narrowing in abstract interpretation.
2. Heuristics in model checking used by abstraction/refinement methods, e.g., selecting predicates from counterexamples.
3. Techniques based on syntactic features of code rather than semantics.
4. Undecidability of nonlinear integer arithmetic (NIA).
5. ...

My Research

I have worked on predictable analyses. In particular, I present:

- A framework for compositional and monotone termination analysis.
- A weak theory of nonlinear arithmetic that enables monotone nonlinear invariant generation and synthesis of polynomial ranking functions.

Outline

Introduction

Background and Preliminaries

A Framework for Compositional and Monotone Termination Analysis

A Weak Theory of Nonlinear Arithmetics with Applications

Experimental Evaluation

Takeaways

Program states and transitions

Fix finite sets of pre-state program variables X and corresponding post-state variables X'.

- A state formula $S(X)$ is a formula with free variables drawn from X.

Program states and transitions

Fix finite sets of pre-state program variables X and corresponding post-state variables X'.

- A state formula $S(X)$ is a formula with free variables drawn from X.
- A transition formula $F(X, X')$ is formula with free variables drawn from $X \cup X'$.

Program states and transitions

Fix finite sets of pre-state program variables X and corresponding post-state variables X'.

- A state formula $S(X)$ is a formula with free variables drawn from X.
- A transition formula $F(X, X')$ is formula with free variables drawn from $X \cup X'$.

Program states and transitions

Fix finite sets of pre-state program variables X and corresponding post-state variables X'.

- A state formula $S(X)$ is a formula with free variables drawn from X.
- A transition formula $F(X, X')$ is formula with free variables drawn from $X \cup X'$.
- Transition formulas characterize semantics of programs
    - statement `i := i + 1`
    - formula $F \triangleq i' = i + 1 \land j' = j$
    - transition system $(\mathbb{Q}^2, \xrightarrow{F})$ with a transition relation on states $\{i \mapsto x, j \mapsto y\} \rightarrow_F \{i \mapsto x + 1, j \mapsto y\}$.

Regular and ω-regular expressions

Let Σ be an alphabet. Define syntax for labels, regular expressions (Kleene), and ω-regular expressions (Büchi) as

$$\begin{aligned} a &\in \Sigma \\ e \in \mathsf{RegExp}(\Sigma) &::= a \mid 0 \mid 1 \mid e_1 + e_2 \mid e_1\, e_2 \mid e^* \\ f \in \omega\text{-}\mathsf{RegExp}(\Sigma) &::= e^\omega \mid ef \mid f_1 + f_2 \end{aligned}$$

Regular and ω-regular expressions

Let Σ be an alphabet. Define syntax for labels, regular expressions (Kleene), and ω-regular expressions (Büchi) as

$$
\begin{aligned}
a &\in \Sigma \\
e \in \mathsf{RegExp}(\Sigma) &::= a \mid 0 \mid 1 \mid e_1 + e_2 \mid e_1\,e_2 \mid e^* \\
f \in \omega\text{-}\mathsf{RegExp}(\Sigma) &::= e^\omega \mid ef \mid f_1 + f_2
\end{aligned}
$$

- A regular expression recognizes a set of finite strings.
- An ω-regular expression recognizes a set of *infinite* strings.
  - A^ω recognize all words obtained by concatenating words from A infinitely many times, e.g., $(b^*\,a)^\omega$ recognizes all words with infinitely many a's.

Algebraic program analysis (1)

Algebraic program analysis [Tar81b, FK15] is a form of abstract interpretation in that it computes summaries that over-approximate loop dynamics.

- Traditional (iterative) abstract interpretation:
    - Start with a system of recursive equations that encode some abstract semantics of the program
    - Interpret the operations in the equations in an abstract domain
    - Solve the equations over the abstract domain by computing a fixpoint

Algebraic program analysis (1)

Algebraic program analysis [Tar81b, FK15] is a form of abstract interpretation in that it computes summaries that over-approximate loop dynamics.

- Traditional (iterative) abstract interpretation:
  - Start with a system of recursive equations that encode some abstract semantics of the program
  - Interpret the operations in the equations in an abstract domain
  - Solve the equations over the abstract domain by computing a fixpoint
- Algebraic program analysis:
  - Compute a system of recursive relations that encode some abstract semantics of the program
  - Compute a closed-form solution, which represents (approximates) the transitive closure
  - Interpret the closed-form solutions in some algebraic structure

Algebraic program analysis (1)

Algebraic program analysis [Tar81b, FK15] is a form of abstract interpretation in that it computes summaries that over-approximate loop dynamics.

- Traditional (iterative) abstract interpretation:
  - Start with a system of recursive equations that encode some abstract semantics of the program
  - Interpret the operations in the equations in an abstract domain
  - Solve the equations over the abstract domain by computing a fixpoint
- Algebraic program analysis:
  - Compute a system of recursive relations that encode some abstract semantics of the program
  - Compute a closed-form solution, which represents (approximates) the transitive closure
  - Interpret the closed-form solutions in some algebraic structure

Convenient to build monotone analyses: APA avoids fixpoint computations in abstract domains thus avoids widening or narrowing.

Algebraic program analysis (2)

General recipe:

Algebraic program analysis (2)

General recipe:

1. Compute a regular expression that represent all finite program paths starting from the program entry

Algebraic program analysis (2)

General recipe:

1. Compute a regular expression that represent all finite program paths starting from the program entry
2. Analyze program paths by interpreting the regular expression in some algebra

Algebraic program analysis (2)

General recipe:

1. Compute a regular expression that represent all finite program paths starting from the program entry
2. Analyze program paths by interpreting the regular expression in some algebra

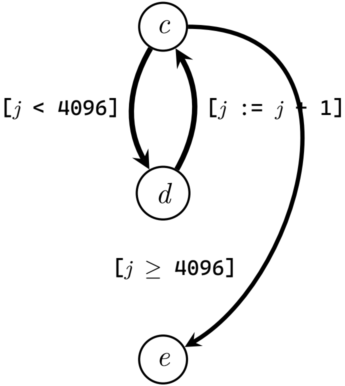
The interpretation over-approximates semantics of all finite paths through the program.

Example: APA Step 1

Consider program `while (j < 4096) j++`.

Example: APA Step 1

Consider program **`while (j < 4096) j++`**.
Its control flow graph is



A path expression representing all paths is $(\langle c, d\rangle \cdot \langle d, c\rangle)^{\star} \cdot \langle c, e\rangle$.

Example: APA Step 2

We focus on interpreting the part of the expression that represents the loop $(\langle c, d\rangle \cdot \langle d, c\rangle)^\star$:

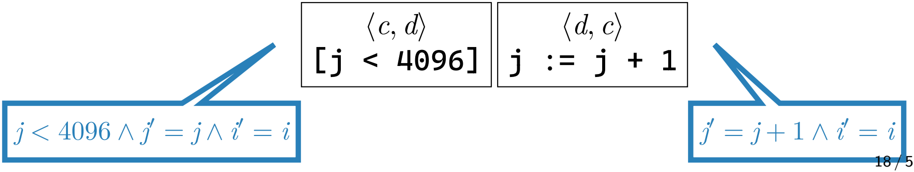
$\langle c, d\rangle$

`[j < 4096]`

$j < 4096 \wedge j' = j \wedge i' = i$

Example: APA Step 2

We focus on interpreting the part of the expression that represents the loop $(\langle c, d\rangle \cdot \langle d, c\rangle)^\star$:



$\langle c, d\rangle$

`[j < 4096]`

$\langle d, c\rangle$

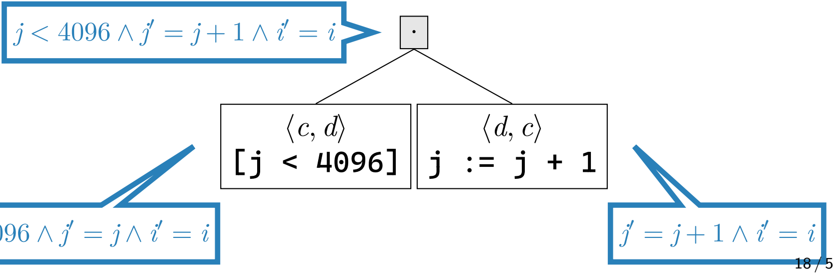
`j := j + 1`

$j < 4096 \wedge j' = j \wedge i' = i$

$j' = j + 1 \wedge i' = i$

Example: APA Step 2

We focus on interpreting the part of the expression that represents the loop $(\langle c, d\rangle \cdot \langle d, c\rangle)^\star$:



$j < 4096 \wedge j' = j + 1 \wedge i' = i$

$\cdot$

$\langle c, d\rangle$

`[j < 4096]`

$\langle d, c\rangle$

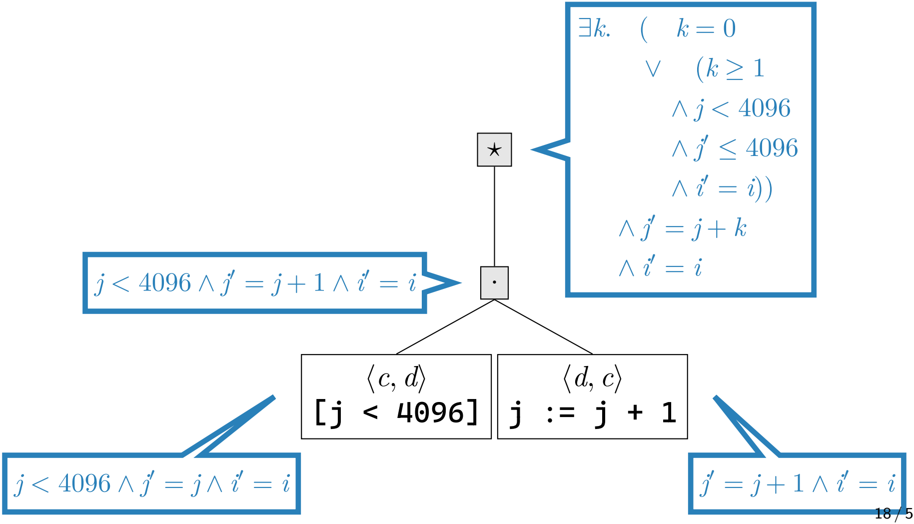
`j := j + 1`

$j < 4096 \wedge j' = j \wedge i' = i$

$j' = j + 1 \wedge i' = i$

Example: APA Step 2

We focus on interpreting the part of the expression that represents the loop $(\langle c, d\rangle \cdot \langle d, c\rangle)^\star$:



$$\begin{aligned}\exists k.\quad (\quad & k = 0\\ \vee\quad & (k \geq 1\\ & \quad \wedge j < 4096\\ & \quad \wedge j' \leq 4096\\ & \quad \wedge i' = i))\\ \wedge\, & j' = j + k\\ \wedge\, & i' = i\end{aligned}$$

$j < 4096 \wedge j' = j + 1 \wedge i' = i$

$j < 4096 \wedge j' = j \wedge i' = i$

$j' = j + 1 \wedge i' = i$

Monotone recurrences through convex hulls

Define the convex hull of an linear integer arithmetic (LIA) formula $F(Y)$, *conv*(F), to be the strongest formula of the form $AY \geq b$ that is entailed by F, where A is an integer matrix and b is an integer vector. There exist practical algorithms for this [FK15].

We can extract recurrences by invoking this procedure.

Monotone recurrences through convex hulls

Define the convex hull of an linear integer arithmetic (LIA) formula $F(Y)$, *conv*(F), to be the strongest formula of the form $AY \geq b$ that is entailed by F, where A is an integer matrix and b is an integer vector. There exist practical algorithms for this [FK15].

We can extract recurrences by invoking this procedure.

1. Compute *conv*$(\exists X, X'.F(X, X') \wedge \bigwedge_{x \in X} \delta_x = x' - x)$.
2. We get the strongest consequence among those with form $A\delta_x \geq b$.
3. This entails that $F \models Ax' \geq Ax + b$ and we get the strongest such formula.

Invariant generation using recurrences

Generating monotone linear loop summaries for APA [ZK21b].

```
while (x >= 0
    && y >= 0) {
  if (*) {
    x--;
  } else {
    y--;
  }
}
```

Invariant generation using recurrences

Generating monotone linear loop summaries for APA [ZK21b].

1. Get strongest recurrence relations.

$$
\begin{aligned}
F \models \quad & (x' + y') = (x + y) - 1 \\
& \wedge x' \leq x \wedge x' \geq x - 1 \\
& \wedge y' \leq y \wedge y' \geq y - 1
\end{aligned}
$$

```
while (x >= 0
    && y >= 0) {
  if (*) {
    x--;
  } else {
    y--;
  }
}
```

Invariant generation using recurrences

Generating monotone linear loop summaries for APA [ZK21b].

❶ Get strongest recurrence relations.

$$\begin{aligned} F \models \quad & (x' + y') = (x + y) - 1 \\ & \wedge x' \leq x \wedge x' \geq x - 1 \\ & \wedge y' \leq y \wedge y' \geq y - 1 \end{aligned}$$

```
while (x >= 0
    && y >= 0) {
  if (*) {
    x--;
  } else {
    y--;
  }
}
```

❷ Solve the symbolic system of recurrences.

$$\begin{aligned} F^\star \triangleq \exists k. k \geq 0 \wedge (x' + y') = (x + y) - k \\ \wedge x' \leq x \wedge x' \geq x - k \\ \wedge y' \leq y \wedge y' \geq y - k \end{aligned}$$

❸ Summarize the behavior of the loop using $F^\star$.

Outline

Introduction

Background and Preliminaries

A Framework for Compositional and Monotone Termination Analysis

A Weak Theory of Nonlinear Arithmetics with Applications

Experimental Evaluation

Takeaways

ComPACT

A framework for *practical*, *compositional* termination analyses with *monotonicity* guarantees that extends the algebraic program analysis framework.

ComPACT

A framework for *practical*, *compositional* termination analyses with *monotonicity* guarantees that extends the algebraic program analysis framework.

Contributions

- We extend Tarjan's method [Tar81a] to compute path expressions for infinite paths.
- We extend algebraic program analysis to handle infinite paths.
- We present a collection of monotone termination analyses that instantiate the framework.

Recipe comparison

Original algebraic program analysis recipe:

1. Compute regular expressions that represent finite program paths
2. Define the analysis as interpretations of operators

Recipe comparison

Original algebraic program analysis recipe:

1. Compute regular expressions that represent finite program paths
2. Define the analysis as interpretations of operators

ComPACT:

1. Compute ω-regular expressions that represent infinite program paths

Recipe comparison

Original algebraic program analysis recipe:

1. Compute regular expressions that represent finite program paths
2. Define the analysis as interpretations of operators

ComPACT:

1. Compute ω-regular expressions that represent infinite program paths
2. Define the analysis as interpretations of operators
    - $\star$ operator – approximate the transitive closure of loops
    - ω operator – obtain terminating conditions for a loop
    - $\cdot$ operator – propagate terminating conditions
    - $+$ operator – combine terminating conditions for different paths

Recipe comparison

Original algebraic program analysis recipe:

1. Compute regular expressions that represent finite program paths
2. Define the analysis as interpretations of operators

ComPACT:

1. Compute ω-regular expressions that represent infinite program paths
2. Define the analysis as interpretations of operators
   - $\star$ operator – approximate the transitive closure of loops
   - ω operator – obtain terminating conditions for a loop
   - $\cdot$ operator – propagate terminating conditions
   - $+$ operator – combine terminating conditions for different paths
3. The interpretation is a condition under which the whole program terminates

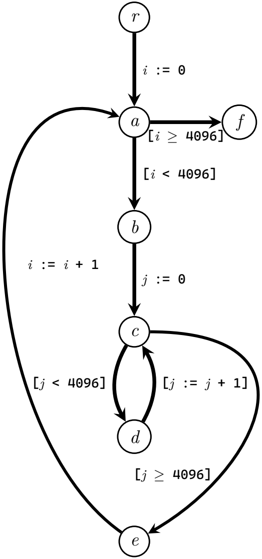
Method Overview

```
for (int i = 0; i < 4096; i++)
  for (int j = 0; j < 4096; j++)
    ; // skip
```

Method Overview

```
for (int i = 0; i < 4096; i++)
  for (int j = 0; j < 4096; j++)
    ; // skip
```

1. CFG



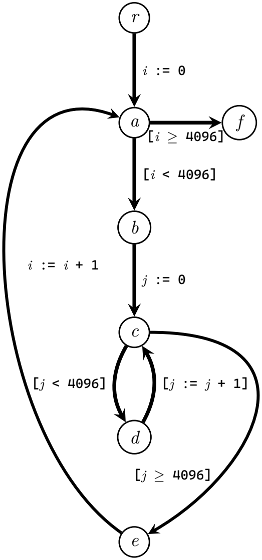
Method Overview

```
for (int i = 0; i < 4096; i++)
  for (int j = 0; j < 4096; j++)
    ; // skip
```

1. CFG

2. Path expression

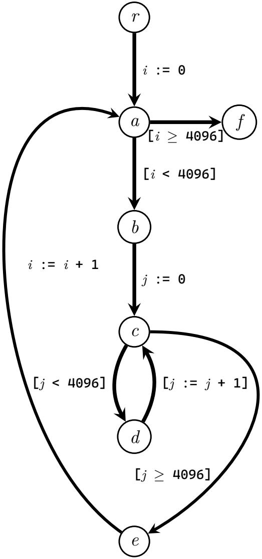
$$\langle r, a\rangle \left(\begin{array}{l} \overbrace{(\langle a, b\rangle\langle b, c\rangle(\langle c, d\rangle\langle d, c\rangle)^{*}\langle c, e\rangle\langle e, a\rangle)^{\omega}}^{\text{outer loop}} \\ +(\langle a, b\rangle\langle b, c\rangle(\langle c, d\rangle\langle d, c\rangle)^{*}\langle c, e\rangle\langle e, a\rangle)^{*}\langle a, b\rangle\langle b, c\rangle\underbrace{(\langle c, d\rangle\langle d, c\rangle)^{\omega}}_{\text{inner loop}} \end{array} \right)$$



Method Overview

```
for (int i = 0; i < 4096; i++)
  for (int j = 0; j < 4096; j++)
    ; // skip
```

1. CFG



2. Path expression

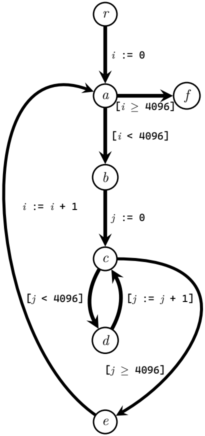
$$\langle r, a\rangle \left(\begin{array}{l} \overbrace{(\langle a, b\rangle\langle b, c\rangle(\langle c, d\rangle\langle d, c\rangle)^*\langle c, e\rangle\langle e, a\rangle)^\omega}^{\text{outer loop}} \\ + (\langle a, b\rangle\langle b, c\rangle(\langle c, d\rangle\langle d, c\rangle)^*\langle c, e\rangle\langle e, a\rangle)^*\langle a, b\rangle\langle b, c\rangle \underbrace{(\langle c, d\rangle\langle d, c\rangle)^\omega}_{\text{inner loop}} \end{array} \right)$$

3. Terminating precondition

true

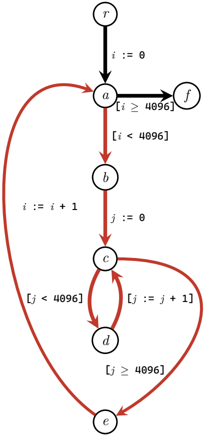
ω-regular expressions

$$\langle r, a\rangle\left(\begin{array}{l}\overbrace{(\langle a, b\rangle\langle b, c\rangle(\langle c, d\rangle\langle d, c\rangle)^*\langle c, e\rangle\langle e, a\rangle)^\omega}^{\text{outer loop}}\\ +(\langle a, b\rangle\langle b, c\rangle(\langle c, d\rangle\langle d, c\rangle)^*\langle c, e\rangle\langle e, a\rangle)^*\langle a, b\rangle\langle b, c\rangle\underbrace{(\langle c, d\rangle\langle d, c\rangle)^\omega}_{\text{inner loop}}\end{array}\right)$$



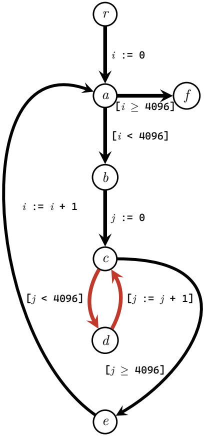
ω-regular expressions

$$\langle r, a\rangle \left(\begin{array}{l} \overbrace{(\langle a, b\rangle\langle b, c\rangle(\langle c, d\rangle\langle d, c\rangle)^*\langle c, e\rangle\langle e, a\rangle)^\omega}^{\text{outer loop}} \\ +(\langle a, b\rangle\langle b, c\rangle(\langle c, d\rangle\langle d, c\rangle)^*\langle c, e\rangle\langle e, a\rangle)^*\langle a, b\rangle\langle b, c\rangle\,\underbrace{(\langle c, d\rangle\langle d, c\rangle)^\omega}_{\text{inner loop}} \end{array} \right)$$



ω-regular expressions

$$\langle r, a\rangle \left(\begin{array}{l} \overbrace{(\langle a, b\rangle\langle b, c\rangle(\langle c, d\rangle\langle d, c\rangle)^*\langle c, e\rangle\langle e, a\rangle)^\omega}^{\text{outer loop}} \\ +(\langle a, b\rangle\langle b, c\rangle(\langle c, d\rangle\langle d, c\rangle)^*\langle c, e\rangle\langle e, a\rangle)^*\langle a, b\rangle\langle b, c\rangle\underbrace{(\langle c, d\rangle\langle d, c\rangle)^\omega}_{\text{inner loop}} \end{array} \right)$$



Interpreting ω-regular path expressions

Path expressions are interpreted compositionally in a bottom-up manner.

- **TF**: a regular algebra of transition formulas
- **SF**: an ω-regular algebra of state formulas that represents sufficient preconditions for termination

Interpreting ω-regular path expressions

Path expressions are interpreted compositionally in a bottom-up manner.

- **TF**: a regular algebra of transition formulas
- **SF**: an ω-regular algebra of state formulas that represents sufficient preconditions for termination

$$\mathbf{TF}[\![0]\!] \triangleq \mathit{false}$$

$$\mathbf{TF}[\![1]\!] \triangleq \bigwedge_{x \in \mathit{Var}} x' = x$$

Interpreting ω-regular path expressions

Path expressions are interpreted compositionally in a bottom-up manner.

- **TF**: a regular algebra of transition formulas
- **SF**: an ω-regular algebra of state formulas that represents sufficient preconditions for termination

$$\mathbf{TF}[\![0]\!] \triangleq \mathit{false}$$

composition of relations

$$\mathbf{TF}[\![1]\!] \triangleq \bigwedge_{x \in \mathit{Var}}$$

$$\mathbf{TF}[\![t_1 \cdot t_2]\!] \triangleq \mathbf{TF}[\![t_1]\!] \circ \mathbf{TF}[\![t_2]\!]$$

Interpreting ω-regular path expressions

Path expressions are interpreted compositionally in a bottom-up manner.

- **TF**: a regular algebra of transition formulas
- **SF**: an ω-regular algebra of state formulas that represents sufficient preconditions for termination

$$\begin{aligned} \mathbf{TF}[\![0]\!] &\triangleq \textit{false} \\ \mathbf{TF}[\![1]\!] &\triangleq \bigwedge_{x \in \textit{Var}} x' = x \\ \mathbf{TF}[\![t_1 \cdot t_2]\!] &\triangleq \mathbf{TF}[\![t_1]\!] \circ \mathbf{TF}[\![t_2]\!] \\ \mathbf{TF}[\![t_1 + t_2]\!] &\triangleq \mathbf{TF}[\![t_1]\!] \vee \mathbf{TF}[\![t_2]\!] \end{aligned}$$

Interpreting ω-regular path expressions

Path expressions are interpreted compositionally in a bottom-up manner.

- **TF**: a regular algebra of transition formulas
- **SF**: an ω-regular algebra of state formulas that represents sufficient preconditions for termination

$$\mathbf{TF}[\![0]\!] \triangleq \mathit{false}$$

$$\mathbf{TF}[\![1]\!] \triangleq \bigwedge_{x \in \mathit{Var}} x' = x$$

$\mathbf{TF}[\![t$

approx. transitive closure [FK15]

$\mathbf{TF}[\![t_1 + t_2]\!] \triangleq \mathbf{TF}[\![t_1]\!] \vee$ $_2]\!]$

$$\mathbf{TF}[\![t^\star]\!] \triangleq (\mathbf{TF}[\![t]\!])^*$$

Interpreting ω-regular path expressions

Path expressions are interpreted compositionally in a bottom-up manner.

- **TF**: a regular algebra of transition formulas
- **SF**: an ω-regular algebra of state formulas that represents sufficient preconditions for termination

$$\begin{aligned}
\mathbf{TF}\llbracket 0 \rrbracket &\triangleq \mathit{false} \\
\mathbf{TF}\llbracket 1 \rrbracket &\triangleq \bigwedge_{x \in \mathit{Var}} x' = x \\
\mathbf{TF}\llbracket t_1 \cdot t_2 \rrbracket &\triangleq \mathbf{TF}\llbracket t_1 \rrbracket \circ \mathbf{TF}\llbracket t_2 \rrbracket \\
\mathbf{TF}\llbracket t_1 + t_2 \rrbracket &\triangleq \mathbf{TF}\llbracket t_1 \rrbracket \vee \mathbf{TF}\llbracket t_2 \rrbracket \\
\mathbf{TF}\llbracket t^\star \rrbracket &\triangleq (\mathbf{TF}\llbracket t \rrbracket)^*
\end{aligned}$$

mortal precondition

$$\mathbf{SF}\llbracket t^\omega \rrbracket \triangleq \mathit{mp}(\mathbf{TF}\llbracket t \rrbracket)$$

Interpreting ω-regular path expressions

Path expressions are interpreted compositionally in a bottom-up manner.

- **TF**: a regular algebra of transition formulas
- **SF**: an ω-regular algebra of state formulas that represents sufficient preconditions for termination

$$\mathbf{TF}[\![0]\!] \triangleq \mathit{false}$$

$$\mathbf{TF}[\![1]\!] \triangleq \bigwedge_{x \in \mathit{Var}} x' = x$$

$$\mathbf{TF}[\![t_1 \cdot t_2]\!] \triangleq \mathbf{TF}[\![t_1]\!] \circ \mathbf{TF}[\![t_2]\!]$$

$$\mathbf{TF}[\![t_1 + t_2]\!] \triangleq \mathbf{TF}[\![t_1]\!] \vee \mathbf{TF}[\![t_2]\!]$$

$$\mathbf{TF}[\![t^\star]\!] \triangleq (\mathbf{T}$$

$$\mathbf{SF}[\![t^\omega]\!] \triangleq \mathit{mp}(\mathbf{TF}[\![t]\!])$$

$$\mathbf{SF}[\![t \cdot s]\!] \triangleq \mathit{wp}(\mathbf{TF}[\![t]\!], \mathbf{SF}[\![s]\!])$$

$$\mathit{wp}(F, S) \triangleq \forall X'. F(X, X') \implies S[X \mapsto X']$$

Interpreting ω-regular path expressions

Path expressions are interpreted compositionally in a bottom-up manner.

- **TF**: a regular algebra of transition formulas
- **SF**: an ω-regular algebra of state formulas that represents sufficient preconditions for termination

$$\begin{aligned}
\mathbf{TF}[\![0]\!] &\triangleq \mathit{false}\\
\mathbf{TF}[\![1]\!] &\triangleq \bigwedge_{x \in \mathit{Var}} x' = x\\
\mathbf{TF}[\![t_1 \cdot t_2]\!] &\triangleq \mathbf{TF}[\![t_1]\!] \circ \mathbf{TF}[\![t_2]\!]\\
\mathbf{TF}[\![t_1 + t_2]\!] &\triangleq \mathbf{TF}[\![t_1]\!] \vee \mathbf{TF}[\![t_2]\!]\\
\mathbf{TF}[\![t^\star]\!] &\triangleq (\mathbf{TF}[\![t]\!])^*
\end{aligned}$$

$$\begin{aligned}
\mathbf{SF}[\![t^\omega]\!] &\triangleq \mathit{mp}(\mathbf{TF}[\![t]\!])\\
\mathbf{SF}[\![t \cdot s]\!] &\triangleq \mathit{wp}(\mathbf{TF}[\![t]\!], \mathbf{SF}[\![s]\!])\\
\mathbf{SF}[\![s_1 + s_2]\!] &\triangleq \mathbf{SF}[\![s_1]\!] \wedge \mathbf{SF}[\![s_2]\!]
\end{aligned}$$

Interpreting ω-regular path expressions

Path expressions are interpreted compositionally in a bottom-up manner.

- **TF**: a regular algebra of transition formulas
- **SF**: an ω-regular algebra of state formulas that represents sufficient preconditions for termination

$$
\begin{aligned}
\mathbf{TF}[\![0]\!] &\triangleq \mathit{false} \\
\mathbf{TF}[\![1]\!] &\triangleq \bigwedge_{x \in \mathit{Var}} x' = x \\
\mathbf{TF}[\![t_1 \cdot t_2]\!] &\triangleq \mathbf{TF}[\![t_1]\!] \circ \mathbf{TF}[\![t_2]\!] \\
\mathbf{TF}[\![t_1 + t_2]\!] &\triangleq \mathbf{TF}[\![t_1]\!] \vee \mathbf{TF}[\![t_2]\!] \\
\mathbf{TF}[\![t^\star]\!] &\triangleq (\mathbf{TF}[\![t]\!])^*
\end{aligned}
$$

$$
\begin{aligned}
\mathbf{SF}[\![t^\omega]\!] &\triangleq \mathit{mp}(\mathbf{TF}[\![t]\!]) \\
\mathbf{SF}[\![t \cdot s]\!] &\triangleq \mathit{wp}(\mathbf{TF}[\![t]\!], \mathbf{SF}[\![s]\!]) \\
\mathbf{SF}[\![s_1 + s_2]\!] &\triangleq \mathbf{SF}[\![s_1]\!] \wedge \mathbf{SF}[\![s_2]\!]
\end{aligned}
$$

- Analyzing loop summaries makes it decidable

Interpreting ω-regular path expressions

Path expressions are interpreted compositionally in a bottom-up manner.

- **TF**: a regular algebra of transition formulas
- **SF**: an ω-regular algebra of state formulas that represents sufficient preconditions for termination

$$
\begin{aligned}
\mathbf{TF}[\![0]\!] &\triangleq \mathit{false} \\
\mathbf{TF}[\![1]\!] &\triangleq \bigwedge_{x \in \mathit{Var}} x' = x \\
\mathbf{TF}[\![t_1 \cdot t_2]\!] &\triangleq \mathbf{TF}[\![t_1]\!] \circ \mathbf{TF}[\![t_2]\!] \\
\mathbf{TF}[\![t_1 + t_2]\!] &\triangleq \mathbf{TF}[\![t_1]\!] \vee \mathbf{TF}[\![t_2]\!] \\
\mathbf{TF}[\![t^\star]\!] &\triangleq (\mathbf{TF}[\![t]\!])^*
\end{aligned}
$$

$$
\begin{aligned}
\mathbf{SF}[\![t^\omega]\!] &\triangleq \mathit{mp}(\mathbf{TF}[\![t]\!]) \\
\mathbf{SF}[\![t \cdot s]\!] &\triangleq \mathit{wp}(\mathbf{TF}[\![t]\!], \mathbf{SF}[\![s]\!]) \\
\mathbf{SF}[\![s_1 + s_2]\!] &\triangleq \mathbf{SF}[\![s_1]\!] \wedge \mathbf{SF}[\![s_2]\!]
\end{aligned}
$$

- Analyzing loop summaries makes it decidable
- Other operators are monotone, only need monotone $\star$ and *mp* operators

Monotone mortal precondition operators (1)

Mortal precondition operator *mp*

$mp(F)$ returns a sufficient condition for a loop with body formula F to terminate.

Monotone mortal precondition operators (1)

Mortal precondition operator *mp*

mp(F) returns a sufficient condition for a loop with body formula F to terminate.

Monotonicity

If $F \models G$ then *mp*$(G) \models$ *mp*(F).

Monotone mortal precondition operators (2)

We present the following operations that generate mortal preconditions.

Monotone mortal precondition operators (2)

We present the following operations that generate mortal preconditions.

- *mp* based on LLRF: returns true if there is a lexicographic linear ranking function for the transition formula [GMR15]

Monotone mortal precondition operators (2)

We present the following operations that generate mortal preconditions.

- *mp* based on LLRF: returns true if there is a lexicographic linear ranking function for the transition formula [GMR15]
- *mp* based on PRF and LPRF: returns true if there is a (lexicographic) polynomial ranking function (next part)

Monotone mortal precondition operators (2)

We present the following operations that generate mortal preconditions.

- *mp* based on LLRF: returns true if there is a lexicographic linear ranking function for the transition formula [GMR15]
- *mp* based on PRF and LPRF: returns true if there is a (lexicographic) polynomial ranking function (next part)
- *mp* based on transitive closure: given a logical representation of transitive closure, encodes the constraint that all computations have bounded length

Monotone mortal precondition operators (2)

We present the following operations that generate mortal preconditions.

- *mp* based on LLRF: returns true if there is a lexicographic linear ranking function for the transition formula [GMR15]
- *mp* based on PRF and LPRF: returns true if there is a (lexicographic) polynomial ranking function (next part)
- *mp* based on transitive closure: given a logical representation of transitive closure, encodes the constraint that all computations have bounded length
- *mp* combinator based on phase structure: partitions the loop into *phases* and analyzes how program evolves across phases

Monotone mortal precondition operators (2)

We present the following operations that generate mortal preconditions.

- *mp* based on LLRF: returns true if there is a lexicographic linear ranking function for the transition formula [GMR15]
- *mp* based on PRF and LPRF: returns true if there is a (lexicographic) polynomial ranking function (next part)
- *mp* based on transitive closure: given a logical representation of transitive closure, encodes the constraint that all computations have bounded length
- *mp* combinator based on phase structure: partitions the loop into *phases* and analyzes how program evolves across phases
- mp_{LDS} based on abstracting the loop into a linear dynamical system

mp through linear dynamical system abstraction

Linear loops (linear dynamical systems)

```
while (G(x)) {  // guard is conjunctive
  x = A x;    // matrix multiplication
}
```

Symbolic closed-forms easy to compute, also line of work on decidability of termination of linear loops

- Over the reals [Tiw04]
- Over the rationals [Bra06]
- Over the integers [HOW19]

A monotone *mp* in two steps

Transfer techniques developed for linear loops to reason about termination of general loops:

- Compute for any loop a *best abstraction* within a particular class of linear dynamical systems.

A monotone *mp* in two steps

Transfer techniques developed for linear loops to reason about termination of general loops:

- Compute for any loop a *best abstraction* within a particular class of linear dynamical systems.
- Generate terminating conditions for these linear dynamical systems.

Simulation allows us to consider simpler systems

Let $(A, \xrightarrow{A})$ and $(B, \xrightarrow{B})$ be transition systems over linear state space. A simulation $S : A \to B$ maps transitions in A to those in B, with inverse S^{-1}.

Simulation allows us to consider simpler systems

Let $(A, \xrightarrow{A})$ and $(B, \xrightarrow{B})$ be transition systems over linear state space.
A simulation $S : A \to B$ maps transitions in A to those in B, with inverse S^{-1}.

$$\begin{tikzcd}
a \arrow[r, "A"] & a'
\end{tikzcd}$$

Simulation allows us to consider simpler systems

Let $(A, \xrightarrow{A})$ and $(B, \xrightarrow{B})$ be transition systems over linear state space.
A simulation $S : A \to B$ maps transitions in A to those in B, with inverse S^{-1}.

$$\begin{tikzcd}
a \arrow[r, "A"] \arrow[d, "S"'] & a' \arrow[d, "S"] \\
S(a) \arrow[r, "B"'] & S(a')
\end{tikzcd}$$

Simulation allows us to consider simpler systems

Let $(A, \xrightarrow{A})$ and $(B, \xrightarrow{B})$ be transition systems over linear state space.
A simulation $S : A \to B$ maps transitions in A to those in B, with inverse S^{-1}.

$$\begin{tikzcd}
a \arrow[r, "A"] \arrow[d, "S"'] & a' \arrow[d, "S"] \\
S(a) \arrow[r, "B"'] & S(a')
\end{tikzcd}$$

A simulation S is linear if S is a linear function.

Simulation allows us to consider simpler systems

Let $(A, \xrightarrow{A})$ and $(B, \xrightarrow{B})$ be transition systems over linear state space.
A simulation $S : A \to B$ maps transitions in A to those in B, with inverse S^{-1}.

$$\begin{tikzcd}
a \arrow[r, "A"] \arrow[d, "S"'] & a' \arrow[d, "S"] \\
S(a) \arrow[r, "B"'] & S(a')
\end{tikzcd}$$

A simulation S is linear if S is a linear function.
If B terminates starting from states within set X, then $S^{-1}(X)$ leads to termination of A.

Simulation allows us to consider simpler systems

Let $(A, \xrightarrow{A})$ and $(B, \xrightarrow{B})$ be transition systems over linear state space.
A simulation $S : A \to B$ maps transitions in A to those in B, with inverse S^{-1}.

$$\begin{tikzcd}
a \arrow[r, "A"] \arrow[d, "S"'] & a' \arrow[d, "S"] \\
S(a) \arrow[r, "B"'] & S(a')
\end{tikzcd}$$

A simulation S is linear if S is a linear function.
If B terminates starting from states within set X, then $S^{-1}(X)$ leads to termination of A. Suppose A is hard to analyze while B is not, then we may use $S^{-1}(\mathit{mp}(B)) \subseteq \mathit{mp}(A)$ to get some mortal preconditions for A.

Example of a linear simulation

```
while (x + y >= 0) {
  if (*) {
    x = x - z;
  } else {
    y = y - z;
  }
}
```

Example of a linear simulation

```
while (x + y >= 0) {
  if (*) {
    x = x - z;
  } else {
    y = y - z;
  }
}
```

$$\begin{bmatrix} a \\ b \end{bmatrix} = \begin{bmatrix} 1 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix}$$

Example of a linear simulation

```
while (x + y >= 0) {
  if (*) {
    x = x - z;
  } else {
    y = y - z;
  }
}
```

$$\begin{bmatrix} a \\ b \end{bmatrix} = \begin{bmatrix} 1 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix}$$

$$\dashrightarrow$$

`while` $(a \geq 0)$

$$\begin{bmatrix} a \\ b \end{bmatrix} := \begin{bmatrix} 1 & -1 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} a \\ b \end{bmatrix}$$

Example of a linear simulation

```
while (x + y >= 0) {
  if (*) {
    x = x - z;
  } else {
    y = y - z;
  }
}
```

$$\begin{bmatrix} a \\ b \end{bmatrix} = \begin{bmatrix} 1 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix}$$

$$\dashrightarrow \quad \texttt{while}\ (a \geq 0) \quad \begin{bmatrix} a \\ b \end{bmatrix} := \begin{bmatrix} 1 & -1 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} a \\ b \end{bmatrix}$$

For the linear loop on the right, $a' = a - kb$ after k iterations. Thus a mortal precondition is $b > 0$, which implies that a mortal precondition for the original loop is $z = b > 0$.

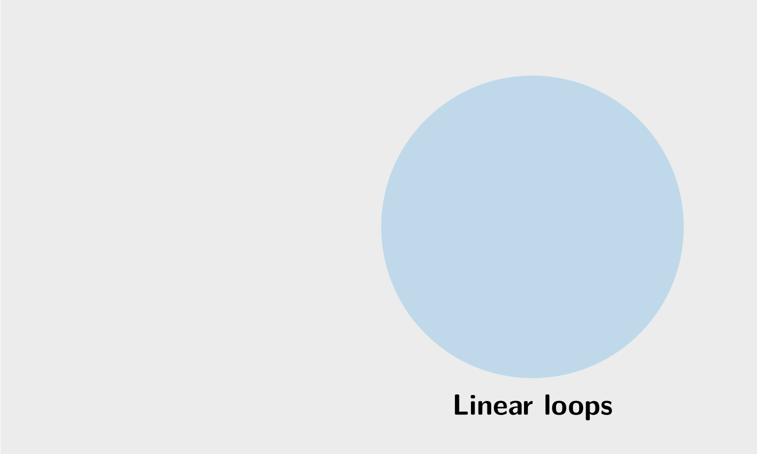
Which linear abstraction to use?

Q: There are many such linear loops that abstract (i.e., soundly over-approximate) the original loop, which one should we use?

Which linear abstraction to use?

Q: There are many such linear loops that abstract (i.e., soundly over-approximate) the original loop, which one should we use?
A: Use the *best* one which yields the weakest mortal precondition for the original loop (omitting a bunch of linear algebra and category theoretic details here)!

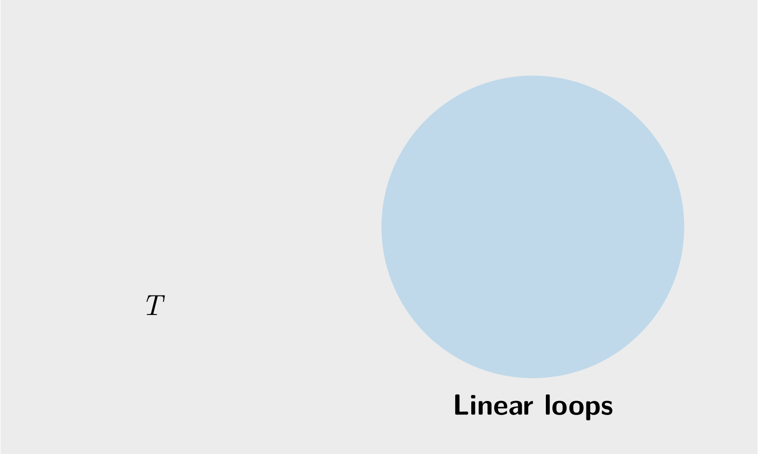
Best abstractions leads to weakest *mp*



Linear loops

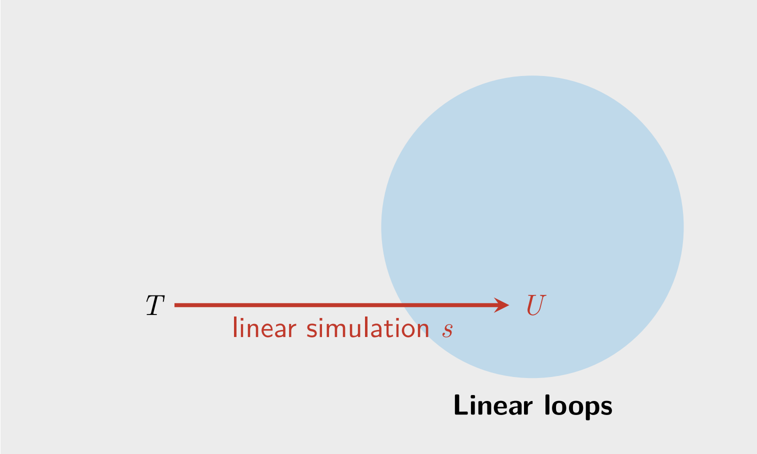
Transition systems

Best abstractions leads to weakest *mp*



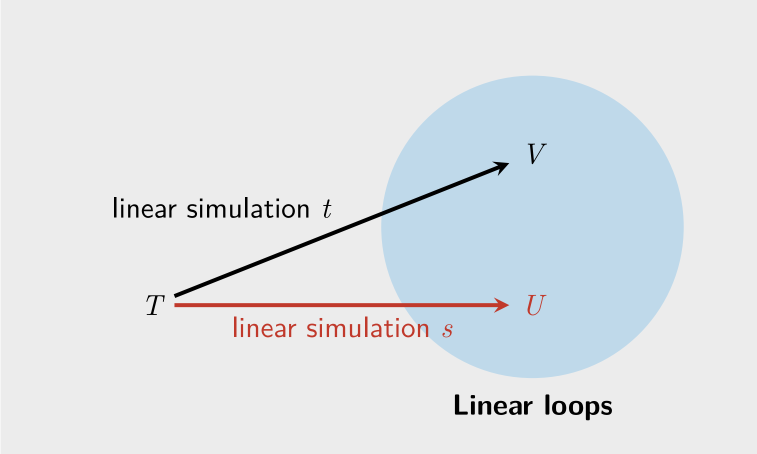
Transition systems

Best abstractions leads to weakest *mp*



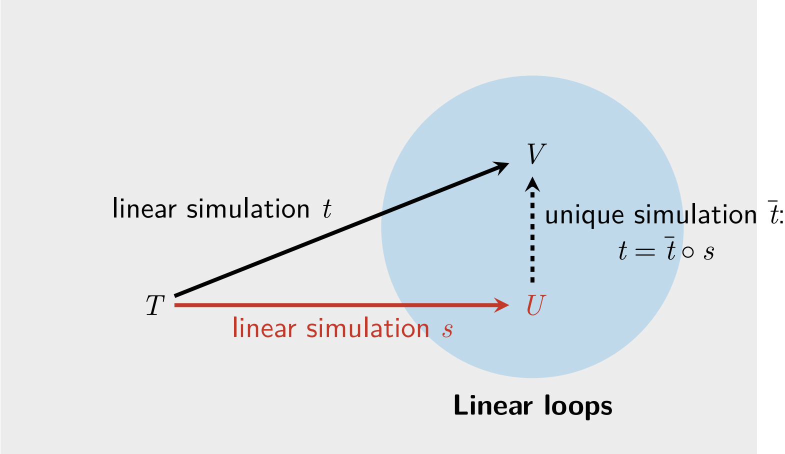
Transition systems

Best abstractions leads to weakest *mp*



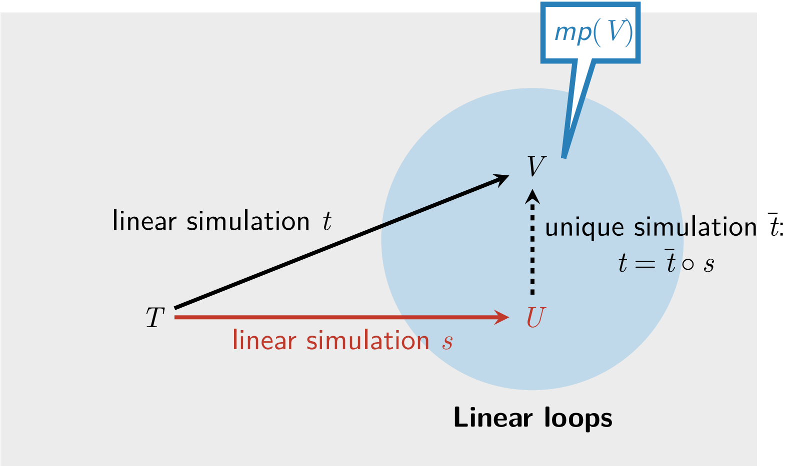
Transition systems

Best abstractions leads to weakest *mp*



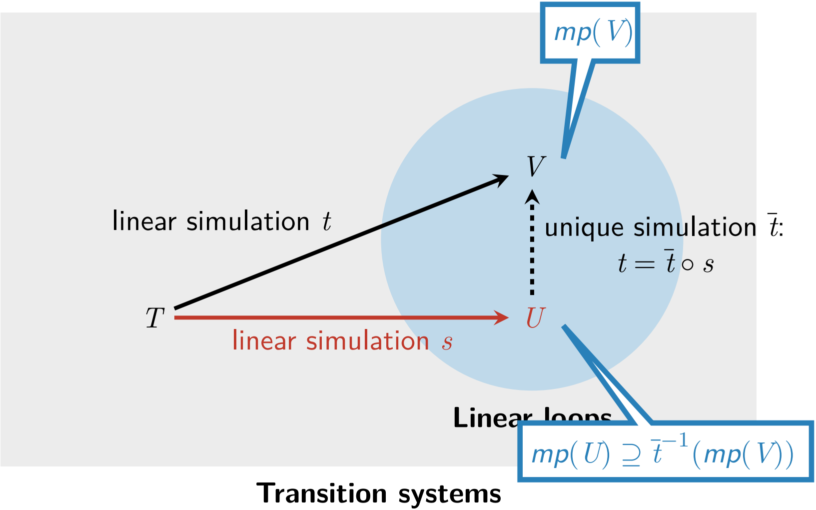
Transition systems

Best abstractions leads to weakest *mp*

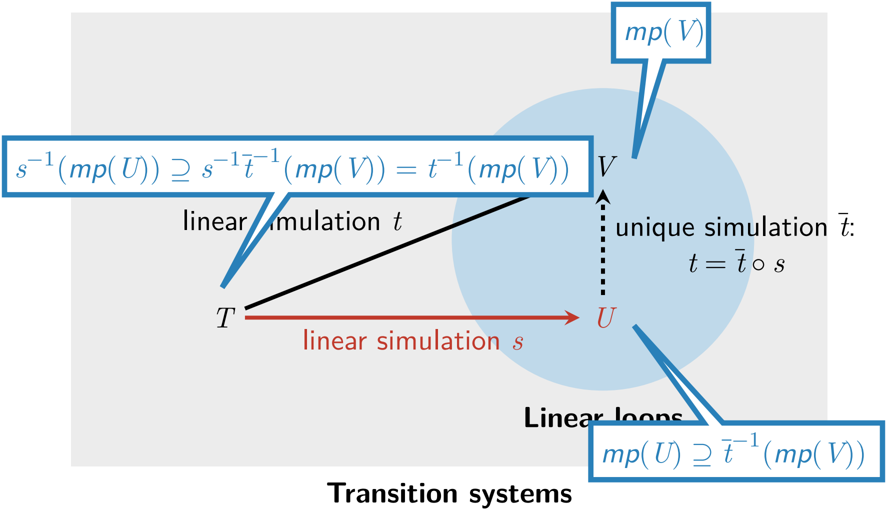


Transition systems

Best abstractions leads to weakest *mp*



Best abstractions leads to weakest *mp*

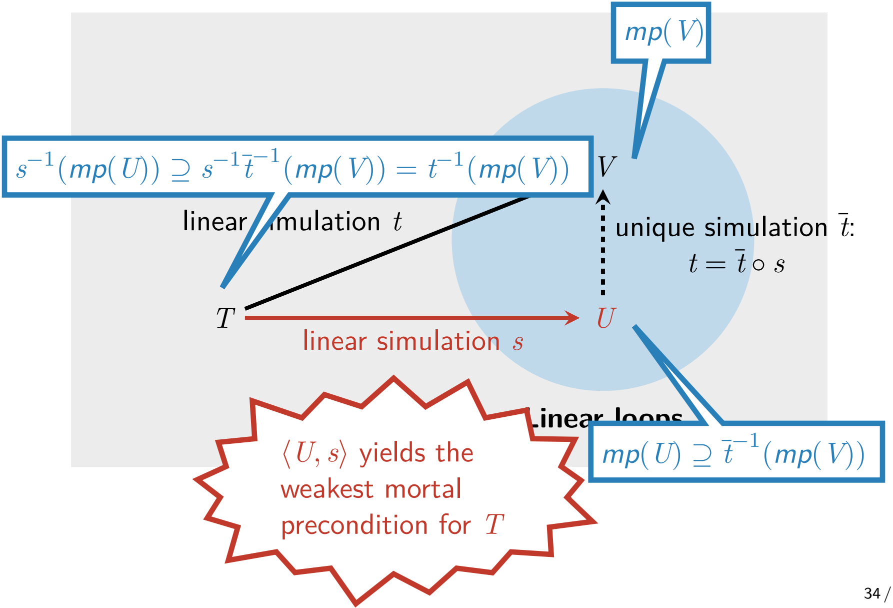


$mp(V)$

$s^{-1}(mp(U)) \supseteq s^{-1}\bar{t}^{-1}(mp(V)) = t^{-1}(mp(V))$

$mp(U) \supseteq \bar{t}^{-1}(mp(V))$

Best abstractions leads to weakest *mp*



$mp(V)$

$s^{-1}(mp(U)) \supseteq s^{-1}\bar{t}^{-1}(mp(V)) = t^{-1}(mp(V))$

linear imulation t

unique simulation $\bar{t}$:
$t = \bar{t} \circ s$

linear simulation s

$mp(U) \supseteq \bar{t}^{-1}(mp(V))$

$\langle U, s \rangle$ yields the
weakest mortal
precondition for T

Key results for linear dynamical system abstraction

Step I: compute best abstractions

For any transition system T, we can compute its best abstraction within a restricted class of linear loops whose asymptotic behavior is easy to analyze.

Key results for linear dynamical system abstraction

Step I: compute best abstractions

For any transition system T, we can compute its best abstraction within a restricted class of linear loops whose asymptotic behavior is easy to analyze.

Step II: compute mortal preconditions for linear loops

Given a guard formula $G(\mathbf{x})$ and a linear map $\mathbf{x}' = A\mathbf{x}$ with certain restrictions on A, we can compute mortal preconditions by analyzing the asymptotic behavior of the symbolic closed form $G(A^k\mathbf{x})$.

Example for step II

This method is inspired by Tiwari [Tiw04].

```
while (a >= 0) {
  a = a - b;
}
```

Example for step II

This method is inspired by Tiwari [Tiw04].

```
while (a >= 0) {
  a = a - b;
}
```

Loop body in matrix form

$$\begin{bmatrix} a' \\ b' \end{bmatrix} = \begin{bmatrix} 1 & -1 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} a \\ b \end{bmatrix}$$

Example for step II

This method is inspired by Tiwari [Tiw04].

```
while (a >= 0) {
  a = a - b;
}
```

Loop body in matrix form

$$\begin{bmatrix} a' \\ b' \end{bmatrix} = \begin{bmatrix} 1 & -1 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} a \\ b \end{bmatrix}$$

Closed form

$$\begin{bmatrix} a^{(k)} \\ b^{(k)} \end{bmatrix} = \begin{bmatrix} 1 & -k \\ 0 & 1 \end{bmatrix} \begin{bmatrix} a \\ b \end{bmatrix}$$

Example for step II

This method is inspired by Tiwari [Tiw04].

```
while (a >= 0) {
  a = a - b;
}
```

Loop body in matrix form

$$\begin{bmatrix} a' \\ b' \end{bmatrix} = \begin{bmatrix} 1 & -1 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} a \\ b \end{bmatrix}$$

Closed form

$$\begin{bmatrix} a^{(k)} \\ b^{(k)} \end{bmatrix} = \begin{bmatrix} 1 & -k \\ 0 & 1 \end{bmatrix} \begin{bmatrix} a \\ b \end{bmatrix}$$

Closed form for **`a`** in the guard

$$a^{(k)} = a - bk$$

Example for step II

This method is inspired by Tiwari [Tiw04].

```
while (a >= 0) {
  a = a - b;
}
```

Loop body in matrix form

$$\begin{bmatrix} a' \\ b' \end{bmatrix} = \begin{bmatrix} 1 & -1 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} a \\ b \end{bmatrix}$$

Closed form

$$\begin{bmatrix} a^{(k)} \\ b^{(k)} \end{bmatrix} = \begin{bmatrix} 1 & -k \\ 0 & 1 \end{bmatrix} \begin{bmatrix} a \\ b \end{bmatrix}$$

Closed form for **a** in the guard

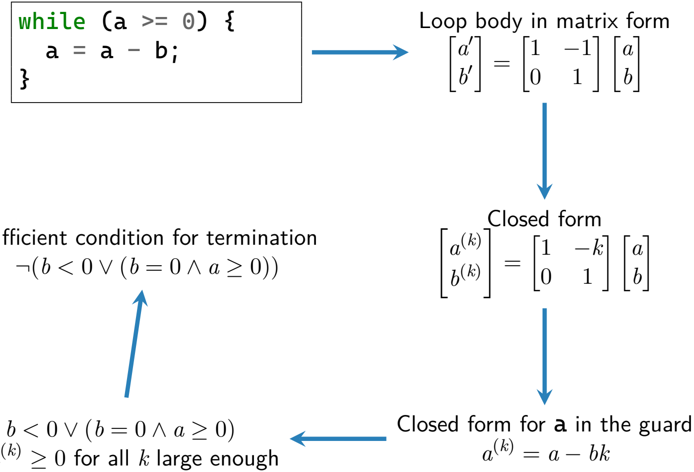
$$a^{(k)} = a - bk$$

$b < 0 \vee (b = 0 \wedge a \geq 0)$

iff $a^{(k)} \geq 0$ for all k large enough

Example for step II

This method is inspired by Tiwari [Tiw04].



```
while (a >= 0) {
  a = a - b;
}
```

Loop body in matrix form

$$\begin{bmatrix} a' \\ b' \end{bmatrix} = \begin{bmatrix} 1 & -1 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} a \\ b \end{bmatrix}$$

Closed form

$$\begin{bmatrix} a^{(k)} \\ b^{(k)} \end{bmatrix} = \begin{bmatrix} 1 & -k \\ 0 & 1 \end{bmatrix} \begin{bmatrix} a \\ b \end{bmatrix}$$

Closed form for **a** in the guard

$$a^{(k)} = a - bk$$

$$b < 0 \lor (b = 0 \land a \geq 0)$$

iff $a^{(k)} \geq 0$ for all k large enough

Sufficient condition for termination

$$\neg(b < 0 \lor (b = 0 \land a \geq 0))$$

Monotonicity of *mp* based on linear abstraction

Monotonicity of mp_{LDS}

Given transition formulas $F \models G$, the condition we generate for F is weaker than what we generate for G.

Monotonicity of *mp* based on linear abstraction

Monotonicity of mp_{LDS}

Given transition formulas $F \models G$, the condition we generate for F is weaker than what we generate for G.

Proof idea: for any transition formula, we have used its best abstraction which yields the weakest possible mortal precondition.

Summary

ComPACT is a practical termination analysis framework, that

- extends algebraic program analysis to handle infinite paths;
- is monotone.

Outline

Introduction

Background and Preliminaries

A Framework for Compositional and Monotone Termination Analysis

A Weak Theory of Nonlinear Arithmetics with Applications

Experimental Evaluation

Takeaways

Linear arithmetic and consequence finding

Linear arithmetic supports *consequence finding*.

Given linear formula F, find the best G of certain form such that $F \models G$.

Linear arithmetic and consequence finding

Linear arithmetic supports *consequence finding*.

Given linear formula F, find the best G of certain form such that $F \models G$.

- Satisfiability: check if `False` is a consequence.

Linear arithmetic and consequence finding

Linear arithmetic supports *consequence finding*.

Given linear formula F, find the best G of certain form such that $F \models G$.

- Satisfiability: check if **`False`** is a consequence.
- Extracting linear recurrences: find all linear inequalities implied by a transition formula T with form $F(X, X') \models cx' \leq cx + d$.

Linear arithmetic and consequence finding

Linear arithmetic supports *consequence finding*.

Given linear formula F, find the best G of certain form such that $F \models G$.

- Satisfiability: check if **`False`** is a consequence.
- Extracting linear recurrences: find all linear inequalities implied by a transition formula T with form $F(X, X') \models cx' \leq cx + d$.
- (Complete) linear ranking function synthesis: find all linear terms cx such that $F(X, X') \models (cx' \leq cx - 1 \land cx \geq 0)$.

Nonlinear arithmetic

Nonlinear integer arithmetic is undecidable, let alone consequence finding!

Nonlinear arithmetic

Nonlinear integer arithmetic is undecidable, let alone consequence finding!
This work presents:

- A decidable theory of nonlinear arithmetic where strongest consequences of certain forms can be computed.
- A scheme for generating nonlinear loop invariants that are monotone with respect to the proposed theory.
- A scheme for synthesizing nonlinear ranking functions that provides a monotone mortal precondition operator for nonlinear loops.

Intuition for generalizing linear invariants

- Invariants based on linear consequences
  1. Extract a system of linear recurrences entailed by the loop. Specifically, linear terms whose changes are bounded by constants.

$$F(X, X') \models \bigwedge_i t_i x' \leq t_i x + c_i$$

  2. Compute the closed form as a loop invariant.

$$F^\star \triangleq \exists k. k \geq 0 \wedge \bigwedge_i t_i x' \leq t_i x + k c_i$$

Intuition for generalizing linear invariants

- Invariants based on linear consequences
  1. Extract a system of linear recurrences entailed by the loop. Specifically, linear terms whose changes are bounded by constants.

     $$F(X, X') \models \bigwedge_i t_i x' \leq t_i x + c_i$$

  2. Compute the closed form as a loop invariant.

     $$F^\star \triangleq \exists k . k \geq 0 \wedge \bigwedge_i t_i x' \leq t_i x + k c_i$$

- Invariants based on nonlinear consequences
  1. Extract a system of recurrences entailed by the loop. Specifically, linear terms whose changes are bounded by invariant polynomials.

     $$F(X, X') \models \bigwedge_i t_i x' \leq t_i x + p_i(x)$$

  2. Compute the closed form as a loop invariant.

     $$F^\star \triangleq \exists k . k \geq 0 \wedge \bigwedge_i t_i x' \leq t_i x + k p_i(x)$$

Example

```
while (*) {
  x = x + z;
  y = y + z;
  t = x - y;
  z = z + t * t;
  if (*)
    w = w + 1;
  else
    w = w + 2;
}
```

Example

1. Invariant linear terms include: $(x - y)$

```
while (*) {
  x = x + z;
  y = y + z;
  t = x - y;
  z = z + t * t;
  if (*)
    w = w + 1;
  else
    w = w + 2;
}
```

Example

1. Invariant linear terms include: $(x - y)$
2. Invariant polynomials generated include: $(x - y), (x - y)^2$

```
while (*) {
  x = x + z;
  y = y + z;
  t = x - y;
  z = z + t * t;
  if (*)
    w = w + 1;
  else
    w = w + 2;
}
```

Example

1. Invariant linear terms include: $(x - y)$
2. Invariant polynomials generated include: $(x - y), (x - y)^2$
3. Extract linear terms whose changes are bounded by constants or invariant polynomials:

$$\begin{aligned} F \models \quad & w + 1 \le w' \le w + 2 \\ & \wedge x' - y' = x - y \\ & \wedge z' = z + (x - y)^2 \end{aligned}$$

```
while (*) {
  x = x + z;
  y = y + z;
  t = x - y;
  z = z + t * t;
  if (*)
    w = w + 1;
  else
    w = w + 2;
}
```

Example

① Invariant linear terms include: $(x - y)$

② Invariant polynomials generated include: $(x - y), (x - y)^2$

③ Extract linear terms whose changes are bounded by constants or invariant polynomials:

$$\begin{aligned} F \models \quad & w + 1 \leq w' \leq w + 2 \\ & \wedge x' - y' = x - y \\ & \wedge z' = z + (x - y)^2 \end{aligned}$$

④ The closed form solution approximates loop behavior:

$$\begin{aligned} F^\star \triangleq \exists k. k \geq 0 \wedge & w + k \leq w' \leq w + 2k \\ & \wedge x' - y' = x - y \\ & \wedge z' = z + k(x - y)^2 \end{aligned}$$

```
while (*) {
  x = x + z;
  y = y + z;
  t = x - y;
  z = z + t * t;
  if (*)
    w = w + 1;
  else
    w = w + 2;
}
```

Generating nonlinear invariants

Given transition formula $F(X, X')$ of a loop body, if we can do consequence findings of certain kinds, then we can implement an intuitive recipe:

Nonlinear invariants through consequence finding

Generating nonlinear invariants

Given transition formula $F(X, X')$ of a loop body, if we can do consequence findings of certain kinds, then we can implement an intuitive recipe:

Nonlinear invariants through consequence finding

1. Compute all consequences of F denoted by $\mathbf{C}(F) \triangleq \{q : F \models q \geq 0\}$.

Generating nonlinear invariants

Given transition formula $F(X, X')$ of a loop body, if we can do consequence findings of certain kinds, then we can implement an intuitive recipe:

Nonlinear invariants through consequence finding

1. Compute all consequences of F denoted by $\mathbf{C}(F) \triangleq \{q : F \models q \geq 0\}$.
2. Extract all *invariant linear terms* c such that $c' - c \in \mathbf{C}(F)$.

Generating nonlinear invariants

Given transition formula $F(X, X')$ of a loop body, if we can do consequence findings of certain kinds, then we can implement an intuitive recipe:

Nonlinear invariants through consequence finding

1. Compute all consequences of F denoted by $\mathbf{C}(F) \triangleq \{q : F \models q \geq 0\}$.
2. Extract all *invariant linear terms* c such that $c' - c \in \mathbf{C}(F)$.
3. Extract all recurrences of the form $t_i x' - t_i x - p_i \in \mathbf{C}(F)$, where p_i's are invariant polynomials "generated" by invariant linear term c's.
4. Abstract the dynamics of the loop with nonlinear invariant

$$F^{\star} \triangleq \exists k.\mathit{Int}(k) \wedge k \geq 0 \wedge \bigwedge_i t_i x' \leq t_i x + k p_i \ .$$

Synthesizing nonlinear ranking functions

Given transition formula $F(X, X')$ of a loop body, we could also use consequence finding to prove termination.

mp_{PRF} through consequence finding

Synthesizing nonlinear ranking functions

Given transition formula $F(X, X')$ of a loop body, we could also use consequence finding to prove termination.

mp$_{\text{PRF}}$ through consequence finding

1. Compute the complete set of polynomials that are bounded from below by F:

$$\textit{Bounded}(F) \triangleq \{q : F \models q \geq 0\} \ .$$

Synthesizing nonlinear ranking functions

Given transition formula $F(X, X')$ of a loop body, we could also use consequence finding to prove termination.

mp$_{\text{PRF}}$ through consequence finding

1. Compute the complete set of polynomials that are bounded from below by F:
$$\textit{Bounded}(F) \triangleq \{q : F \models q \geq 0\} .$$
2. Given the previous set, find the subset of polynomials that are also entailed to be decreasing by F:
$$\textit{PRF}(F) \triangleq \{q : q \in \textit{Bounded}(F) \wedge F \models q' \leq q - 1\} .$$

Synthesizing nonlinear ranking functions

Given transition formula $F(X, X')$ of a loop body, we could also use consequence finding to prove termination.

mp$_{\text{PRF}}$ through consequence finding

1. Compute the complete set of polynomials that are bounded from below by F:

$$\textit{Bounded}(F) \triangleq \{q : F \models q \geq 0\} \ .$$

2. Given the previous set, find the subset of polynomials that are also entailed to be decreasing by F:

$$\textit{PRF}(F) \triangleq \{q : q \in \textit{Bounded}(F) \wedge F \models q' \leq q - 1\} \ .$$

3. Return true if *PRF*(F) is not empty since F has a polynomial ranking function.

The theory of linear integer/real rings **LIRR**

Consequences modulo the standard theory are hard to represent or manipulate. We thus develop **LIRR**, a weak theory of nonlinear integer arithmetic:

- Consequences modulo **LIRR** can be finitely represented.

The theory of linear integer/real rings **LIRR**

Consequences modulo the standard theory are hard to represent or manipulate. We thus develop **LIRR**, a weak theory of nonlinear integer arithmetic:

- Consequences modulo **LIRR** can be finitely represented.
- **LIRR** admits a complete consequence finding procedure.

The theory of linear integer/real rings **LIRR**

Consequences modulo the standard theory are hard to represent or manipulate. We thus develop **LIRR**, a weak theory of nonlinear integer arithmetic:

- Consequences modulo **LIRR** can be finitely represented.
- **LIRR** admits a complete consequence finding procedure.

Specifically, **LIRR** achieves this by

The theory of linear integer/real rings **LIRR**

Consequences modulo the standard theory are hard to represent or manipulate. We thus develop **LIRR**, a weak theory of nonlinear integer arithmetic:

- Consequences modulo **LIRR** can be finitely represented.
- **LIRR** admits a complete consequence finding procedure.

Specifically, **LIRR** achieves this by

- only defines a weakly axiomatized multiplication symbol that generates fewer consequences than the standard model, e.g., it cannot derive $0 \leq xy$ from $0 \leq x \wedge 0 \leq y$.

The theory of linear integer/real rings **LIRR**

Consequences modulo the standard theory are hard to represent or manipulate. We thus develop **LIRR**, a weak theory of nonlinear integer arithmetic:

- Consequences modulo **LIRR** can be finitely represented.
- **LIRR** admits a complete consequence finding procedure.

Specifically, **LIRR** achieves this by

- only defines a weakly axiomatized multiplication symbol that generates fewer consequences than the standard model, e.g., it cannot derive $0 \leq xy$ from $0 \leq x \wedge 0 \leq y$.
- admitting methods for manipulating polynomial ideals (for equations) and polyhedral cones (for inequalities).

Monotonicity

Monotonicity of the $\star$ operator

If F, G are transition formulas and $F \models_{\textsf{\textbf{LIRR}}} G$, then $F^\star \models_{\textsf{\textbf{LIRR}}} G^\star$.

Monotonicity

Monotonicity of the $\star$ operator

If F, G are transition formulas and $F \models_{\textbf{LIRR}} G$, then $F^\star \models_{\textbf{LIRR}} G^\star$.

Proof intuition: We have computed all consequences of the transition formulas with certain forms. Thus invariants constructed based on these consequences are monotone.

Monotonicity

Monotonicity of the $\star$ operator

If F, G are transition formulas and $F \models_{\textbf{LIRR}} G$, then $F^\star \models_{\textbf{LIRR}} G^\star$.

Proof intuition: We have computed all consequences of the transition formulas with certain forms. Thus invariants constructed based on these consequences are monotone.

Monotonicity of the *mp*$_{\mathsf{PRF}}$ operator

If F, G are transition formulas, $F \models_{\textbf{LIRR}} G$, and *mp*$_{\mathsf{PRF}}(G)$ is true. Then *mp*$_{\mathsf{PRF}}(F)$ is true.

Monotonicity

Monotonicity of the $\star$ operator

If F, G are transition formulas and $F \models_{\textbf{LIRR}} G$, then $F^\star \models_{\textbf{LIRR}} G^\star$.

Proof intuition: We have computed all consequences of the transition formulas with certain forms. Thus invariants constructed based on these consequences are monotone.

Monotonicity of the *mp*$_{\text{PRF}}$ operator

If F, G are transition formulas, $F \models_{\textbf{LIRR}} G$, and $mp_{\text{PRF}}(G)$ is true. Then $mp_{\text{PRF}}(F)$ is true.

Proof intuition: We have computed all polynomial terms that are implied by F to be bounded from below and decreasing in **LIRR**. Thus the RF synthesis procedure is complete and monotone (in **LIRR**).

Summary

Nonlinear reasoning through **LIRR**

- We can compute strongest consequences of certain forms modulo **LIRR**.
- The complete consequence finding makes it possible to have monotone decision procedures for both safety and liveness properties that require nonlinear reasoning.

Outline

Introduction

Background and Preliminaries

A Framework for Compositional and Monotone Termination Analysis

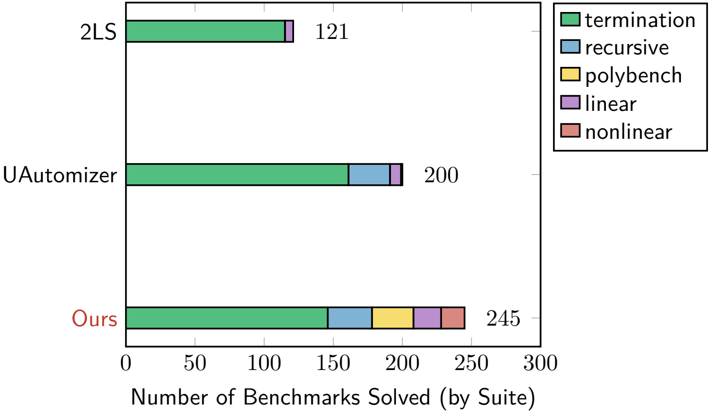
A Weak Theory of Nonlinear Arithmetics with Applications

Experimental Evaluation

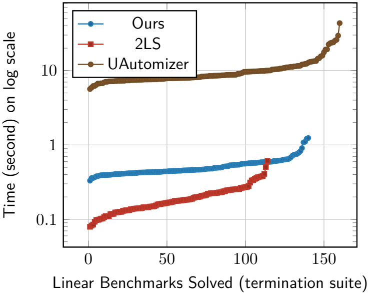
Takeaways

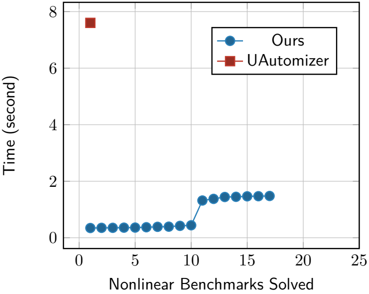
Comparing termination analyses: #benchmarks solved

- Suites termination, recursive: linear programs from SV-COMP/Termination
- Suite polybench: real-world numerical C programs with 10K LOC
- Suite linear: integer linear loops with mod and div
- Suite nonlinear: nonlinear programs from SV-COMP/Termination



Comparing termination analyses: running time





Linear and nonlinear invariant generation

LIRR is weak by design. So are our invariants useful?

Linear and nonlinear invariant generation

LIRR is weak by design. So are our invariants useful?

- Benchmarks from SV-COMP's c/ReachSafety-Loops

Linear and nonlinear invariant generation

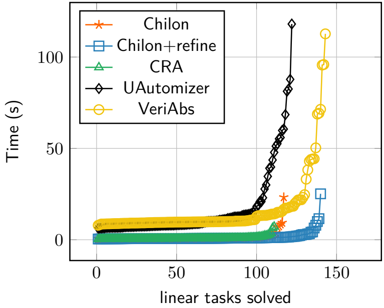
LIRR is weak by design. So are our invariants useful?

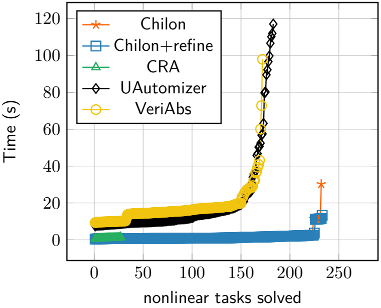
- Benchmarks from SV-COMP's c/ReachSafety-Loops
- Chilon: our invariants
- Chilon+refine: our invariants + control-flow refinement [CBKR19]
- CRA: monotone invariants by solving linear recurrences [FK15]
- Ultimate Automizer, VeriAbs: top performers in SV-COMP

Linear and nonlinear invariant generation

LIRR is weak by design. So are our invariants useful?

- Benchmarks from SV-COMP's c/ReachSafety-Loops
- Chilon: our invariants
- Chilon+refine: our invariants + control-flow refinement [CBKR19]
- CRA: monotone invariants by solving linear recurrences [FK15]
- Ultimate Automizer, VeriAbs: top performers in SV-COMP





Outline

Introduction

Background and Preliminaries

A Framework for Compositional and Monotone Termination Analysis

A Weak Theory of Nonlinear Arithmetics with Applications

Experimental Evaluation

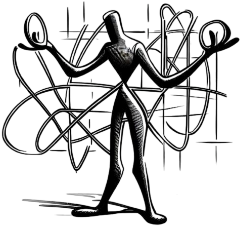
Takeaways

Conclusion

My research shows it is possible to design monotone program analyses that are competitive with state-of-the-art tools in terms of capability and running speed. In particular, I have presented

- A framework for monotone termination analysis [ZK21b, ZK21a].
- A weak theory of nonlinear arithmetic that enables monotone invariant generation [KKZ23] and ranking function synthesis [ZK24].

Questions?



References I

Mark Braverman.
Termination of integer linear programs.
In Thomas Ball and Robert B. Jones, editors, *CAV*, pages 372–385, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

John Cyphert, Jason Breck, Zachary Kincaid, and Thomas Reps.
Refinement of path expressions for static analysis.
Proc. ACM Program. Lang., 3(POPL), January 2019.

A. Farzan and Z. Kincaid.
Compositional recurrence analysis.
In *FMCAD*, pages 57–64. IEEE, 2015.

Laure Gonnord, David Monniaux, and Gabriel Radanne.
Synthesis of ranking functions using extremal counterexamples.
SIGPLAN Not., 50(6):608–618, June 2015.

References II

Mehran Hosseini, Joël Ouaknine, and James Worrell.
Termination of linear loops over the integers.
In Christel Baier, Ioannis Chatzigiannakis, Paola Flocchini, and Stefano Leonardi, editors, *ICALP*, volume 132 of *LIPIcs*, pages 118:1–118:13. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019.

Zachary Kincaid, Nicolas Koh, and Shaowei Zhu.
When less is more: Consequence-finding in a weak theory of arithmetic.
Proc. ACM Program. Lang., 7(POPL), jan 2023.

R. E. Tarjan.
Fast algorithms for solving path problems.
J. ACM, 28(3):594–614, July 1981.

References III

R. E. Tarjan.
A unified approach to path problems.
J. ACM, 28(3):577–593, July 1981.

Ashish Tiwari.
Termination of linear programs.
In Rajeev Alur and Doron A. Peled, editors, *CAV*, pages 70–82, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.

Shaowei Zhu and Zachary Kincaid.
Reflections on termination of linear loops.
In Alexandra Silva and K. Rustan M. Leino, editors, *Computer Aided Verification*, pages 51–74, Cham, 2021. Springer International Publishing.

References IV

Shaowei Zhu and Zachary Kincaid.
Termination analysis without the tears.
In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, PLDI 2021, page 1296–1311, New York, NY, USA, 2021. Association for Computing Machinery.