Counterexample-guided Approach to Finding Numerical Invariants

ThanhVu (Vu) Nguyen*,
Timos Antonopoulos†, Andrew Ruef‡, Michael Hicks‡

*University of Nebraska, †Yale University, ‡University of Maryland

FSE 2017
Introduction

Invariants are asserted properties, such as relations among variables that always hold at certain locations in a program

- Assertions
- Pre/Post conditions
- Loop invariants
Invariants are asserted properties, such as relations among variables that always hold at certain locations in a program

- Assertions
- Pre/Post conditions
- Loop invariants

Techniques for automatic invariant generation

- Static: examine program code, compute sound results, but can be expensive and limited to simple invariants
- Dynamic: analyze exec traces, produce expressive invariants, but unsound
Introduction

Invariants are asserted properties, such as relations among variables that always hold at certain locations in a program

- Assertions
- Pre/Post conditions
- Loop invariants

Techniques for automatic invariant generation

- Static: examine program code, compute sound results, but can be expensive and limited to simple invariants
- Dynamic: analyze exec traces, produce expressive invariants, but unsound

Numerical invariants, e.g., relations among numerical variables

- E.g., $x = 2y + 3, 0 \leq idx \leq |arr| - 1, x \leq y^2, x = qy + r$
- Nonlinear polynomial invariants: $x \leq y^2, x = qy + r, \ldots$
Invariants can help understanding programs

```c
int cohendiv(int x, int y){
    assert(x>0 && y>0);
    int q=0; int r=x;
    while(r \geq y){
        int a=1;
        int b=y;
        while[L1](r \geq 2*b){
            a = 2*a;
            b = 2*b;
        }
        r=r-b;
        q=q+a;
    }
    [L2]
    return q;
}
```

What does this program do? What properties hold at L1 and L2?

<table>
<thead>
<tr>
<th>Loop Invariants at L1</th>
</tr>
</thead>
<tbody>
<tr>
<td>x = qy + r</td>
</tr>
<tr>
<td>b = ya</td>
</tr>
<tr>
<td>y \leq b</td>
</tr>
<tr>
<td>r \leq x</td>
</tr>
<tr>
<td>a \leq b</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Postconditions at L2</th>
</tr>
</thead>
<tbody>
<tr>
<td>x = qy + r</td>
</tr>
<tr>
<td>1 \leq q+ r</td>
</tr>
<tr>
<td>r \leq y - 1</td>
</tr>
<tr>
<td>0 \leq r</td>
</tr>
<tr>
<td>r \leq x</td>
</tr>
</tbody>
</table>
Invariants can help understanding programs

```c
int cohendiv(int x, int y){
    assert(x>0 && y>0);
    int q=0; int r=x;
    while(r ≥ y){
        int a=1;
        int b=y;
        while[L1](r ≥ 2*b){
            a = 2*a;
            b = 2*b;
        }
        r=r-b;
        q=q+a;
    }
    [L2]
    return q;
}
```

What does this program do? What properties hold at L1 and L2?

- **Loop invariants at L1:**
  \[
  \begin{align*}
  x &= qy + r \\
  b &= ya \\
  y &≤ b \\
  b &≤ r \\
  r &≤ x \\
  a &≤ b \\
  2 &≤ a + y
  \end{align*}
  \]

- **Postconditions at L2:**
  \[
  \begin{align*}
  x &= qy + r \\
  1 &≤ q + r \\
  r &≤ y − 1 \\
  0 &≤ r \\
  r &≤ x
  \end{align*}
  \]

Describe the semantic of the program (e.g., \(x = qy + r\) for integer division) and reveal useful information (e.g., remainder \(r\) is non-negative).
Invariants can help understanding programs

```c
int cohendiv(int x, int y){
    assert(x>0 && y>0);
    int q=0; int r=x;
    while(r ≥ y){
        int a=1;
        int b=y;
        while[L1](r ≥ 2*b){
            a = 2*a;
            b = 2*b;
        }
        r=r-b;
        q=q+a;
    }
    [L2]
    return q;
}
```

What does this program do? What properties hold at L1 and L2?

- **loop invariants at L1:**
  - \( x = qy + r \)
  - \( b = ya \)
  - \( y ≤ b \)
  - \( b ≤ r \)
  - \( r ≤ x \)
  - \( a ≤ b \)
  - \( 2 ≤ a + y \)

- **postconditions at L2:**
  - \( x = qy + r \)
  - \( 1 ≤ q + r \)
  - \( r ≤ y - 1 \)
  - \( 0 ≤ r \)
  - \( r ≤ x \)

Describe the semantic the program (e.g., \( x = qy + r \) for integer division) and reveal useful information (e.g., remainder \( r \) is non-negative)
void triple(int M, int N, int P){
    assert (0 <= M);
    assert (0 <= N);
    assert (0 <= P);
    int i = 0, j = 0, k = 0;
    int t = 0;
    while(i < N){
        j = 0; t++;  
        while(j < M){
            j++; k = i; t++;
            while (k < P){
                k++; t++;
            }
            i = k;
        }
        i++;
    }
}
Invariants can help analyze program complexities

```c
void triple(int M, int N, int P){
    assert (0 <= M);
    assert (0 <= N);
    assert (0 <= P);
    int i = 0, j = 0, k = 0;
    int t = 0;
    while(i < N){
        j = 0; t++;
        while(j < M){
            j++; k = i; t++;
            while (k < P){
                k++; t++;
            }
            i = k;
        }
        i++;
    }
    [L]
}
```

Complexity of this program?

- Use $t$ to count loop iterations
- At first glance: $t = O(MNP)$
- A more precise complexity bound: $t = O(N + NM + P)$
- Both are nonlinear invariants
Invariants can help verify programs

```c
void f(int u1, int u2) {
    assert(u1 > 0 && u2 > 0);
    int a = 1, b = 1, c = 2, d = 2;
    int x = 3, y = 3;
    int i1 = 0, i2 = 0;
    while (i1 < u1) {
        i1++;
        x = a + c; y = b + d;
        if ((x + y) % 2 == 0) {
            a++; d++;
        } else { a--;}
    i2 = 0;
    while (i2 < u2 ) {
        i2++; c--; b--;
    }
    }
    assert(a + c == b + d);
}

void g(int n, int u1) {
    assert(u1 > 0);
    int x = 0;
    int m = 0;
    while (x < n) {
        if (u1) {
            m = x;
        }
        x = x + 1;
    }
    if (n > 0){
        assert(0 <= m && m < n);
    }
}
```
Invariants can help verify programs

```c
void f(int u1, int u2) {
    assert(u1 > 0 && u2 > 0);
    int a = 1, b = 1, c = 2, d = 2;
    int x = 3, y = 3;
    int i1 = 0, i2 = 0;
    while (i1 < u1) {
        i1++;
        x = a + c; y = b + d;
        if ((x + y) % 2 == 0) {
            a++; d++;
        } else { a--;}
    } else { a--;}
    i2 = 0;
    while (i2 < u2 ) {
        i2++; c--; b--;
    }
    [L]
    assert(a + c == b + d);
}
```

```c
void g(int n, int u1) {
    assert(u1 > 0);
    int x = 0;
    int m = 0;
    while (x < n) {
        if (u1) {
            m = x;
        }
        x = x + 1;
    } [L]
    if (n > 0){
        assert(0 <= m && m < n);
    }
}
```

Assertions hold if matched or implied by discovered invariants at L
Focus on polynomial invariants over numerical variables

Use CounterExample-Guided Invariant Generation (CEGIR) approach

- Dynamic Inference: use DIG’s algorithms to infer nonlinear equalities and linear inequalities from traces
- Static Checking: use KLEE to check candidate invariants and generate counterexample inputs
Example: Dynamic Inference using DIG

```c
int cohendiv(int x, int y){
    assert(x>0 ; y>0);
    int q=0; int r=x;
    while(r >= y){
        int a=1; int b=y;
        while(L1)(r >= 2*b){
            a = 2*a; b = 2*b;
        }
        r=r-b; q=q+a;
    }
    return q;
}
```

Traces:

- x y a b q r
- 15 2 1 2 0 15
- 15 2 2 4 0 15
- 15 2 1 2 4 7
...
- 4 1 1 1 0 4
- 4 1 2 2 0 4
...

Loop invariants at L1:

- Equations:
  - \( x = qy + r \)
  - \( b = ya \)
- Inequalities:
  - \( 2 \leq a + y \)
  - \( y \leq b \)
  - \( b \leq r \)
  - \( r \leq x \)
int cohendiv(int x, int y){
    assert(x>0 ; y>0);
    int q=0; int r=x;
    while(r >= y){
        int a=1; int b=y;
        while[L1](r >= 2*b){
            a = 2*a; b = 2*b;
        }
        r=r-b; q=q+a;
    }
    return q;
}

**Traces:**

<table>
<thead>
<tr>
<th>x</th>
<th>y</th>
<th>a</th>
<th>b</th>
<th>q</th>
<th>r</th>
</tr>
</thead>
<tbody>
<tr>
<td>15</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>0</td>
<td>15</td>
</tr>
<tr>
<td>15</td>
<td>2</td>
<td>2</td>
<td>4</td>
<td>0</td>
<td>15</td>
</tr>
<tr>
<td>15</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>4</td>
<td>7</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Example: Dynamic Inference using DIG

```c
int cohendiv(int x, int y){
    assert(x>0 ; y>0);
    int q=0; int r=x;
    while(r >= y){
        int a=1; int b=y;
        while[L1](r >= 2*b){
            a = 2*a; b = 2*b;
        }
        r=r-b; q=q+a;
    }
    return q;
}
```

Traces:

<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>15</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>0</td>
<td>15</td>
</tr>
<tr>
<td>15</td>
<td>2</td>
<td>2</td>
<td>4</td>
<td>0</td>
<td>15</td>
</tr>
<tr>
<td>15</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>4</td>
<td>7</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Loop invariants at L1:

- **equations**: \( x = qy + r \) \( b = ya \)
- **inequalities**: \( 2 \leq a + y \) \( a \leq b \) \( y \leq b \) \( b \leq r \) \( r \leq x \)
Infer Nonlinear Equations using Equation Solver

Terms and degrees

\[ V = \{ r, y, a \}; \quad \text{deg} = 2 \]

\[ T = \{ 1, r, y, a, ry, ra, ya, r^2, y^2, a^2 \} \]

Nonlinear equation template

\[ c_1 + c_2 r + c_3 y + c_4 a + c_5 ry + c_6 ra + c_7 ya + c_8 r^2 + c_9 y^2 + c_10 a^2 = 0 \]

System of linear equations

trace 1 → \{ r = 15, y = 2, a = 1 \}

eq \{ c_1 + 15 c_2 + 2 c_3 + c_4 + 30 c_5 + 15 c_6 + 2 c_7 + 225 c_8 + 4 c_9 + c_10 = 0 \}

... Solve for coefficients

\[ V = \{ x, y, a, b, q, r \}; \quad \text{deg} = 2 \]

\[ x = qy + r, \quad b = ya \]

<table>
<thead>
<tr>
<th>x</th>
<th>y</th>
<th>a</th>
<th>b</th>
<th>q</th>
<th>r</th>
</tr>
</thead>
<tbody>
<tr>
<td>15</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>0</td>
<td>15</td>
</tr>
<tr>
<td>15</td>
<td>2</td>
<td>2</td>
<td>4</td>
<td>0</td>
<td>15</td>
</tr>
<tr>
<td>15</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>4</td>
<td>7</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>0</td>
<td>4</td>
</tr>
</tbody>
</table>
Infer Nonlinear Equations using Equation Solver

**Terms and degrees**

\[ V = \{r, y, a\}; \quad \text{deg} = 2 \]

\[ T = \{1, r, y, a, ry, ra, ya, r^2, y^2, a^2\} \]

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>y</td>
<td>a</td>
<td>b</td>
<td>q</td>
</tr>
<tr>
<td>15</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>15</td>
<td>2</td>
<td>2</td>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>15</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>0</td>
</tr>
</tbody>
</table>
Infer Nonlinear Equations using Equation Solver

- Terms and degrees
  
  \[V = \{r, y, a\}; \text{ deg } = 2\]
  
  \[\downarrow\]
  
  \[T = \{1, r, y, a, ry, ra, ya, r^2, y^2, a^2\}\]

- Nonlinear equation template

  \[c_1 + c_2 r + c_3 y + c_4 a + c_5 ry + c_6 ra + c_7 ya + c_8 r^2 + c_9 y^2 + c_{10} a^2 = 0\]
Infer Nonlinear Equations using Equation Solver

- **Terms and degrees**
  \[ V = \{r, y, a\}; \text{ deg } = 2 \]
  \[ T = \{1, r, y, a, ry, ra, ya, r^2, y^2, a^2\} \]

- **Nonlinear equation template**
  \[ c_1 + c_2 r + c_3 y + c_4 a + c_5 ry + c_6 ra + c_7 ya + c_8 r^2 + c_9 y^2 + c_{10} a^2 = 0 \]

- **System of linear equations**
  \[ \begin{array}{c}
    \text{trace 1} \rightarrow \{r = 15, y = 2, a = 1\} \\
    \text{eq 1} \rightarrow c_1 + 15c_2 + 2c_3 + c_4 + 30c_5 + 15c_6 + 2c_7 + 225c_8 + 4c_9 + c_{10} = 0 \\
    \vdots
  \end{array} \]
Infer Nonlinear Equations using Equation Solver

- **Terms and degrees**
  \[ V = \{r, y, a\}; \text{ deg } = 2 \]
  \[ \downarrow \]
  \[ T = \{1, r, y, a, ry, ra, ya, r^2, y^2, a^2\} \]

- **Nonlinear equation template**
  \[ c_1 + c_2 r + c_3 y + c_4 a + c_5 ry + c_6 ra + c_7 ya + c_8 r^2 + c_9 y^2 + c_{10} a^2 = 0 \]

- **System of linear equations**
  
  \[
  \begin{array}{c|ccccc}
  x & y & a & b & q & r \\
  \hline
  15 & 2 & 1 & 2 & 0 & 15 \\
  15 & 2 & 2 & 4 & 0 & 15 \\
  15 & 2 & 1 & 2 & 4 & 7 \\
  4 & 1 & 1 & 1 & 0 & 4 \\
  4 & 1 & 2 & 2 & 0 & 4 \\
  \end{array}
  \]

  \[ x = qy + r, \quad b = ya \]
Static Checking

- **Goal**: prove/refute candidate invariants using program code
- **Approach**: reduce invariant checking to reachability
Static Checking

- **Goal**: prove/refute candidate invariants using program code
- **Approach**: reduce invariant checking to reachability
  - Transform program and invariant into another program consisting of a special location $L'$

```
... if (!(x==qy+r)){
  [L'] //x=qy+r is invalid
  abort();
}
//x=qy+r is valid
...
```

Use the symbolic execution tool KLEE to check reachability

Unsound: KLEE can timeout, but in practice is very effective in refuting bad invariants and finding counterexamples.

Can use other test-input generation tools or verifiers instead of KLEE.
Static Checking

- **Goal**: prove/refute candidate invariants using program code
- **Approach**: reduce invariant checking to reachability
  - Transform program and invariant into another program consist of a special location $L'$

```
... if (!(x==qy+r)){
    [L'] //x=qy+r is invalid
    abort();
}
//x=qy+r is valid
...
```

- $L'$ reachable $\implies$ inv is spurious (inputs reaching $L'$ represent cex's)
- $L'$ not reachable (within a time bound) $\implies$ NumInv accepts the invariant

Use the symbolic execution tool KLEE to check reachability

Unsound: KLEE can timeout, but in practice is very effective in refuting bad invariants and finding cex's

Can use other test-input generation tools or verifiers instead of KLEE
Static Checking

- **Goal**: prove/refute candidate invariants using program code
- **Approach**: reduce invariant checking to reachability
  
  - Transform program and invariant into another program consist of a special location \( L' \)

  ```
  ...  
  if (!(x==qy+r)){
    [L'] //x=qy+r is invalid  
    abort();  
  }
  //x=qy+r is valid  
  ...
  ```

  - \( L' \) reachable \( \implies \) inv is spurious (inputs reaching \( L' \) represent cex’s)
  - \( L' \) not reachable (within a time bound) \( \implies \) NumInv accepts the invariant

- Use the symbolic execution tool **KLEE** to check reachability
  
  - *Unsound*: KLEE can timeout, but in practice is *very effective* in refuting bad invariants and finding cex’s
  - Can use other test-input generation tools or verifiers instead of KLEE
Evaluation

Setup

- NumInv is implemented in SAGE/Python (with Z3 backend solver)
- Test machine: 10-core 2.4GHz CPU, 128GB Ram, Linux OS

Benchmark

- Program Understanding: NLA testsuite, 27 programs with nonlinear invariants
- Complexity Analysis: 19 programs collected from static complexity analysis work
- Program Verification: HOLA benchmark, 46 programs with assertions, compare against PIE
Example: Program Understanding

```c
int cohendiv(int x, int y){
    assert(x>0 && y>0);
    int q=0; int r=x;
    while(r ≥ y){
        int a=1;
        int b=y;
        while[L1](r ≥ 2*b){
            a = 2*a;
            b = 2*b;
        }
        r=r-b;
        q=q+a;
    }
    [L2]
    return q;
}
```

What does this program do? What properties hold at L1 and L2?

loop invariants at L1:
- \( x = qy + r \)
- \( b = ya \)
- \( y ≤ b \)
- \( r ≤ x \)
- \( a ≤ b \)
- \( 2 ≤ a + y \)

postconditions at L2:
- \( x = qy + r \)
- \( 1 ≤ q + r \)
- \( r ≤ y - 1 \)
- \( 0 ≤ r \)
- \( r ≤ x \)

Indicate the exact semantic of integer division and reveal other useful correctness information (e.g., remainder is non-negative)
Example: Program Understanding

```c
int cohendiv(int x, int y){
    assert(x>0 && y>0);
    int q=0; int r=x;
    while(r ≥ y){
        int a=1;
        int b=y;
        while[L1](r ≥ 2*b){
            a = 2*a;
            b = 2*b;
        }
        r=r-b;
        q=q+a;
    }
    return q;
}
```

What does this program do? What properties hold at L1 and L2?

- **Loop invariants at L1:**
  
  \[
  \begin{align*}
  &x = qy + r & b = ya \\
  &y \leq b & b \leq r \\
  &r \leq x & a \leq b \\
  &2 \leq a + y
  \end{align*}
  \]

- **Postconditions at L2:**

  \[
  \begin{align*}
  &x = qy + r \\
  &1 \leq q + r & r \leq y - 1 \\
  &0 \leq r & r \leq x
  \end{align*}
  \]

Indicate the exact semantic of integer division and reveal other useful correctness information (e.g., remainder is non-negative).
What does this program do? What properties hold at \textbf{L1} and \textbf{L2}?

- **Loop invariants at \textbf{L1}:**
  \begin{align*}
  x &= qy + r \\
  b &= ya \\
  y &\leq b \\
  b &\leq r \\
  r &\leq x \\
  a &\leq b \\
  2 &\leq a + y
  \end{align*}

- **Postconditions at \textbf{L2}:**
  \begin{align*}
  x &= qy + r \\
  1 &\leq q + r \\
  r &\leq y - 1 \\
  0 &\leq r \\
  r &\leq x
  \end{align*}

Indicate the exact semantic of integer division and reveal other useful correctness information (e.g., remainder is non-negative)
## Results: Program Understanding

<table>
<thead>
<tr>
<th>Prog</th>
<th>Locs</th>
<th>Invs</th>
<th>Time (s)</th>
<th>Correct</th>
</tr>
</thead>
<tbody>
<tr>
<td>cohen</td>
<td>2</td>
<td>11</td>
<td>24.5</td>
<td>✓</td>
</tr>
<tr>
<td>divbin</td>
<td>2</td>
<td>12</td>
<td>116.8</td>
<td>✓</td>
</tr>
<tr>
<td>manna</td>
<td>1</td>
<td>5</td>
<td>30.8</td>
<td>✓</td>
</tr>
<tr>
<td>hard</td>
<td>2</td>
<td>13</td>
<td>71.4</td>
<td>✓</td>
</tr>
<tr>
<td>sqrt1</td>
<td>1</td>
<td>5</td>
<td>19.3</td>
<td>✓</td>
</tr>
<tr>
<td>dijkstra</td>
<td>2</td>
<td>14</td>
<td>89.3</td>
<td>✓</td>
</tr>
<tr>
<td>freire1</td>
<td>1</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>freire2</td>
<td>1</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>cohenuc</td>
<td>1</td>
<td>5</td>
<td>22.5</td>
<td>✓</td>
</tr>
<tr>
<td>egcd1</td>
<td>1</td>
<td>9</td>
<td>284.5</td>
<td>✓</td>
</tr>
<tr>
<td>egcd2</td>
<td>2</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>egcd3</td>
<td>3</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>prodbin</td>
<td>1</td>
<td>7</td>
<td>45.1</td>
<td>✓</td>
</tr>
<tr>
<td>prod4br</td>
<td>1</td>
<td>11</td>
<td>87.3</td>
<td>✓</td>
</tr>
<tr>
<td>knuth</td>
<td>1</td>
<td>9</td>
<td>84.6</td>
<td>✓</td>
</tr>
<tr>
<td>fermat1</td>
<td>3</td>
<td>26</td>
<td>185.3</td>
<td>✓</td>
</tr>
<tr>
<td>fermat2</td>
<td>1</td>
<td>8</td>
<td>101.8</td>
<td>✓</td>
</tr>
<tr>
<td>lcm1</td>
<td>3</td>
<td>22</td>
<td>175.2</td>
<td>✓</td>
</tr>
<tr>
<td>lcm2</td>
<td>1</td>
<td>7</td>
<td>163.8</td>
<td>✓</td>
</tr>
<tr>
<td>geo1</td>
<td>1</td>
<td>7</td>
<td>24.4</td>
<td>✓</td>
</tr>
<tr>
<td>geo2</td>
<td>1</td>
<td>9</td>
<td>24.3</td>
<td>✓</td>
</tr>
<tr>
<td>geo3</td>
<td>1</td>
<td>7</td>
<td>32.3</td>
<td>✓</td>
</tr>
<tr>
<td>ps2</td>
<td>1</td>
<td>3</td>
<td>17.0</td>
<td>✓</td>
</tr>
<tr>
<td>ps3</td>
<td>1</td>
<td>4</td>
<td>17.8</td>
<td>✓</td>
</tr>
<tr>
<td>ps4</td>
<td>1</td>
<td>4</td>
<td>18.5</td>
<td>✓</td>
</tr>
<tr>
<td>ps5</td>
<td>1</td>
<td>4</td>
<td>19.3</td>
<td>✓</td>
</tr>
<tr>
<td>ps6</td>
<td>1</td>
<td>3</td>
<td>21.0</td>
<td>✓</td>
</tr>
</tbody>
</table>

### Experiment

- **NLA suite**: 27 programs
- Require nonlinear invariants
- Use documented invariants (loop invariants and postconds) as **ground truths**
- **Goal**: obtain invariants and compare to ground truths
Results: Program Understanding

<table>
<thead>
<tr>
<th>Prog</th>
<th>Locs</th>
<th>Invs</th>
<th>Time (s)</th>
<th>Correct</th>
</tr>
</thead>
<tbody>
<tr>
<td>cohendiv</td>
<td>2</td>
<td>11</td>
<td>24.5</td>
<td>✓</td>
</tr>
<tr>
<td>divbin</td>
<td>2</td>
<td>12</td>
<td>116.8</td>
<td>✓</td>
</tr>
<tr>
<td>manna</td>
<td>1</td>
<td>5</td>
<td>30.8</td>
<td>✓</td>
</tr>
<tr>
<td>hard</td>
<td>2</td>
<td>13</td>
<td>71.4</td>
<td>✓</td>
</tr>
<tr>
<td>sqrt1</td>
<td>1</td>
<td>5</td>
<td>19.3</td>
<td>✓</td>
</tr>
<tr>
<td>dijkstra</td>
<td>2</td>
<td>14</td>
<td>89.3</td>
<td>✓</td>
</tr>
<tr>
<td>freire1</td>
<td>1</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>freire2</td>
<td>1</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>cohencu</td>
<td>1</td>
<td>5</td>
<td>22.5</td>
<td>✓</td>
</tr>
<tr>
<td>egcd1</td>
<td>1</td>
<td>9</td>
<td>284.5</td>
<td>✓</td>
</tr>
<tr>
<td>egcd2</td>
<td>2</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>egcd3</td>
<td>3</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>prodbin</td>
<td>1</td>
<td>7</td>
<td>45.1</td>
<td>✓</td>
</tr>
<tr>
<td>prod4br</td>
<td>1</td>
<td>11</td>
<td>87.3</td>
<td>✓</td>
</tr>
<tr>
<td>knuth</td>
<td>1</td>
<td>9</td>
<td>84.6</td>
<td>✓</td>
</tr>
<tr>
<td>fermat1</td>
<td>3</td>
<td>26</td>
<td>185.3</td>
<td>✓</td>
</tr>
<tr>
<td>fermat2</td>
<td>1</td>
<td>8</td>
<td>101.8</td>
<td>✓</td>
</tr>
<tr>
<td>lcm1</td>
<td>3</td>
<td>22</td>
<td>175.2</td>
<td>✓</td>
</tr>
<tr>
<td>lcm2</td>
<td>1</td>
<td>7</td>
<td>163.8</td>
<td>✓</td>
</tr>
<tr>
<td>geo1</td>
<td>1</td>
<td>7</td>
<td>24.4</td>
<td>✓</td>
</tr>
<tr>
<td>geo2</td>
<td>1</td>
<td>9</td>
<td>24.3</td>
<td>✓</td>
</tr>
<tr>
<td>geo3</td>
<td>1</td>
<td>7</td>
<td>32.3</td>
<td>✓</td>
</tr>
<tr>
<td>ps2</td>
<td>1</td>
<td>3</td>
<td>17.0</td>
<td>✓</td>
</tr>
<tr>
<td>ps3</td>
<td>1</td>
<td>4</td>
<td>17.8</td>
<td>✓</td>
</tr>
<tr>
<td>ps4</td>
<td>1</td>
<td>4</td>
<td>18.5</td>
<td>✓</td>
</tr>
<tr>
<td>ps5</td>
<td>1</td>
<td>4</td>
<td>19.3</td>
<td>✓</td>
</tr>
<tr>
<td>ps6</td>
<td>1</td>
<td>3</td>
<td>21.0</td>
<td>✓</td>
</tr>
</tbody>
</table>

Experiment

- **NLA suite**: 27 programs
- Require nonlinear invariants
- Use documented invariants (loop invariants and postconds) as ground truths
- **Goal**: obtain invariants and compare to ground truths

Results: NumInv found correct invariants in 23/27 progs

- Most results equiv to or stronger than (imply) ground truths
- Several unexpected and undocumented invariants
- Some invariants reveal “how” program works in details
Example: Complexity Analysis

```c
void triple(int M, int N, int P) {
    assert (0 <= M);
    assert (0 <= N);
    assert (0 <= P);
    int i = 0, j = 0, k = 0;
    int t = 0;
    while (i < N) {
        j = 0; t++;
        while (j < M) {
            j++; k = i; t++;
            while (k < P) {
                k++; t++;
            }
            i = k;
        }
        i++;
    }
    [L]
}
```

Complexity of this program?

- Existing result: \( t = O(N + NM + P) \)
void triple(int M, int N, int P){
    assert (0 <= M);
    assert (0 <= N);
    assert (0 <= P);
    int i = 0, j = 0, k = 0;
    int t = 0;
    while(i < N){
        j = 0; t++;
        while(j < M){
            j++; k = i; t++;
            while (k < P){
                k++; t++;
            }
        i = k;
    }
    i++;
}

Complexity of this program?

- Existing result: $t = O(N + NM + P)$
- NumInv found a very unexpected inv:

$$P^2 Mt + PM^2 t - PMN t - M^2 N t - PM t^2 + MN t^2 + PM t - PN t - 2MN t + Pt^2 + Mt^2 + N t^2 - t^3 - N t + t^2 = 0$$

Nonlinear invariants can represent disjunctive properties capturing different complexity bounds.
void triple(int M, int N, int P) {
    assert (0 <= M);
    assert (0 <= N);
    assert (0 <= P);
    int i = 0, j = 0, k = 0;
    int t = 0;
    while (i < N) {
        j = 0; t++;
        while (j < M) {
            j++; k = i; t++;
            while (k < P) {
                k++; t++;
            }
        }
    }
    i = k;
}

Complexity of this program?

- Existing result: $t = O(N + NM + P)$
- NumInv found a very unexpected inv:

  $$\begin{align*}
  P^2Mt + PM^2t - PMNt - M^2Nt \\
  - PMt^2 + MNt^2 + PMt - PNT - 2MNT \\
  + Pt^2 + Mt^2 + Nt^2 - t^3 - Nt + t^2 = 0
  \end{align*}$$

- Solve for $t$ yields the most precise, unpublished bound:

  $$\begin{align*}
  t &= 0 \quad \text{when } N = 0, \\
  t &= P + M + 1 \quad \text{when } N \leq P, \\
  t &= N - M(P - N) \quad \text{when } N > P
  \end{align*}$$

- Nonlinear invariants can represent disjunctive properties capturing different complexity bounds
### Results: Complexity Analysis

<table>
<thead>
<tr>
<th>Prog</th>
<th>Invs</th>
<th>Time (s)</th>
<th>Valid</th>
</tr>
</thead>
<tbody>
<tr>
<td>cav09_fig1a</td>
<td>1</td>
<td>14.3</td>
<td>✓</td>
</tr>
<tr>
<td>cav09_fig1d</td>
<td>1</td>
<td>14.2</td>
<td>✓</td>
</tr>
<tr>
<td>cav09_fig2d</td>
<td>3</td>
<td>36.0</td>
<td>✓</td>
</tr>
<tr>
<td>cav09_fig3a</td>
<td>3</td>
<td>14.2</td>
<td>✓</td>
</tr>
<tr>
<td>cav09_fig5b</td>
<td>5</td>
<td>46.8</td>
<td>✓</td>
</tr>
<tr>
<td>pldi09_ex6</td>
<td>7</td>
<td>54.1</td>
<td>✓</td>
</tr>
<tr>
<td>pldi09_fig2 (triple)</td>
<td>6</td>
<td>93.5</td>
<td>✓✓</td>
</tr>
<tr>
<td>pldi09_fig4_1</td>
<td>3</td>
<td>44.2</td>
<td>✓</td>
</tr>
<tr>
<td>pldi09_fig4_2</td>
<td>5</td>
<td>43.7</td>
<td>✓</td>
</tr>
<tr>
<td>pldi09_fig4_3</td>
<td>3</td>
<td>37.5</td>
<td>✓</td>
</tr>
<tr>
<td>pldi09_fig4_4</td>
<td>4</td>
<td>56.6</td>
<td>-</td>
</tr>
<tr>
<td>pldi09_fig4_5</td>
<td>3</td>
<td>31.6</td>
<td>✓</td>
</tr>
<tr>
<td>popl09_fig2_1</td>
<td>2</td>
<td>211.7</td>
<td>✓✓</td>
</tr>
<tr>
<td>popl09_fig2_2</td>
<td>2</td>
<td>65.1</td>
<td>✓✓</td>
</tr>
<tr>
<td>popl09_fig3_4</td>
<td>4</td>
<td>54.7</td>
<td>✓</td>
</tr>
<tr>
<td>popl09_fig4_1</td>
<td>2</td>
<td>42.7</td>
<td>✓</td>
</tr>
<tr>
<td>popl09_fig4_2</td>
<td>2</td>
<td>158.3</td>
<td>✓✓</td>
</tr>
<tr>
<td>popl09_fig4_3</td>
<td>5</td>
<td>39.2</td>
<td>✓</td>
</tr>
<tr>
<td>popl09_fig4_4</td>
<td>3</td>
<td>34.2</td>
<td>✓</td>
</tr>
</tbody>
</table>

**Experiment**

- 19 progs from static complexity work
- Obtain postconds representing complexity
- **Goal**: compare against results from prev work
Results: Complexity Analysis

<table>
<thead>
<tr>
<th>Prog</th>
<th>Invs</th>
<th>Time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>cav09.fig1a</td>
<td>1</td>
<td>14.3</td>
</tr>
<tr>
<td>cav09.fig1d</td>
<td>1</td>
<td>14.2</td>
</tr>
<tr>
<td>cav09.fig2d</td>
<td>3</td>
<td>36.0</td>
</tr>
<tr>
<td>cav09.fig3a</td>
<td>3</td>
<td>14.2</td>
</tr>
<tr>
<td>cav09.fig5b</td>
<td>5</td>
<td>46.8</td>
</tr>
<tr>
<td>pldi09.ex6</td>
<td>7</td>
<td>54.1</td>
</tr>
<tr>
<td>pldi09.fig2 (triple)</td>
<td>6</td>
<td>93.5</td>
</tr>
<tr>
<td>pldi09.fig4_1</td>
<td>3</td>
<td>44.2</td>
</tr>
<tr>
<td>pldi09.fig4_2</td>
<td>5</td>
<td>43.7</td>
</tr>
<tr>
<td>pldi09.fig4_3</td>
<td>3</td>
<td>37.5</td>
</tr>
<tr>
<td>pldi09.fig4_4</td>
<td>4</td>
<td>56.6</td>
</tr>
<tr>
<td>pldi09.fig4_5</td>
<td>3</td>
<td>31.6</td>
</tr>
<tr>
<td>popl09.fig2_1</td>
<td>2</td>
<td>211.7</td>
</tr>
<tr>
<td>popl09.fig2_2</td>
<td>2</td>
<td>65.1</td>
</tr>
<tr>
<td>popl09.fig3_4</td>
<td>4</td>
<td>54.7</td>
</tr>
<tr>
<td>popl09.fig4_1</td>
<td>2</td>
<td>42.7</td>
</tr>
<tr>
<td>popl09.fig4_2</td>
<td>2</td>
<td>158.3</td>
</tr>
<tr>
<td>popl09.fig4_3</td>
<td>5</td>
<td>39.2</td>
</tr>
<tr>
<td>popl09.fig4_4</td>
<td>3</td>
<td>34.2</td>
</tr>
</tbody>
</table>

Experiment

- 19 progs from static complexity work
- Obtain postconds representing complexity
- **Goal**: compare against results from prev work

Results: Obtain equiv (14) or more precise bounds (4) in 18/19 progs
Example: Verification

```c
void f(int u1, int u2) {
  assert(u1 > 0 && u2 > 0);
  int a = 1, b = 1, c = 2, d = 2;
  int x = 3, y = 3;
  int i1 = 0, i2 = 0;
  while (i1 < u1) {
    i1++;
    x = a + c; y = b + d;
    if ((x + y) % 2 == 0) {
      a++; d++;
    } else { a--;}
    i2 = 0;
    while (i2 < u2 ) {
      i2++; c--; b--; 
    }
  }
  [L] //NumInv found:
  //b + 1 = c, a + 1 = d,
  //a + b <= 2, 2 <= a
  assert(a + c == b + d);
}
```

```c
void g(int n, int u1) {
  assert(u1 > 0);
  int x = 0;
  int m = 0;
  while (x < n) {
    if (u1) {
      m = x;
    }
    x = x + 1;
  }
  [L] //NumInv found:
  //m^2 = nx - m - x, mn = x^2 - x
  //m <= x, x <= m + 1, n <= x
  if (n > 0){
    assert(0 <= m && m < n);
  }
}
```
Results: Verification

Experiment

- HOLA benchmark: 46 programs
- Various assertions (mostly postconds)
- **Goal:**
  - Obtain and compare invariants: if match or imply assertions, then assertions hold
  - Also compare with existing tool PIE

Results: Found equiv (23) or stronger (13) invariants in 36/46 programs

Time: mean 30s, median 13s

Nonlinear invariants can prove many nontrivial and unsupported properties
Results: Verification

Experiment

- HOLA benchmark: 46 programs
- Various assertions (mostly postconds)
- Goal:
  - Obtain and compare invariants: if match or imply assertions, then assertions hold
  - Also compare with existing tool PIE

Results: Found equiv (23) or stronger (13) invariants in 36/46 programs

- Time: mean 30s, median 13s
- Nonlinear invariants can prove many nontrivial and unsupported properties
Conclusion

NumInv

- Use CEGIR for *numerical* invariant generation
  - Dynamic Inference: use DIG to compute nonlinear invariants
  - Static Checking: use KLEE to check candidate invariants and obtain cex’s

- *Unsound*, but experience shows practical and effective in removing invalid results and can handle complex invariants

[https://bitbucket.org/nguyenthanhvuh/dig2/](https://bitbucket.org/nguyenthanhvuh/dig2/)
Conclusion

NumInv

- Use CEGIR for numerical invariant generation
  - Dynamic Inference: use DIG to compute nonlinear invariants
  - Static Checking: use KLEE to check candidate invariants and obtain cex’s
- Unsound, but experience shows practical and effective in removing invalid results and can handle complex invariants

Results

- Discover necessary nonlinear invariants to understand programs
- Find useful invariants capturing nontrivial runtime complexity
- Compete well with existing work
- General polynomial invariants (e.g., nonlinear properties) can surprisingly represent/prove many nontrivial, complex, and unsupported properties

https://bitbucket.org/nguyenthanhvuh/dig2/