Non-Linear Reasoning for Invariant Synthesis

Zachary Kincaid\textsuperscript{1} \hspace{1em} John Cyphert\textsuperscript{2} \hspace{1em} Jason Breck\textsuperscript{2} \hspace{1em} Thomas Reps\textsuperscript{2,3}

\textsuperscript{1}Princeton University \hspace{1em} \textsuperscript{2}University of Wisconsin-Madison \hspace{1em} \textsuperscript{3}GrammaTech, Inc

January 12, 2018 @ 15:50
The problem: generating non-linear numerical loop invariants

- Resource-bound analysis
- Side channel analysis
- Secure information flow
- ...

while \( i < n \):
    \[ x = x + i \]
    \[ i = i + 1 \]

\[
i^{(k)} = i^{(k-1)} + 1
\]
\[
x^{(k)} = x^{(k-1)} + i^{(k-1)}
\]

\[
i' = i + k
\]
\[
x' = x + \frac{k(k-1)}{2} + ki
\]

\[
\exists k. k \geq 0 \land \left( i' = i + k \quad x' = x + \frac{k(k-1)}{2} + ki \right)
\]
while \((i < n)\):
\[
x = x + i \\
i = i + 1
\]

\[
\begin{align*}
i^{(k)} &= i^{(k-1)} + 1 \\
x^{(k)} &= x^{(k-1)} + i^{(k-1)}
\end{align*}
\]

- branching
- nested loops
- non-determinism

\[
\exists k. k \geq 0 \land \left( \begin{array}{c}
i' = i + k \\
x' = x + \frac{k(k-1)}{2} + ki
\end{array} \right)
\]
while \( i < n \):
    \[
    \begin{align*}
    x &= x + i \\
    i &= i + 1
    \end{align*}
    \]

\[
\begin{align*}
i^{(k)} &= i^{(k-1)} + 1 \\
x^{(k)} &= x^{(k-1)} + i^{(k-1)}
\end{align*}
\]

- branching
- nested loops
- non-determinism

algebraic numbers
binary-search(A,target):
lo = 1, hi = size(A), ticks = 0
while (lo <= hi):
ticks++;
    mid = lo + (hi-lo)/2
    if A[mid] == target:
        return mid
    else if A[mid] < target:
        lo = mid+1
    else:
        hi = mid-1

log(A) times

ticks' = ticks + 1
mid' = lo + (hi-lo)/2

log(A) times

hi' = hi
lo' = mid + 1

2^k:
k = 0

ticks' = ticks + k (hi-lo) / 2
binary-search(A,target):
lo = 1, hi = size(A), ticks = 0
while (lo <= hi):
    ticks++;
    mid = lo + (hi-lo)/2
    if A[mid] == target:
        return mid
    else if A[mid] < target:
        lo = mid+1
    else:
        hi = mid-1

\begin{align*}
ticks' &= ticks + 1 \\
\land mid' &= lo + (hi - lo)/2 \\
\land ((A[mid] < target \\
\land lo' &= mid + 1 \\
\land hi' &= hi) \\
\lor (A[mid] > target \\
\land lo' &= lo \\
\land hi' &= mid - 1))
\end{align*}
binary-search(A, target):
  lo = 1, hi = size(A), ticks = 0
  while (lo <= hi):
    ticks++;
    mid = lo + (hi-lo)/2
    if A[mid] == target:
      return mid
    else if A[mid] < target:
      lo = mid+1
    else:
      hi = mid-1
  \[ ticks^{(k+1)} = ticks^{(k)} + 1 \]
  \[ (hi' - lo')^{(k+1)} \leq (hi - lo)^{(k)}/2 - 1 \]
binary-search(A, target):
  lo = 1, hi = size(A), ticks = 0
  while (lo <= hi):
    ticks++;
    mid = lo + (hi-lo)/2
    if A[mid] == target:
      return mid
    else if A[mid] < target:
      lo = mid+1
    else:
      hi = mid-1

\[ ticks^{(k)} = ticks^{(0)} + k \]
\[ (h' - l')^{(k)} \leq \left( \frac{1}{2} \right)^k (h - l + 2)^{(0)} - 2 \]
binary-search(A, target):
lo = 1, hi = size(A), ticks = 0
while (lo <= hi):
    ticks++;
    mid = lo + (hi-lo)/2
    if A[mid] == target:
        return mid
    else if A[mid] < target:
        lo = mid+1
    else:
        hi = mid-1

\[
\exists k. k \geq 0 \\
\text{ticks'} = \text{ticks} + k \\
(hi' - lo') \leq \left(\frac{1}{2}\right)^k (hi - lo + 2) - 2
\]
for (i = 0; i < n; i++):
    for (j = 0; j < i; j++):
        ticks++
for (i = 0; i < n; i++):
    for (j = 0; j < i; j++):
        ticks++
for (i = 0; i < n; i++):
    for (j = 0; j < i; j++):
        ticks++

\[
\begin{align*}
ticks^{(k+1)} &= ticks^{(k)} + 1 \\
^{(k+1)} &= j^{(k)} + 1 \\
^{(k+1)} &= i^{(k)} \\
^{(k+1)} &= n^{(k)}
\end{align*}
\]
\begin{align*}
  ticks^{(k)} &= ticks^{(0)} + k \\
  j^{(k)} &= j^{(0)} + k \\
  i^{(k)} &= i^{(0)} \\
  n^{(k)} &= n^{(0)}
\end{align*}

\begin{align*}
  \text{for } (i = 0; i < n; i++): \\
  \text{for } (j = 0; j < i; j++): \\
  \text{ticks}++ \\
\end{align*}
\begin{align*}
\textbf{for} \quad (i = 0; i < n; i++) & : \\
\quad \textbf{for} \quad (j = 0; j < i; j++) & : \\
\quad \quad \text{ticks}++ & \\
\quad \} & \left. \begin{array}{l}
i' = i \\
\land n' = n \\
\land j' \leq i \land (\exists k. \quad k \geq 0 \\
\quad \land \quad \text{ticks}' = \text{ticks} + k \\
\quad \land \quad j' = j + k \end{array} \right) 
\end{align*}
for (i = 0; i < n; i++):
    for (j = 0; j < i; j++):
        ticks++
    
\[
\begin{align*}
\text{for } (i = 0; i < n; i++) & : \\
\text{for } (j = 0; j < i; j++) : \\
ticks & ++
\end{align*}
\]

\[
\begin{align*}
& i < n \\
\land & i' = i + 1 \\
\land & n' = n \\
\land & j' = i \\
\land & \left( \exists k. \quad k \geq 0 \\
\land & ticks' = ticks + k \\
\land & j' = k \right)
\end{align*}
\]
\[
\text{for } (i = 0; i < n; i++):
\begin{align*}
&\text{for } (j = 0; j < i; j++):\ \\
&\quad \text{ticks++}
\end{align*}
\]

\[
\begin{align*}
ticks^{(k+1)} &= ticks^{(k)} + i^{(k)} \\
i^{(k+1)} &= i^{(k)} + 1 \\
n^{(k+1)} &= n^{(k)}
\end{align*}
\]
\[
\begin{align*}
ticks^{(k)} &= ticks^{(0)} + k(k+1)/2 + ki^{(0)} \\
i^{(k)} &= i^{(0)} + k \\
n^{(k)} &= n^{(0)}
\end{align*}
\]

\[
\text{for } (i = 0; i < n; i++):
\begin{align*}
&\text{for } (j = 0; j < i; j++): \\
&\quad \text{ticks++}
\end{align*}
\]

\[
\begin{align*}
& \exists k. \quad k \geq 0 \\
& \quad \land ticks' = ticks + k \\
& \quad \land j' = k
\end{align*}
\]
for (i = 0; i < n; i++):
    for (j = 0; j < i; j++):
        ticks++
Warm up: the linear case

Suppose loop body formula $F(x, x')$ is \textit{linear}.

Goal: find a linear system $y' = Ay + b + \text{linear transformation } T$ s.t

$$F(x, x') \models (Tx') = A(Tx) + b$$
Suppose loop body formula $F(x, x')$ is \textit{linear}.
Goal: find a linear system $y' = Ay + b + \text{linear transformation } T$ s.t

\[
F(x, x') \models (Tx') = A(Tx) + b
\]
Warm up: the linear case

Suppose loop body formula $F(x, x')$ is \textit{linear}.
Goal: find a linear system $y' = Ay + b + \text{linear transformation } T \text{ s.t.}

$$F(x, x') \models (Tx') = A(Tx) + b$$

Algorithm:

1. Compute the \textit{affine hull} of $F$ by sampling linearly independent models of $F$ using an SMT solver.
   Result is system of (all) equations $Ax' = Bx + c$ entailed by $F(x, x')$
Warm up: the linear case

Suppose loop body formula $F(x, x')$ is linear.

Goal: find a linear system $y' = Ay + b + \text{linear transformation } T$ s.t.

$$F(x, x') = (Tx') = A(Tx) + b$$

Algorithm:

1. Compute the affine hull of $F$ by sampling linearly independent models of $F$ using an SMT solver.
   Result is system of (all) equations $Ax' = Bx + c$ entailed by $F(x, x')$

2. Fixpoint computation:
   We have: $Ax' = Bx + c$

   Linear transformation $T$

   We need: $y' = By + c$
Warm up: the linear case

Suppose loop body formula $F(x, x')$ is **linear**.

Goal: find a linear system $y' = Ay + b + \text{linear transformation } T$ s.t

$$F(x, x') \models (Tx') = A(Tx) + b$$

Algorithm:

1. Compute the **affine hull** of $F$ by sampling linearly independent models of $F$ using an SMT solver. Result is system of (all) equations $Ax' = Bx + c$ entailed by $F(x, x')$

2. Fixpoint computation:

   We have: $Ax' = Bx + c$

   \[ T_0x' = T_0Bx + T_0c \]

We need: $y' = By + c$
Warm up: the linear case

Suppose loop body formula $F(x, x')$ is \textit{linear}.

Goal: find a linear system $y' = Ay + b + \text{linear transformation } T$ s.t

$$F(x, x') \models (Tx') = A(Tx) + b$$

Algorithm:

1. Compute the \textit{affine hull} of $F$ by sampling linearly independent models of $F$ using an SMT solver.
   Result is system of (all) equations $Ax' = Bx + c$ entailed by $F(x, x')$

2. Fixpoint computation:

   We have: $Ax' = Bx + c$

   $T_0x' = T_0Bx + T_0c$ \hspace{1cm} \text{computes best abstraction}

   $T_1x' = T_1Bx + T_1c$

   \vdots

   We need: $y' = By + c$
Reasoning about non-linear arithmetic
\[
\begin{align*}
\text{for } & (i = 0; \ i < n; \ i++): \\
\text{if } & (*): \ 	ext{continue} \\
\text{for } & (j = 0; \ j < n; \ j++): \\
& \quad \text{for } (k = 0; \ k < n; \ k++): \\
& \quad \quad \text{ticks++}
\end{align*}
\]
for (i = 0; i < n; i++):
    if (*):
        continue
    for (j = 0; j < n; j++):
        for (k = 0; k < n; k++):
            ticks++

\[
i' = i + 1
\wedge i < n
\wedge n' = n
\wedge \exists y \geq 0.
\left( \begin{aligned}
&\text{ticks'} = \text{ticks} \\
&\text{j'} = j \\
&\text{k'} = k \\
&\text{ticks'} = \text{ticks} + y \times n
\end{aligned} \right)
\]

\[\text{ticks'} \leq \text{ticks} + n^2\]
The wedge abstract domain

• The wedge domain is an abstract domain for reasoning about non-linear integer/rational arithmetic
  • The properties expressible by wedges correspond to the conjunctive fragment of non-linear arithmetic \((x \times y, x/y, x^y, \log_x(y), x \text{ mod } y, \ldots)\)
The *wedge* abstract domain

- The wedge domain is an abstract domain for reasoning about non-linear integer/rational arithmetic
  - The properties expressible by wedges correspond to the *conjunctive fragment* of non-linear arithmetic \((x \times y, x/y, x^y, \log_x(y), x \mod y, ...)\)

- Treat non-linear terms as independent dimensions
- Ignore \(\leq\), treat non-polynomial terms as independent dimensions
The **wedge** abstract domain

- The wedge domain is an abstract domain for reasoning about non-linear integer/rational arithmetic
  - The properties expressible by wedges correspond to the *conjunctive fragment* of non-linear arithmetic ($x \times y, x/y, x^y, \log_x(y), x \mod y, \ldots$)

![Diagram](image)

- Treat non-linear terms as independent dimensions
- Ignore $\leq$, treat non-polynomial terms as independent dimensions
- Linear equations
- Polyhedron
- Algebraic variety
The wedge abstract domain

- The wedge domain is an abstract domain for reasoning about non-linear integer/rational arithmetic.
  - The properties expressible by wedges correspond to the conjunctive fragment of non-linear arithmetic \((x \times y, x/y, x^y, \log_x(y), x \mod y, \ldots)\)

- Wedge Polyhedron
- Algebraic variety
- Inference rules
- Congruence closure

Wedge:
- treat non-linear terms as independent dimensions
- ignore \(\leq\), treat non-polynomial terms as independent dimensions

Linear equations:
- Polyhedron
- Algebraic variety

Diagram representation:
- Wedge connects Polyhedron and Algebraic variety.
- Inference rules and Congruence closure are connected to Polyhedron and Algebraic variety, respectively.
Symbolic abstraction

\[i' = i + 1\]
\[\land i < n\]
\[\land n' = n\]
\[\land \left( \begin{array}{c}
ticks' = ticks \\
\land j' = j \\
\land k' = k
\end{array} \right)\]
\[\land \left( \exists y \geq 0. \right) \left( \begin{array}{c}
ticks' = ticks + y \times n \\
\land j' = y = n \\
\land k' = n
\end{array} \right)\]
Symbolic abstraction

\[
\left( \begin{array}{c}
i' = i + 1 \\
\land i < n \\
\land n' = n \\
\land \text{ticks}' = \text{ticks} \\
\land j' = j
\end{array} \right) \lor \left( \begin{array}{c}
i' = i + 1 \\
\land i < n \\
\land n' = n \\
\land \text{ticks}' = \text{ticks} + \text{sk}_y \times n \\
\land j' = \text{sk}_y = n \\
\land k' = n
\end{array} \right)
\]
Symbolic abstraction

\[
\begin{align*}
&i' = i + 1 \\
&i < n \\
&n' = n \\
&\text{ticks}' = \text{ticks} \\
&j' = j \\
&0 \leq n \times n
\end{align*}
\]
\[
\vee
\begin{align*}
&i' = i + 1 \\
&i < n \\
&n' = n \\
&\text{ticks}' = \text{ticks} + n \times n \\
&j' = n \\
&k' = n \\
&0 \leq n \times n
\end{align*}
\]
Symbolic abstraction

\[
\begin{align*}
&i' = i + 1 \\
&i \leq n \\
&n' = n \\
&ticks \leq ticks' \leq ticks + n \times n \\
&j' = j \\
&0 \leq n \times n
\end{align*}
\]
Extracting recurrences

Given: non-linear transition formula \( F(x, x') \)

1. Compute wedge \( w \) that over-approximates \( F \)
2. Extract recurrences from \( w \)
Extracting recurrences

Given: non-linear transition formula \( F(x, x') \)

1. Compute wedge \( w \) that over-approximates \( F \)
2. Extract recurrences from \( w \)

Class of extractable recurrences:

\[
(Tx') = A(Tx) + t
\]

Additive term \( t \) involves polynomials & exponentials.
How can we solve recurrence equations?
Operational calculus is an algebra of infinite sequences. Idea:

1. Translate recurrence into equation in operational calculus
   \[ x^{(k+1)} = x^{(k)} + 1 \rightarrow qx - (q - 1)x_0 = x + 1 \]

2. Solve the equation
   \[ x = x_0 + \frac{1}{q - 1} \]

3. Translate solution back
   \[ x^{(k)} = x^{(0)} + k \]
**Operational Calculus**

**Field of operators:**
- Operator is a sequence with finitely many **negative positions**

\[ a = (a_{-2}, a_{-1} \parallel a_0, a_1, a_2, \ldots) \]

\[ b = (\parallel b_0, b_1, b_2, \ldots) \]

- **Addition** is pointwise: \((a + b)_i \triangleq a_i + b_i\)
- **Multiplication** is convolution difference:

\[ (ab)_n = \sum_{i=-\infty}^{n} a_i b_{n-i} + \sum_{i=-\infty}^{n-1} a_i b_{n-i-1} \]
Operational Calculus

Field of operators:
• Operator is a sequence with finitely many negative positions

\[ a = (a_{-2}, a_{-1} \| a_0, a_1, a_2, \ldots) \]

\[ b = (\| b_0, b_1, b_2, \ldots) \]

• Addition is pointwise: \((a + b)_i \triangleq a_i + b_i\)

• Multiplication is convolution difference:

\[ (ab)_n = \sum_{i=-\infty}^{n} a_i b_{n-i} + \sum_{i=-\infty}^{n-1} a_i b_{n-i-1} \]

• Left shift operator \(q = (1 \| 1, 1, 1, \ldots)\)

\[ qa = (a_{-2} a_{-1} a_0 \| a_1, a_2, a_3, \ldots) \]
Recurrence → operational calculus

Recurrences are equations in operational calculus

\[ x^{(k+1)} = x^k + t \quad \rightarrow \quad qx - (q - 1)x_0 = x + T_k(t) \]

- Think of \( x \) as an sequence (\( \| x_0, x_1, x_2, \ldots \)
Recurrence → operational calculus

**Recurrences are equations in operational calculus**

\[ x^{(k+1)} = x^k + t \iff qx - (q - 1)x_0 = x + T_k(t) \]

- Think of \( x \) as a sequence (\( \| x_0, x_1, x_2, \ldots \) )
- Use left-shift operator to write recurrence as an equation

\[
qx = (x_0 \| x_1, x_2, x_3, \ldots ) \\
(q - 1)x_0 = (x_0 \| 0, 0, 0, \ldots ) \\
qx - (q - 1)x_0 = (\| x_1, x_2, x_3, \ldots )
\]
Recurrence → operational calculus

Recurrences are equations in operational calculus

\[ x^{(k+1)} = x^k + t \implies qx - (q - 1)x_0 = x + T_k(t) \]

- Think of \( x \) as an sequence \((x_0, x_1, x_2, \ldots)\)
- Use left-shift operator to write recurrence as an equation

\[ qx = (x_0 \| x_1, x_2, x_3, \ldots) \]

\[ (q - 1)x_0 = (x_0 \| 0, 0, 0, \ldots) \]

\[ qx - (q - 1)x_0 = (\| x_1, x_2, x_3, \ldots) \]

Can translate any expression in the grammar

\[ s, t \in Expr(k) ::= c \in \mathbb{Q} \mid k \mid c^k \mid s + t \mid st \]

\[ T_k(c) = c \]

\[ T_k(ct) = cT_k(t) \]

\[ T_k(s + t) = T_k(s) + T_k(t) \]

\[ T_k(k) = \frac{1}{q - 1} \]

\[ \vdots \]
Operational Calculus $\rightarrow$ classical algebra

\[
\mathcal{T}_k(c) = c
\]
\[
\mathcal{T}_k(ct) = c\mathcal{T}_k(t)
\]
\[
\mathcal{T}_k(s + t) = \mathcal{T}_k(s) + \mathcal{T}_k(t)
\]
\[
\mathcal{T}_k(k) = \frac{1}{q-1}
\]

\[
\mathcal{T}_k^{-1}(c) = c
\]
\[
\mathcal{T}_k^{-1}(ct) = c\mathcal{T}_k^{-1}(t)
\]
\[
\mathcal{T}_k^{-1}(s + t) = \mathcal{T}_k^{-1}(s) + \mathcal{T}_k^{-1}(t)
\]
\[
\mathcal{T}_k^{-1}\left(\frac{1}{q-1}\right) = k
\]

\[
\vdots
\]

\[
\vdots
\]

Translation is not complete!
Operational Calculus $\rightarrow$ classical algebra

\[ T_k(c) = c \]
\[ T_k(ct) = cT_k(t) \]
\[ T_k(s + t) = T_k(s) + T_k(t) \]
\[ T_k(k) = \frac{1}{q-1} \]
\[ \vdots \]
\[ T_k^{-1}(c) = c \]
\[ T_k^{-1}(ct) = cT_k^{-1}(t) \]
\[ T_k^{-1}(s + t) = T_k^{-1}(s) + T_k^{-1}(t) \]
\[ T_k^{-1}\left(\frac{1}{q-1}\right) = k \]
\[ \vdots \]
\[ T_k^{-1}(t) = ? \]

Operational Calculus $\rightarrow$ classical algebra translation is not complete!
Operational Calculus $\rightarrow$ classical algebra

$$\mathcal{T}_k(c) = c$$
$$\mathcal{T}_k(ct) = c\mathcal{T}_k(t)$$
$$\mathcal{T}_k(s + t) = \mathcal{T}_k(s) + \mathcal{T}_k(t)$$
$$\mathcal{T}_k(k) = \frac{1}{q-1}$$

$$\mathcal{T}_k^{-1}(c) = c$$
$$\mathcal{T}_k^{-1}(ct) = c\mathcal{T}_k^{-1}(t)$$
$$\mathcal{T}_k^{-1}(s + t) = \mathcal{T}_k^{-1}(s) + \mathcal{T}_k^{-1}(t)$$
$$\mathcal{T}_k^{-1}\left(\frac{1}{q-1}\right) = k$$

$$\vdots$$

$$\vdots$$

$$\mathcal{T}_k^{-1}(t) = f_t(k)$$

Implicitly interpreted function
Operational Calculus $\rightarrow$ classical algebra

\[ T_k(c) = c \]
\[ T_k(ct) = cT_k(t) \]
\[ T_k(s + t) = T_k(s) + T_k(t) \]
\[ T_k(k) = \frac{1}{q-1} \]
\[ \vdots \]
\[ T_k(f_t(k)) = t \]

\[ T_k^{-1}(c) = c \]
\[ T_k^{-1}(ct) = cT_k^{-1}(t) \]
\[ T_k^{-1}(s + t) = T_k^{-1}(s) + T_k^{-1}(t) \]
\[ T_k^{-1}\left(\frac{1}{q-1}\right) = k \]
\[ \vdots \]
\[ T_k^{-1}(t) = f_t(k) \]
Experiments

ICRA: built on top of Z3, Apron.
Analyzes recursive procedures via [Kincaid, Breck, Boroujeni, Reps PLDI 2017]

<table>
<thead>
<tr>
<th>Benchmark Suite</th>
<th>Total</th>
<th>ICRA</th>
<th>UAut.</th>
<th>CPA</th>
<th>SEA</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>#A</td>
<td>Time</td>
<td>#A</td>
<td>Time</td>
<td>#A</td>
</tr>
<tr>
<td>HOLA</td>
<td>46</td>
<td>123.5</td>
<td>33</td>
<td>1571.9</td>
<td>20</td>
</tr>
<tr>
<td>functional</td>
<td>21</td>
<td>77.9</td>
<td>11</td>
<td>732.8</td>
<td>0</td>
</tr>
<tr>
<td>relational</td>
<td>10</td>
<td>8.1</td>
<td>10</td>
<td>473</td>
<td>0</td>
</tr>
<tr>
<td>Total</td>
<td>77</td>
<td>209.5</td>
<td>54</td>
<td>2777.7</td>
<td>20</td>
</tr>
</tbody>
</table>
Contributions:

- Wedge abstract domain
- Algorithm for extracting recurrences from loop bodies with control flow & non-determinism
- Recurrence solver that avoids algebraic numbers

Result: non-linear invariant generation for arbitrary loops