Compositional Recurrence Analysis

Azadeh Farzan          Zachary Kincaid

University of Toronto

September 28, 2015
Compositional program analysis

\[ P_1; P_2 \]
Compositional program analysis

Break program into parts

$P_1; P_2$

$P_1$

$P_2$
Compositional program analysis

Break program into parts

Analyze each part
Compositional program analysis

1. Break program into parts
   - $P_1; P_2$

2. Analyze each part
   - $[P_1]$
   - $[P_2]$

3. Compose the results
   - $[P_1; P_2]$
Compositional program analysis

Incremental analysis

Program into parts

Analyze each part

Compose the results

\[ P_1 \]

\[ P_2 \]

\[ \circ \]

\[ [P_1; P_2] \]
Compositional program analysis

1. Break program into parts: $P_1; P_2$
2. Analyze each part: $[P_1], [P_2]$
3. Compose the results: $[P_1; P_2]$
4. Incremental analysis
   - Compute in parallel
x := 0
c := 1
n := 100

while (x < n):
    x := x + c
assert (x == n)
\[ x := 0 \]
\[ c := 1 \]
\[ n := 100 \]

\textbf{while} (\( x < n \)):
\[ x := x + c \]
\textbf{assert} (\( x == n \))
Context

c := 1 ∧ n = 100 ∧ 0 ≤ x ≤ 100

x := 0

c := 1

n := 100

while (x < n):
    x := x + c

assert (x == n)
\[
\exists k. ((k \geq 1 \land x < n) \lor k = 0) \land x' = x + kc 
\]
How can we analyze programs compositionally and precisely?
Recurrence Analysis

```python
while(*):
    x := x + 1
    y := y - 2
```
Recurrence Analysis

while(*) :
    x := x + 1
    y := y - 2

Recurrences:

\[ x^{(k)} = x^{(k-1)} + 1 \]
\[ y^{(k)} = y^{(k-1)} - 2 \]
Recurrence Analysis

**while(\(*\))**:  
\[
\begin{align*}
  & x := x + 1 \\
  & y := y - 2
\end{align*}
\]

**Recurrences:**  
\[
\begin{align*}
  x^{(k)} &= x^{(k-1)} + 1 \\
  y^{(k)} &= y^{(k-1)} - 2
\end{align*}
\]

**Closed forms:**  
\[
\begin{align*}
  x^{(k)} &= x^{(0)} + 1k \\
  y^{(k)} &= y^{(0)} - 2k
\end{align*}
\]
Recurrence Analysis

\[\text{while}(\ast) :\]
\[
\begin{align*}
x & := x + 1 \\
y & := y - 2
\end{align*}
\]

Recurrences:
\[
\begin{align*}
x^{(k)} &= x^{(k-1)} + 1 \\
y^{(k)} &= y^{(k-1)} - 2
\end{align*}
\]

Closed forms:
\[
\begin{align*}
x^{(k)} &= x^{(0)} + 1k \\
y^{(k)} &= y^{(0)} - 2k
\end{align*}
\]

Loop abstraction:
\[
\exists k. k \geq 0 \land x' = x + k \land y' = y - 2k
\]
while (x + y < 10):
    z := z + 1
    if (x):
        x := x + rand(1, 3)
    else
        y := y + 1

w := w + x
\textbf{while} (z < 100):
\hspace{2em} x := 0
\hspace{2em} y := 0
\hspace{2em} \textbf{while} (x + y < 10):
\hspace{3em} z := z + 1
\hspace{3em} \textbf{if} (*):
\hspace{4em} x := x + \text{rand}(1, 3)
\hspace{3em} \textbf{else}
\hspace{4em} y := y + 1
\hspace{2em} w := w + x
How can we use recurrence analysis to compute approximations of arbitrary programs?
Compositional Recurrence Analysis
1. Compute a *path expression* to a point of interest (e.g., an assertion)
2. Evaluate the path expression in the *semantic algebra* defining the analysis
x := 0
n := 10
i := 0
outer: if(i >= n):
    goto end
    i := i + 1
inner: j := 0
if(*):
    if(*):
        x := x + 1
        j := j + 1
    if(j < n):
        goto inner
    goto inner
end: assert(x <= 100)
x := 0
n := 10
i := 0

outer: if(i >= n):
    goto end
i := i + 1

inner: j := 0

if(*):
    x := x + 1
j := j + 1
if(j < n):
    goto inner
goto outer

end: assert(x <= 100)
\begin{verbatim}
x := 0
n := 10
i := 0

outer: if (i >= n):
    goto end
i := i + 1

inner: j := 0
if (*):
    if (x):
        x := x + 1
j := j + 1
if (j < n):
    goto inner
goto outer
end: assert (x <= 100)
\end{verbatim}
x := 0
n := 10
i := 0

outer: if(i >= n):
    goto end
i := i + 1

inner: j := 0
if(*):
    x := x + 1
j := j + 1
if(j < n):
    goto inner
goto outer

end: assert(x <= 100)
x := 0
n := 10
i := 0

outer: if(i >= n):
    goto end
    i := i + 1

inner: j := 0
    if(*):
        x := x + 1
        j := j + 1
    if(j < n):
        goto inner
    goto outer

end: assert(x <= 100)
\[x := 0\]
\[n := 10\]
\[i := 0\]
\textbf{outer:} \hspace{1em} \textbf{if} (i >= n):
\hspace{2em} \textbf{goto} \hspace{1em} \textbf{end}
\hspace{2em} i := i + 1
\textbf{inner:} \hspace{1em} j := 0
\hspace{2em} \textbf{if} (*):
\hspace{3em} x := x + 1
\hspace{3em} j := j + 1
\hspace{2em} \textbf{if} (j < n):
\hspace{3em} \textbf{goto} \hspace{1em} \textbf{inner}
\hspace{2em} \textbf{goto} \hspace{1em} \textbf{outer}
\textbf{end:} \hspace{1em} \textbf{assert} (x <= 100)
\[x := 0\]
\[n := 10\]
\[i := 0\]
outer: if(i \geq n):
    \textbf{goto} end
    i := i + 1
inner: j := 0
    if(*):
        x := x + 1
        j := j + 1
    if(j < n):
        \textbf{goto} inner
\textbf{goto} outer
end: \textbf{assert}(x \leq 100)
x := 0
n := 10
i := 0
outer: if (i >= n):
    goto end
i := i + 1
inner: j := 0
if (*):
    if (*):
        x := x + 1
j := j + 1
if (j < n): goto inner
goto outer
end: assert (x <= 100)
Interpretation: \( \mathcal{I} = \langle \mathcal{D}, \llbracket \cdot \rrbracket \rangle \)

- \( \mathcal{D} = \langle D, \otimes, \oplus, \otimes, 0, 1 \rangle \) is a \textit{semantic algebra}
Interpretation: $\mathcal{I} = \langle \mathcal{D}, [\cdot] \rangle$

- $\mathcal{D} = \langle D, \circ, \oplus, \otimes, 0, 1 \rangle$ is a semantic algebra
Interpretation: \( \mathcal{I} = \langle \mathcal{D}, [\cdot] \rangle \)

- \( \mathcal{D} = \langle D, \circ, \oplus, \otimes, 0, 1 \rangle \) is a semantic algebra
Interpretation: $\mathcal{I} = \langle \mathcal{D}, [\cdot] \rangle$

- $\mathcal{D} = \langle D, \odot, \oplus, \otimes, 0, 1 \rangle$ is a semantic algebra
- $[\cdot] : \text{Control flow edges} \rightarrow D$ is a semantic function
**Interpretation:** $\mathcal{I} = \langle \mathcal{D}, [[ \cdot ]] \rangle$

- $\mathcal{D} = \langle D, \odot, \oplus, \otimes, 0, 1 \rangle$ is a **semantic algebra**
- $[[ \cdot ]] :$ *Control flow edges* $\rightarrow D$ is a **semantic function**

\[
\begin{align*}
[abc(def((h+g)ij)^* (h+g)ik)^*lm] &= [a] \odot [b] \odot [c] \\
&\quad \odot ( [d] \odot [e] \odot [f] \\
&\quad \quad \odot ([h] \oplus [g]) \odot [i] \odot [j])^* \\
&\quad \odot ([h] \oplus [g]) \odot [i] \odot [k])^* \\
&\quad \odot [l] \odot [m]
\end{align*}
\]
**Interpretation:** \( \mathcal{I} = \langle D, [\cdot] \rangle \)

- \( D = \langle D, \odot, \oplus, \otimes, 0, 1 \rangle \) is a **semantic algebra**
- \([\cdot] : \text{Control flow edges} \rightarrow D\) is a **semantic function**

**Compositional Recurrence Analysis**

- \( D\): set of arithmetic **transition formulas**

\[
[x := x + 1] \triangleq x' = x + 1 \land y' = y \land i' = i \land j' = j \land n' = n
\]
Interpretation: $\mathcal{I} = \langle \mathcal{D}, [\cdot] \rangle$

- $\mathcal{D} = \langle D, \circ, \oplus, \otimes, 0, 1 \rangle$ is a semantic algebra
- $[\cdot] : \text{Control flow edges} \rightarrow D$ is a semantic function

Compositional Recurrence Analysis

- $D$: set of arithmetic transition formulas

\[
[x := x + 1] \triangleq x' = x + 1 \land y' = y \land i' = i \land j' = j \land n' = n
\]

- $\varphi \odot \psi \triangleq \exists \bar{x}''. \varphi[\bar{x}' \mapsto \bar{x}''] \land \psi[\bar{x} \mapsto \bar{x}'']$
**Interpretation:** $\mathcal{I} = \langle \mathcal{D}, \llbracket \cdot \rrbracket \rangle$

- $\mathcal{D} = \langle D, \otimes, \ominus, \oplus, 0, 1 \rangle$ is a **semantic algebra**
- $\llbracket \cdot \rrbracket : \text{Control flow edges } \rightarrow D$ is a **semantic function**

**Compositional Recurrence Analysis**

- $D$: set of arithmetic **transition formulas**

\[ [x := x + 1] \triangleq x' = x + 1 \land y' = y \land i' = i \land j' = j \land n' = n \]

- $\varphi \otimes \psi \triangleq \exists \bar{x}''. \varphi[\bar{x}' \mapsto \bar{x}''] \land \psi[\bar{x} \mapsto \bar{x}'']$

- $\varphi \oplus \psi \triangleq \varphi \lor \psi$
**Interpretation:** \( \mathcal{I} = \langle D, \llbracket \cdot \rrbracket \rangle \)

- \( D = \langle D, \otimes, \oplus, \oslash, 0, 1 \rangle \) is a **semantic algebra**
- \( \llbracket \cdot \rrbracket : \) Control flow edges \( \rightarrow D \) is a **semantic function**

**Compositional Recurrence Analysis**

- \( D \): set of arithmetic **transition formulas**

\[
\llbracket x := x + 1 \rrbracket \triangleq x' = x + 1 \land y' = y \land i' = i \land j' = j \land n' = n
\]

- \( \phi \otimes \psi \triangleq \exists \vec{x}''. \phi[\vec{x}' \mapsto \vec{x}''] \land \psi[\vec{x} \mapsto \vec{x}''] \)
- \( \phi \oplus \psi \triangleq \phi \lor \psi \)
- \( \phi \oslash \triangleq \ldots \)
\[ [p^*] = [p]^\circ \]
**Problem**

*Given a transition formula $\varphi$ (representing the body of a loop), compute a formula $\varphi^\ast$ representing any number of iterations of the loop.*
Problem

Given a transition formula $\varphi$ (representing the body of a loop), compute a formula $\varphi^\ast$ representing any number of iterations of the loop.

First, **linearize** $\varphi$: compute a linear formula $\text{lin}(\varphi)$ such that $\varphi \models \text{lin}(\varphi)$.
Problem

Given a transition formula $\varphi$ (representing the body of a loop), compute a formula $\varphi^*$ representing any number of iterations of the loop.

First, linearize $\varphi$: compute a linear formula $\text{lin}(\varphi)$ such that $\varphi \models \text{lin}(\varphi)$.

Linearization via optimization modulo theories:
If $\varphi \models x \in [1, 10]$ and $y \in [2, 3]$, then

$$\varphi \models y \leq xy \leq 10y \land 2x \leq xy \leq 3x$$
Simple recurrences

while(*):
    c := 2 * x
if (c = 1):
    x := x + 2
else
    x := x + 1
y := y - 2
Simple recurrences

while(*):
    c := 2 * x
    if (c = 1):
        x := x + 2
    else
        x := x + 1
    y := y - 2

\[ c' = 2x \]
\[ \land ((c' = 1 \land x' = x + 2) \lor (c' \neq 1 \land x' = x + 1)) \]
\[ \land y' = y - 2 \]
Simple recurrences

\textbf{while}(*):
    \begin{align*}
    c & := 2 \times x \\
    \text{if } (c = 1): & \quad c' = 2x \\
    & \quad \land ((c' = 1) \\
    & \quad \quad \quad \land x' = x + 2) \\
    \text{else} & \quad \lor (c' \neq 1) \\
    & \quad \quad \land x' = x + 1) \\
    & \quad \land y' = y - 2 \\
    \end{align*}

\hspace{2cm}

\begin{align*}
    m : [c \mapsto 0, x \mapsto 0, y \mapsto 0, c' \mapsto 0, x' \mapsto 1, y' \mapsto -2] & \models \varphi_{\text{body}}
\end{align*}
Simple recurrences

while(*):
    c := 2 * x
if (c = 1):
    x := x + 2
else
    x := x + 1
y := y - 2

\(c' = 2x\)
\(\land ((c' = 1 \land x' = x + 2) \lor (c' \neq 1 \land x' = x + 1))\)
\(\land y' = y - 2\)

\(m : [c \mapsto 0, x \mapsto 0, y \mapsto 0, c' \mapsto 0, x' \mapsto 1, y' \mapsto -2] \models \varphi_{\text{body}}\)

\((c' - c)^m = 0\)
\((x' - x)^m = 1\)
\((y' - y)^m = -2\)
Simple recurrences

while(\(*\)):
    \(c := 2 \times x\)
    if \((c = 1)\):
        \(x := x + 2\)
    else
        \(x := x + 1\)
    \(y := y - 2\)

\[m : [c \mapsto 0, x \mapsto 0, y \mapsto 0, c' \mapsto 0, x' \mapsto 1, y' \mapsto -2] \models \varphi_{\text{body}}\]

\[(c' - c)^m = 0\]
\[(x' - x)^m = 1\]
\[(y' - y)^m = -2\]
Simple recurrences

while(*):
    c := 2 * x
    if (c = 1):
        x := x + 2
    else
        x := x + 1
    y := y - 2

m : [c → 0, x → 0, y → 0, c' → 0, x' → 1, y' → -2] |= \varphi_{\text{body}}

(c' - c)^m = 0
(x' - x)^m = 1
(y' - y)^m = -2

\varphi_{\text{body}} \not\models c' = c + 0
\varphi_{\text{body}} \models x' = x + 1
\varphi_{\text{body}} \models y' = y - 2
Stratified recurrences

```python
while(*):
    x := x + 1
    y := y + x
    z := z + y
```
Linear recurrences (in)equations

while (0 <= i < 100):
    if (*):
        x := x + i
    else
        y := y + i
    i := i + 1

\( 0 \leq i \land i < 100 \)
\( \land x' = x + i \)
\( \land y' = y + i \)
\( \land i' = i + 1 \)
Linear recurrences (in)equations

\[
\text{while}(0 \leq i < 100): \quad 0 \leq i \land i < 100 \\
\text{if } (*):\ \\
\quad x := x + i \quad \land x' = x + i \\
\text{else} \\
\quad y := y + i \quad \land y' = y + i \\
i := i + 1 \quad \land i' = i + 1
\]

1. Introduce \textit{difference variables} for non-induction variables:

\[
\psi \triangleq \varphi_{\text{body}} \land \delta_x = x' - x \land \delta_y = y' - y
\]
Linear recurrences (in)equations

\begin{verbatim}
while(0 <= i < 100):
    if (*):
        x := x + i  \quad \wedge x' = x + i
    else
        y := y + i  \quad \wedge y' = y + i
    i := i + 1  \quad \wedge i' = i + 1
\end{verbatim}

1) Introduce \textit{difference variables} for non-induction variables:

\[
\psi \triangleq \varphi_{\text{body}} \land \delta_x = x' - x \land \delta_y = y' - y
\]

2) Project + compute the \textit{convex hull}:
   \begin{itemize}
   \item Smallest polyhedron \( P \) such that \( \exists x, y, x', y', i'.\psi \models P \)
Linear recurrences (in)equations

\[
\text{while}(0 \leq i < 100): \quad 0 \leq i \land i < 100
\]

\[
\text{if } (\ast): \quad x := x + i \quad \land x' = x + i
\]

\[
\text{else} \quad y := y + i \quad \land y' = y + i
\]

\[
i := i + 1 \quad \land i' = i + 1
\]

\[
\delta_x + \delta_y = i
\]

\[
0 \leq \delta_x \leq i
\]

\[
0 \leq \delta_y \leq i
\]

\[
\ldots
\]
Linear recurrences (in)equations

\[\text{while}(0 \leq i < 100): \quad 0 \leq i \land i < 100\]

\[\begin{align*}
\text{if (*)}: & \quad \begin{aligned}
& x := x + i \\
& \land x' = x + i
\end{aligned} \\
\text{else} & \quad \begin{aligned}
& y := y + i \\
& \land y' = y + i \\
& i := i + 1 \\
& \land i' = i + 1
\end{aligned}
\end{align*}\]

\[\begin{align*}
\delta_x + \delta_y &= i \\
0 \leq \delta_x \leq i \\
0 \leq \delta_y \leq i \\
\end{align*}\]

\[\begin{align*}
(x' - x) + (y' - y) &= i \\
0 \leq (x' - x) \leq i \\
0 \leq (y' - y) \leq i \\
\end{align*}\]
Linear recurrences (in)equations

\[\text{while}(0 \leq i < 100):\]
\[\begin{align*}
\text{if (\star)}: \\
&x := x + i \\
&\text{else} \\
&y := y + i \\
i := i + 1
\end{align*}\]
\[\begin{align*}
\delta_x + \delta_y &= i \\
0 &\leq \delta_x \leq i \\
0 &\leq \delta_y \leq i \\
0 &\leq (x' - x) \leq i \\
0 &\leq (y' - y) \leq i \\
\end{align*}\]
\[\begin{align*}
(x' - x) + (y' - y) &= i \\
x' + y' &= x + y + i \\
x &\leq x' &\leq x + i \\
x &\leq y' &\leq y + i
\end{align*}\]
Linear recurrences (in)equations

\[
\textbf{while}(0 \leq i < 100): \quad 0 \leq i \land i < 100 \\
\quad \textbf{if} (\ast): \quad x := x + i \quad \land x' = x + i \\
\quad \textbf{else} \quad y := y + i \quad \land y' = y + i \\
\quad i := i + 1 \quad \land i' = i + 1
\]

\[
\delta_x + \delta_y = i \\
0 \leq \delta_x \leq i \\
0 \leq \delta_y \leq i \\
\ldots
\]

\[
(x' - x) + (y' - y) = i \\
0 \leq (x' - x) \leq i \\
0 \leq (y' - y) \leq i
\]

\[
x^{(k)} + y^{(k)} = x^{(0)} + y^{(0)} + ki^{(0)} + k(k+1)/2 \\
x^{(0)} \leq x^{(k)} \leq x^{(0)} + ki^{(0)} + k(k+1)/2 \\
x^{(0)} \leq y^{(k)} \leq y^{(0)} + ki^{(0)} + k(k+1)/2
\]
Putting it all together

$$\varphi_{\text{body}} = \bigwedge_r \sum_i a_{ri} x'_i \leq \sum_i a_{ri} x_i + \sum_j b_{rj} y_{rj} + c_r$$
Putting it all together

\[ \varphi_{\text{body}} = \bigwedge_r \sum_i a_{ri} x'_ri \leq \sum_i a_{ri} x_{ri} + \sum_j b_{rj} y_{rj} + c_r \]

\[ \sum_{ri} a_{ri} x_{ri}^{(k)} \leq \sum_{ri} a_{ri} x_{ri}^{(0)} + \sum_j p_{rj}(k) y_{rj}^{(0)} + kc_r \]

Closed form
Putting it all together

$$\varphi_{body} \equiv \bigwedge_r \sum_i a_r i x'_r i \leq \sum_i a_r i x_r i + \sum_j b_r j y_{r j} + c_r$$

$$\sum_{r_i} a_r i x_{r_i}^{(k)} \leq \sum_{r_i} a_r i x_{r_i}^{(0)} + \sum_j p_r j (k) y_{r j}^{(0)} + k c_r$$

$$\varphi_{\text{body}}^\otimes \equiv \bigwedge_i x'_i = x_i$$

$$\lor (\exists k. k \geq 1 \land (\exists \bar{x}'. \varphi_{\text{body}}) \land (\exists \bar{x}. \varphi_{\text{body}})$$

$$\land \bigwedge_r \sum_i a_r i x'_r i \leq \sum_i a_r i x_r i + \sum_j p_r j (k) y_{r j} + k c_r)$$
Experimental evaluation on

- 74 safe benchmarks from SVComp15
- 7 safe non-linear benchmarks

<table>
<thead>
<tr>
<th>Tool</th>
<th>Solved</th>
</tr>
</thead>
<tbody>
<tr>
<td>CRA</td>
<td>65%</td>
</tr>
<tr>
<td>CRA+Oct</td>
<td>88%</td>
</tr>
<tr>
<td>SeaHorn</td>
<td>85%</td>
</tr>
<tr>
<td>CPAChecker</td>
<td>47%</td>
</tr>
</tbody>
</table>
Summary

CRA is *compositional* yet *precise*
Summary

CRA is *compositional* yet *precise*

**Compositional analysis**

+ **SMT-based recurrence detection**

*Approximate* recurrence analysis for *arbitrary loops*