Static bounds on program running time

Caleb Stanford

University of Pennsylvania

December 12, 2016
Partial and total correctness

\{P\} C \{Q\}
Partial and total correctness

\[
\{P\} \ C \ \{Q\}
\]

Partial Correctness

\[(P \land h) \rightarrow Q\]
Partial and total correctness

\{P\} \quad C \quad \{Q\}

**Total Correctness**

\[ P \rightarrow (h \land Q) \]

**Partial Correctness**

\[ (P \land h) \rightarrow Q \]
Partial and total correctness

\[ \{P\} \ C \ \{Q\} \]

**Total Correctness**

\[ P \rightarrow (h \land Q) \]

✓ semantic bugs

**Partial Correctness**

\[ (P \land h) \rightarrow Q \]

✓ semantic bugs
Partial and total correctness

\[ \{ P \} \ C \ \{ Q \} \]

**Total Correctness**

\[ P \rightarrow (h \land Q) \]

- ✔ semantic bugs
- ✔ nontermination

**Partial Correctness**

\[ (P \land h) \rightarrow Q \]

- ✔ semantic bugs
- X nontermination

\[ \{ X = n \} \]

```
Total := 0
I := 1
while I <= X :
    Total := Total + I
```

\[ \{ \text{Total} = \frac{n(n+1)}{2} \} \]
Partial and total correctness

\[ \{ P \} \quad C \quad \{ Q \} \]

**Total Correctness**

\[ P \rightarrow (h \land Q) \]

- ✔️ semantic bugs
- ✔️ nontermination
- ✗ performance bugs

**Partial Correctness**

\( (P \land h) \rightarrow Q \)

- ✔️ semantic bugs
- ✗ nontermination
- ✗ performance bugs

\[ \{ X = n \} \]

```
Total := 0
I := 1
while I <= 2000000000 :
    if (I <= X) :
        Total := Total + I
    I := I + 1

\{ Total = \frac{n(n+1)}{2} \} 
```
Partial and total correctness

\[ \{ P \} \ C \ \{ Q \} \]

**Total Correctness** \( P \rightarrow (h \land Q) \)
- ✓ semantic bugs
- ✓ nontermination
- ✗ performance bugs

**Partial Correctness** \((P \land h) \rightarrow Q\)
- ✓ semantic bugs
- ✗ nontermination
- ✗ performance bugs

**Partial Correctness + Time > Total Correctness** [1]
Overview

1. Introduction
2. SPEED: Goals
3. SPEED: Technique
4. SPEED: Implementation
5. Future research
SPEED: Precise and Efficient Static Estimation of Program Computational Complexity

[2]: Sumit Gulwani, Krishna K Mehra, and Trishul Chilimbi, Microsoft Research, 2009
SPEED: Precise and Efficient Static Estimation of Program Computational Complexity

[2]: Sumit Gulwani, Krishna K Mehra, and Trishul Chilimbi, Microsoft Research, 2009

**Goal:** Automatically generate upper bounds on the running time of simple programs
SPEED: Precise and Efficient Static Estimation of Program Computational Complexity

[2]: Sumit Gulwani, Krishna K Mehra, and Trishul Chilimbi, Microsoft Research, 2009

**Goal:** Automatically generate upper bounds on the running time of number of loop iterations of simple programs
SPEED: Precise and Efficient Static Estimation of Program Computational Complexity

[2]: Sumit Gulwani, Krishna K Mehra, and Trishul Chilimbi, Microsoft Research, 2009

Goal: Automatically generate upper bounds on the running time of number of loop iterations of simple programs

Requirements:
- Able to contain $+, \cdot, \min, \max$
SPEED: Precise and Efficient Static Estimation of Program Computational Complexity

[2]: Sumit Gulwani, Krishna K Mehra, and Trishul Chilimbi, Microsoft Research, 2009

**Goal:** Automatically generate upper bounds on the running time of number of loop iterations of simple programs

**Requirements:**
- Able to contain $+,\cdot,\min,\max$
- Precise (tight)
SPEED: Precise and Efficient Static Estimation of Program Computational Complexity

[2]: Sumit Gulwani, Krishna K Mehra, and Trishul Chilimbi, Microsoft Research, 2009

**Goal:** Automatically generate upper bounds on the running time of number of loop iterations of simple programs

**Requirements:**
- Able to contain $+,-,\min,\max$
- Precise (tight)
- Correct
SPEED bound generation

Requirements:

- Able to contain $+, \cdot, \text{min}, \text{max}$
- Precise (tight)
- Correct

Also:

★ Built from linear constraints
SPEED bound generation

Requirements:

- Able to contain $+$, $\cdot$, min, max
- Precise (tight)
- Correct

Also:

- Built from linear constraints (allows use of linear invariant generation tool)
SPEED bound generation

Requirements:

- Able to contain $+, \cdot, \min, \max$
- Precise (tight)
- Correct

Also:

- ★ Built from linear constraints (allows use of linear invariant generation tool)
- ★ Small enough search space
def f1(x0, y0, m, n):
    x := x0; y := y0
    while (x < m):
        if (y < n):
            y++
        else:
            x++
def f1(x0, y0, m, n):
    x := x0; y := y0
    while (x < m):
        if (y < n):
            y++
        else:
            x++

**Loop iterations:** \( (m - x_0) + (n - y_0) \)
def f1(x0, y0, m, n):
    x := x0; y := y0
    while (x < m):
        if (y < n):
            y++
        else:
            x++

Loop iterations: 0 if \( x_0 \geq m \); otherwise, \( m - x_0 + \max(0, n - y_0) \).
Upper bound: \( \max(0, m - x_0) + \max(0, n - y_0) \).
def f1(x0, y0, m, n):
    x := x0; y := y0; T := 0
    while (x < m):
        if (y < n):
            y++
        else:
            x++
        T++

Loop iterations: 0 if \( x_0 \geq m \); otherwise, \( m-x_0 + \max(0, n-y_0) \).
Upper bound: \( \max(0, m-x_0) + \max(0, n-y_0) \).
```python
def f1(x0, y0, m, n):
    x := x0; y := y0; T := 0
    while (x < m):
        if (y < n):
            y++
        else:
            x++
        T++ {T + x0 + y0 = x + y}
```

**Loop iterations:** 0 if $x_0 \geq m$; otherwise, $m - x_0 + \max(0, n - y_0)$.

**Upper bound:** $\max(0, m - x_0) + \max(0, n - y_0)$. 
def f1(x0, y0, m, n):
    x := x0; y := y0; T := 0
    while (x < m):
        if (y < n):
            y++
        else:
            x++
        T++
    \{ T + x_0 + y_0 = x + y \land x \leq n \land y \leq \max(n, y_0) \} \\
    \implies \{ T \leq \max(0, m - x_0) + \max(0, n - y_0) \} \\

**Loop iterations:** 0 if $x_0 \geq m$; otherwise, $m - x_0 + \max(0, n - y_0)$.

**Upper bound:** $\max(0, m - x_0) + \max(0, n - y_0)$. 
def f1(x0, y0, m, n):
    x := x0; y := y0; c1 := 0; c2 := 0
    while (x < m):
        if (y < n):
            y++; c2++
        else:
            x++; c1++

Loop iterations: 0 if $x_0 \geq m$; otherwise, $m - x_0 + \max(0, n - y_0)$.  
Upper bound: $\max(0, m - x_0) + \max(0, n - y_0)$. 

def f1(x0, y0, m, n):
    x := x0; y := y0; c1 := 0; c2 := 0
    while (x < m):
        if (y < n):
            y++; c2++  \{y_0 + c_2 = y \land y \leq n\}
            \implies \{c_2 \leq n - y_0\}
        else :
            x++; c1++  \{x_0 + c_1 = x \land x \leq m\}
            \implies \{c_1 \leq m - x_0\}

\{T = c_1 + c_2 \leq \max(0, m - x_0) + \max(0, n - y_0)\}

Loop iterations: 0 if \(x_0 \geq m\); otherwise, \(m - x_0 + \max(0, n - y_0)\).
Upper bound: \(\max(0, m - x_0) + \max(0, n - y_0)\).
def f2(x0, y0, n):
    x := x0; y := y0
    while (x < n):
        if (x < y):
            x++
        else:
            y++
def f2(x0, y0, n):
x := x0; y := y0
while (x < n):
    if (x < y):
        x++
    else:
        y++

Loop iterations:

• 0 if $x_0 \geq n$
• $n - x_0$ if $x_0 < n \leq y_0$
• $(n - x_0) + (n - y_0)$ if $x_0, y_0 < n$

Upper bound: $\max(0, m - x_0) + \max(0, n - y_0)$. 
def f2(x0, y0, n):
x := x0; y := y0; c1 := 0; c2 := 0
while (x < n):
    if (x < y):
        x++; c1++  \(\{x_0 + c_1 = x \land x \leq n\}\)
        \(\implies \{c_1 \leq n - x_0\}\)
    else:
        y++; c2++  \(\{y_0 + c_2 = y \land y \leq n\}\)
        \(\implies \{c_2 \leq n - y_0\}\)

\(\{T = c_1 + c_2 \leq \max(0, n - x_0) + \max(0, n - y_0)\}\)

**Loop iterations:**

- 0 if \(x_0 \geq n\)
- \(n - x_0\) if \(x_0 < n \leq y_0\)
- \((n - x_0) + (n - y_0)\) if \(x_0, y_0 < n\)

**Upper bound:** \(\max(0, m - x_0) + \max(0, n - y_0)\).
Strategy

• Assign a loop counter for every back-edge in the program
Strategy

- Assign a loop counter for every back-edge in the program
- Compute a linear bound on the value of the loop counter at each back-edge
Strategy

- Assign a loop counter for every back-edge in the program
- Compute a linear bound on the value of the loop counter at each back-edge

\[
\begin{align*}
c_3 & \leq B_1 \\
c_1 & \leq B_2 \\
c_2 & \leq B_3 \\
c_3 & \leq B_4 \\
c_1 & \leq B_5
\end{align*}
\]
Strategy

- Assign a loop counter for every back-edge in the program
- Compute a **linear** bound on the value of the loop counter at each back-edge

\[
\begin{align*}
c_3 & \leq B_1 \\
c_1 & \leq B_2 \\
c_2 & \leq B_3 \\
c_3 & \leq B_4 \\
c_1 & \leq B_5
\end{align*}
\]

\[
T = c_1 + c_2 + c_3 \leq \max(0, B_2, B_5) + \max(0, B_3) + \max(0, B_1, B_4)
\]
def f3(m, n):
  x := 0; y := 0
  while (x < m):
    if (y < n):
      y++
    else:
      y = 0
      x++
def f3(m, n):
    x := 0; y := 0
    while (x < m):
        if (y < n):
            y++
        else:
            y = 0
        x++

Loop iterations:
- 0 if $m \leq 0$
- $m$ if $n \leq 0 < m$
- $m(n + 1)$ if $0 < m, n$

Upper bound: $\max(m, 0) + (\max(m, 0) + 1) \cdot \max(n, 0)$. 
def f3(m, n):
    x := 0; y := 0; c1 := 0; c2 := 0
    while (x < m):
        if (y < n):
            y++; c2++
        else:
            y = 0
            x++; c1++; c2 = 0

Loop iterations:
- 0 if $m \leq 0$
- $m$ if $n \leq 0 < m$
- $m(n + 1)$ if $0 < m, n$

Upper bound: $\max(m, 0) + (\max(m, 0) + 1) \cdot \max(n, 0)$. 
def f3(m, n):
    x := 0; y := 0; c1 := 0; c2 := 0
    while (x < m):
        if (y < n):
            y++; c2++ \{y = c_2 \land y \leq n\}
            \implies \{c_2 \leq n\}
        else:
            y = 0
            x++; c1++; c2 = 0 \{x = c_1 \land x \leq m\}
            \implies \{c_1 \leq m\}

Loop iterations:

- 0 if \(m \leq 0\)
- \(m\) if \(n \leq 0 < m\)
- \(m(n + 1)\) if \(0 < m, n\)

Upper bound: \( \max(m, 0) + (\max(m, 0) + 1) \cdot \max(n, 0). \)
Strategy

- Assign a loop counter for every back-edge in the program
- **Assign DAG of loop counter dependencies**
- Compute a **linear** bound on the value of the loop counter at each back-edge
Strategy

- Assign a loop counter for every back-edge in the program
- Assign DAG of loop counter dependencies
- Compute a **linear** bound on the value of the loop counter at each back-edge

\[
\begin{align*}
    c_3 & \leq B_1 \\
    c_1 & \leq B_2 \\
    c_2 & \leq B_3 \\
    c_4 & \leq B_4 \\
    c_1 & \leq B_5
\end{align*}
\]

\[
\begin{align*}
    T_1 & \leq \max(0, B_2) \\
    T_2 & \leq (1 + T_1) \max(0, B_3) \\
    T_3 & \leq (1 + T_2) \max(0, B_1) \\
    T_4 & \leq \max(0, B_4) \\
    T = T_1 + T_2 + T_3 + T_4
\end{align*}
\]
Strategy

- Assign a loop counter for every back-edge in the program
- Assign DAG of loop counter dependencies
- Compute a **linear** bound on the value of the loop counter at each back-edge

\[ c_3 \leq B_1 \]
\[ c_1 \leq B_2 \]
\[ c_2 \leq B_3 \]
\[ c_4 \leq B_4 \]
\[ c_1 \leq B_5 \]

\[ T_1 \leq \max(0, B_2, B_5) \]
\[ T_2 \leq (1 + T_1) \max(0, B_3) \]
\[ T_3 \leq (1 + T_2) \max(0, B_1) \]
\[ T_4 \leq \max(0, B_4) \]

\[ T = T_1 + T_2 + T_3 + T_4 \]
SPEED bound generation

Requirements:

✓ Able to contain $+, \cdot, \min, \max$

★ Precise (tight)

✓ Correct

✓ Built from linear constraints (allows use of linear invariant generation tool)

✓/★ Small enough search space
Search Space

- Pick a number of variables
Search Space

- Pick a number of variables
- Assign a variable to each back-edge (> exponential)
Search Space

- Pick a number of variables
- Assign a variable to each back-edge (> exponential)
- Assign DAG of variable dependencies (> exponential)
Search Space

- Pick a number of variables
- Assign a variable to each back-edge ($>\text{exponential}$)
- Assign DAG of variable dependencies ($>\text{exponential}$)

Optimal bound in search space
Search Space

- Pick a number of variables
- Assign a variable to each back-edge ($>\text{exponential}$)
- Assign DAG of variable dependencies ($>\text{exponential}$)

Optimal bound in search space

“Counter-optimal” bound
Strategy

- Assign a loop counter for every back-edge in the program
- Assign DAG of loop counter dependencies
- Compute a **linear** bound on the value of the loop counter at each back-edge

And:

- **Minimize number of counters**
Strategy

- Assign a loop counter for every back-edge in the program
- Assign DAG of loop counter dependencies
- Compute a **linear** bound on the value of the loop counter at each back-edge

And:

- Minimize number of counters
- Minimize number of dependencies
• Minimize number of counters

def f4(n):
    x = 0
    while (x < n):
        if (*) :
            x++
        else
            x++
• **Minimize number of counters**

```python
def f5(n):
    x = 0
    while (x < n and rand({0,1}) == 0):
        x++
    while (x < n):
        x++
```
Algorithm

Repeat:
  • Pick an unassigned back-edge and assign it a counter.
Algorithm

Repeat:
  • Pick an unassigned back-edge and assign it a counter.
  • If an existing counter works, use that.
Algorithm

Repeat:
  • Pick an unassigned back-edge and assign it a counter.
    • If an existing counter works, use that.
    • Try to define a new counter.
Algorithm

Repeat:

- Pick an unassigned back-edge and assign it a counter.
  - If an existing counter works, use that.
  - Try to define a new counter.
  - Otherwise, fail.
Algorithm

Repeat:
- Pick an unassigned back-edge and assign it a counter.
  - If an existing counter works, use that.
  - Try to define a new counter.
  - Otherwise, fail.

Whenever a new counter is added:
- Add all dependencies from previous counters.
Algorithm

Repeat:

- Pick an unassigned back-edge and assign it a counter.
  - If an existing counter works, use that.
  - Try to define a new counter.
  - Otherwise, fail.

Whenever a new counter is added:

- Add all dependencies from previous counters.
- Remove one at a time until the invariant generation fails.
Implementation

- Quantitative functions on data structures

\[ \text{len } A, \text{ size } T, \text{ location of } x \text{ in } L \]
Implementation

- Quantitative functions on data structures
  \[ \text{len } A, \text{size } T, \text{location of } x \text{ in } L \]

- C/C++
Implementation

• Quantitative functions on data structures
  \[ \text{len } A, \text{ size } T, \text{ location of } x \text{ in } L \]

• C/C++

• Precise bounds on over 50% of loops in Microsoft product code
Possible research directions

• Nested max

\[
\begin{align*}
\max(0, m - x_0 + \max(0, n - y_0)) \\
\leq \max(0, m - x_0) + \max(0, n - y_0)
\end{align*}
\]
Possible research directions

- Nested max

\[
\begin{align*}
\max(0, m - x_0 + \max(0, n - y_0)) \\
\leq \max(0, m - x_0) + \max(0, n - y_0)
\end{align*}
\]

- Optimal bound (rather than minimizing counters and dependencies)
Possible research directions

- **Nested max**

\[
\begin{align*}
\max(0, m - x_0 + \max(0, n - y_0)) & \\
\leq \max(0, m - x_0) + \max(0, n - y_0)
\end{align*}
\]

- **Optimal bound** (rather than minimizing counters and dependencies)
- **Scenarios where the invariant generation fails:**
Possible research directions

• Nested max

\[
\max(0, m - x_0 + \max(0, n - y_0)) \\
\leq \max(0, m - x_0) + \max(0, n - y_0)
\]

• Optimal bound (rather than minimizing counters and dependencies)

• Scenarios where the invariant generation fails:
  • Invariant generation tool required a global fact
Possible research directions

- Nested max

\[
\max(0, m - x_0 + \max(0, n - y_0)) \\
\leq \max(0, m - x_0) + \max(0, n - y_0)
\]

- Optimal bound (rather than minimizing counters and dependencies)

- Scenarios where the invariant generation fails:
  - Invariant generation tool required a global fact
  - Linear bounds require path-sensitive invariant generation
Possible research directions

- Nested max

\[
\max(0, m - x_0 + \max(0, n - y_0)) \leq \max(0, m - x_0) + \max(0, n - y_0)
\]

- Optimal bound (rather than minimizing counters and dependencies)

- Scenarios where the invariant generation fails:
  - Invariant generation tool required a global fact
  - Linear bounds require path-sensitive invariant generation

- Other types of counters, placement, and dependency
References I

Eric CR Hehner.
Specifications, programs, and total correctness.

Sumit Gulwani, Krishna K Mehra, and Trishul Chilimbi.
Speed: precise and efficient static estimation of program computational complexity.