SPEED: precise and efficient static estimation of program computational complexity

Andrey Breslav

report on a paper by
S. Gulwani, K. Mehra and T. Chilimbi

University of Tartu
2010
Outline

• Linear Invariant Generators
• Instrumenting Programs with Counters
• The notion of a “Proof Structure”
• Optimal Proof Structures
• Quantitative Functions over Data Structures
• Inter-procedural Bounds
• Summary
The Problem

- Given a program $P(\text{in}_1, \ldots, \text{in}_n)$
- Find a total function $T(\text{in}_1, \ldots, \text{in}_n) : \text{Nat}$
  - NB: not $T(|\text{in}_1|, \ldots, |\text{in}_n|)$
- Such that
  - Execution time of $P \in O(T)$
- Properties
  - Existence of an explicit form of $T$ implies termination of $P$
  - The problem is undecidable in general
- Solution
  - Reduce it to a “less undecidable” problem
Control Flow Graph (CFG)

- \( x := \text{input()} \)
- \( y := \text{input()} \)
- \( \text{while } x \times y \neq 0 \text{ do} \)
  - \( \text{if } x > y \text{ then} \)
    - \( x := x \mod y \)
  - \( \text{else} \)
    - \( y := y \mod x \)
- \( \text{WriteLn}(x + y) \)

Number of loop iterations = number of times each back-edge is taken (maybe + 1)
Invariant

- **Invariant** is a boolean expression
  - over program variables
  - written on a CFG edge
  - which is true whenever this edge is taken
- Example
  - `while x * y > 0 do`
    - `{x <> 0} {y <> 0}`
      - if `x > y` then
        - `x := x mod y {x < y}`
      - else
        - `y := y mod x {y < x}`
  - `{x = 0 \lor y = 0}
  - `WriteLn(x + y)`

**Linear Invariant Generators**

Can sometimes generate invariants of the form

\[ c_1 x + c_2 y + c_3 > 0 \]
Linear Invariant Generator

Program

Target Edges

Linear Invariant Generator

Invariants

Hopefully, nontrivial
Approach

- **procedure** `Dis1(x0, y0, n, m : Int)`
  - `c1 := 0; c2 := 0;`
  - `x := x0; y := y0;`
  - **while** `(x < n)`
    - **if** `(y < m)`
      - `y := y + 1; c1++; {c1 <= m - y0}`
    - **else**
      - `x := x + 1; c2++; {c2 <= n - x0}`

$O(\text{Max}(0, m - y0) + \text{Max}(0, n - x0))$
Points to Refine

- How many counters do we need?
  - We are looking for precise bounds
- Where should we initialize the counters?
- How do we combine the linear invariants?
- What if the generator says "TRUE"?
  - Too many/few counters
  - Complex data structures
“Proof Structure”

- At each **back-edge** $E$ we increment some counter $C(E)$

- **Dependencies** form a DAG
  - Nodes are **counters** and the **entry-point** $R$
  - If $N$ depends on $M$: $N := 0$; and $M++;$ go together
  - If $N$ depends on $R$: $N := 0$; is on the first line

- At every edge $E$ our invariant generator **can generate** a $Bound(E)$ for the counter $C(E)$
  - The generator is an important parameter
Example of a Proof Structure

- **procedure Dis1(int x0, y0, n, m)**
  - c1 := 0; c2 := 0;
  - x := x0; y := y0;
  - **while** (x < n)
    - **if** (y < m)
      - y := y + 1;
    - **else**
      - x := x + 1;

Dependency graph:
- c1++;
- c2++;
Computing Bounds

• Total bound for a counter:

\[
\text{TotalBound} \left( R \right) = 0 \\
\text{TotalBound} \left( c \right) = \text{Max} \left( \left\{ 0 \right\} \cup \left\{ \text{Bound} \left( E \right) \mid C \left( E \right) = c \right\} \right) \times \left( 1 + \sum_{(c',c) \in G} \text{TotalBound} \left( c' \right) \right)
\]

- We do not compute the maximum, but use it symbolically
- Each loop might never be entered: Max(0, …)
- Sum of all total bounds may be zero: 1 + …

• Total bound for the procedure:

\[
\text{UpperBound} = \sum_c \text{TotalBound} \left( c \right)
\]
Example of Bound Calculation

- **procedure** Dis1(int x0, y0, n, m)
  - c1 := 0; c2 := 0;
  - x := x0; y := y0;
  - **while** (x < n)
    - **if** (y < m)
      - y := y + 1;
    - **else**
      - x := x + 1;
  
  \[
  \text{TotalBound}(c1) = \text{Max}(0, y0 - m) \times 1 \\
  \text{TotalBound}(c2) = \text{Max}(0, x0 - n) \times 1 \\
  \text{UpperBound} = \text{Max}(0, y0 - m) + \text{Max}(0, x0 - n)
  \]
Optimality of Proof Structures

- Optimal PS: A PS that gives UpperBound not greater than any other PS for the same P
  - Hard to compare
  - Hard to construct
- C-Optimal PS (Counter-Optimal)
  - Less counters (no node merge is possible)
  - Less dependencies (no edge deletion is possible)
  - The generator must be able to find bounds
  - No loops are allowed in dependence graph
Finding C-Optimal PS (1)

- **Node Merging**
  - Let $c_1$ and $c_2$ not depend on each other (transitively)
  - Replace $C(E) = c_2$ with $C(E) = c_1$
  - Merge $c_1$ and $c_2$ in the dependency Graph

Back-edges in CFG

Dependency graph
Finding C-Optimal PS (2)

- Edge Deletion
  - Remove (c1, c2) from the dependency graph
Finding C-Optimal PS (3)

• **while** some $C(E) = \text{null}$, **until** a fix-point
  • **for** every existing counter $c$
    – $C(E) := c$
    – **if** finding bounds succeeds
      • break
  • **if** finding bounds fails
    – $C(E) := \text{fresh counter } c'$
    – $c'$ depends on every other counter and $R$
    – **if** finding bounds fails
      • Postpone decision for $E$
    – **while** finding bounds succeeds
      • Try to delete every edge $(c, c')$
Intermediate Summary

• Having a procedure P
  • In $O(N^2)$ time ($N =$ number of back-edges)
    – Create a C-Optimal PS
    – Or say that a PS does not exist
• Compute the UpperBound
Handling of Complicated Data Structures

• We want to
  • support trees, lists, bit-vectors, etc.
  • avoid analyzing heap shapes

• Solution
  • Introduce uninterpreted quantitative functions
    – Described symbolically by the user
    – Processed by the generator

• Examples
  – Length of a list
  – Height of a tree
Quantitative Functions for Lists

- We define functions $\text{Pos}(e : \text{Item}, L : \text{List})$ and $\text{Len}(L : \text{List})$
- We instrument each use of Lists's methods with
  - $e := \text{L.Head}()$
    - Assume($e = \text{null}$, $\text{implies}$ $\text{Len}(L) = 0$)
    - Assume($e <> \text{null}$, $\text{implies}$ $\text{Len}(L) > 0$)
    - $\text{Pos}(e, L) := 0$
  - $t := \text{L.IsEmpty}()$
    - Assume($t = \text{true}$, $\text{implies}$ $\text{Len}(L) = 0$)
    - Assume($t = \text{false}$, $\text{implies}$ $\text{Len}(L) > 0$)
  - $e1 := \text{L.GenNext}(e2)$
    - $\text{Pos}(e1, L) := \text{Pos}(e2, L) + 1$
    - Assume($0 \leq \text{Pos}(e2, L) < \text{Len}(L)$)
Instrumentation Example

- Initial program
  - e := f; while (e <> null) { e := L.GenNext(e) }

- Instrumented program
  - c := 0; e := f;
  - while (e <> null)
    - Old(e) := e
    - e := L.GenNext(e)
    - Pos(e, L) := Pos(Old(e), L) + 1
    - Assume(0 <= Pos(Old(e), L) < Len(L))
    - c++

- Invariant: \( \{ c \leq Pos(e, L) - Pos(f, L) \} \{ Pos(e, L) \leq Len(L) \} \)

- UpperBound = Len(L) – Pos(f, L)
Bound Examples

- **for** (; not L.IsEmpty(); L.RemoveHead())
  - \{c <= Old(Len(L)) – Len(L)\} \{Len(L) >= 0\}
  - UpperBound = Old(Len(L))
- **for** (e := L.Head(); e <> null;)
  - tmp := e;
  - e := L.GetNext();
  - **if** (…)
    - L.remove(tmp);
  - \{c <= Pos(e, L) + Old(Len(L)) – Len(L)\} \{Pos(e,L) <= Len(L)\}
  - UpperBound = Old(Len(L))
Capabilities of the Generator

- Support for uninterpreted function symbols
  - Semantics was encoded explicitly into the program
- Support for aliasing
  - To establish equality of function results
  - To establish influence of a call on one pointer to properties of another
Inter-procedural Analysis

- Simple case: no recursion
- Traverse the call graph in reversed topological order

\[
\begin{align*}
\text{UpperBoundA}(x_1, x_2, \ldots, x_m) \\
\text{proc } A(x_1, x_2, \ldots, x_m) \\
B(z_1, z_2, \ldots, z_n) \\
\text{UpperBoundB}(y_1, y_2, \ldots, y_n) \\
\text{UpperBoundB}(z_1, z_2, \ldots, z_n) \\
\text{proc } B(y_1, y_2, \ldots, y_n)
\end{align*}
\]
Expressing Arguments

- Problem:
  - having an expression $z$
  - represent it's value as a function of $(x_1, ..., x_m)$

- Solution
  - Compute invariant for $z$ (a bound)
  - Replace $z$ with this bound in $\text{UpperBoundB}(z)$

- Example
  - $\text{proc } B(y_1, y_2)$
    - $\text{UpperBoundB}(y_1, y_2) = y_1 - 2y_2$
  - $\text{proc } A(x_1, x_2)$
    - $\{z_1 - z_2 \leq x_1\} \{z_2 \geq x_2\}$
    - $B(z_1, z_2)$
      - $\exists z_1, z_2 \left[ (U \leq z_1 - 2z_2) \land (z_1 - z_2 \leq x_1) \land (z_2 \geq x_2) \right]$
Recursion

- Divide the call graph into SCCs
  - Process one SCC at a time
  - In reversed topological order
- Inside an SCC
  - There's an entry-point
  - Counters are now global
- We instrument
  - Loop back-edges
  - Positions immediately before recursive calls
- If a counter depends on R it is initialized at entry-point
Example

- **globals**
  - // Initial values of arguments
    - T' : Tree
    - e' : Node
  - // Counters
    - c : Int          // { c <= 2 * (1 + Nodes(e' T') – Nodes(e, T)) } => { x <= 2*(1 + Nodes(e', T')) }
- // Entry-point procedure
- **proc** Traverse'(T : Tree, e : Node)
  - c := 0
  - T' := T
  - e' := e
  - Traverse(T, e)
- **proc** Traverse(T : Tree, e : Node)
  - y := e
  - **while** (y <> null)
    - r := T.GetRight(y)
    - c++
    - Traverse(T, r)
    - y := T.GetLeft(y)
    - c++
Linear Invariant Generator

Summary

• Required features
  • For complex data structures
    – Uninterpreted functions
    – Aliasing
  • For recursion
    – Inter-procedural reasoning
    – Global variables
Overall Summary

- We can compute upper bounds for
  - Loops
  - With complex data structures, such as lists or trees
    - Requires annotations
  - Inter-procedural dependencies
  - Recursion
References

• Sumit Gulwani, Krishna K. Mehra, Trishul Chilimbi (Microsoft Research)
  **SPEED: precise and efficient static estimation of program computational complexity**