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 $P(in_1, ..., in_n)$
- Find a total function $T(in_1, ..., in_n)$: Nat
  - NB: **not** $T(|in_1|, ..., |in_n|)$
- 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 := input()
- y := input()
- while x * y <> 0 do
  - if x > y then
    - x := x mod y
  - else
    - y := y mod x
- WriteLn(x + y)

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

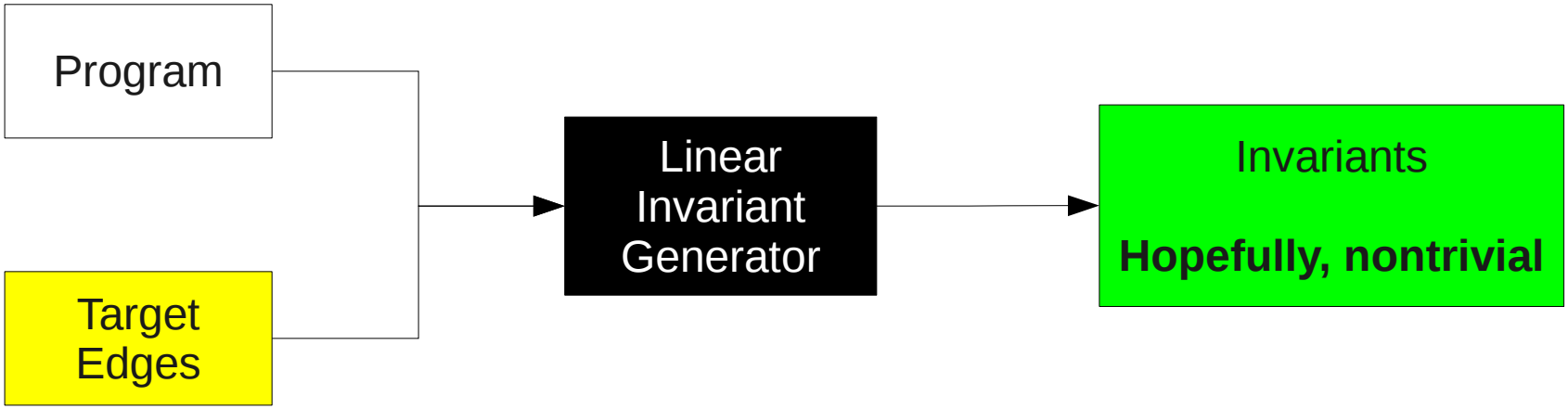
Invariant

- **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 \/ 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



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}**

- **Instrument** back-edges with counters
- Compute linear invariants for counters
- Combine results to obtain a complexity bound

O(Max(0, m – y0) + 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

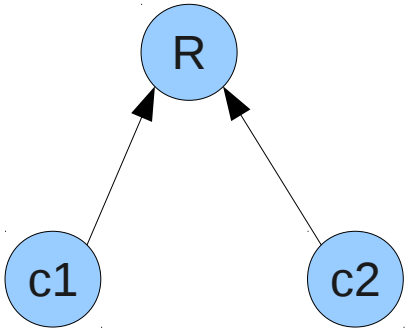
Example of a Proof Structure

- **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;

c1++;

c2++;

Dependency graph:



Computing Bounds

- Total bound for a counter:

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

  - 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:

$$UpperBound = \sum_{c} TotalBound(c)$$

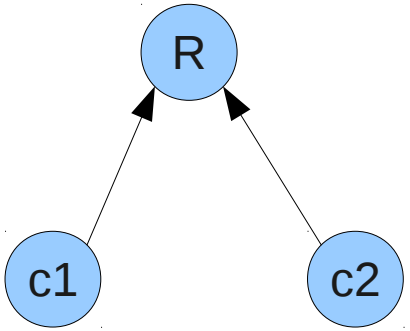
Example of Bound Calculation

- **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;

c1++;

c2++;

Dependency graph:



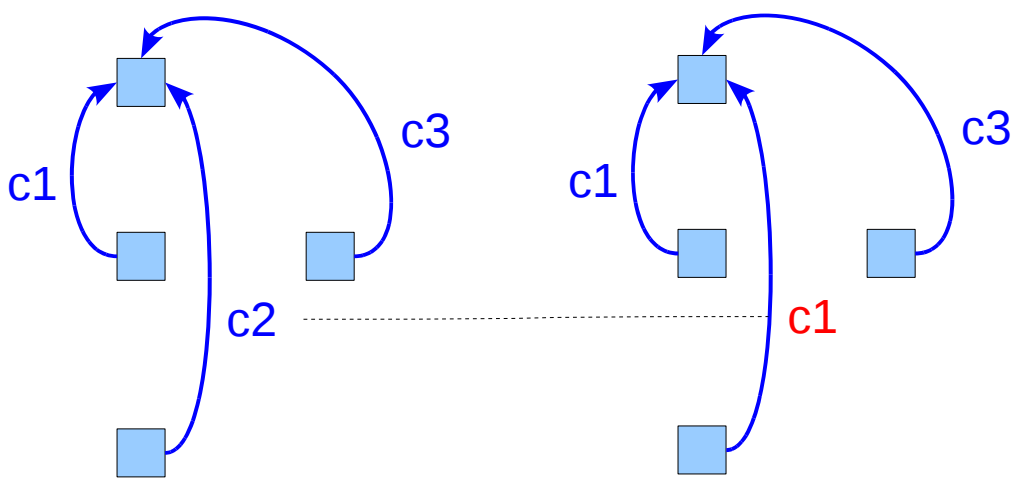
TotalBound(c1) = Max(0, y0 – m) * 1
TotalBound(c2) = Max(0, x0 – n) * 1
UpperBound = Max(0, y0 – m) + Max(0, x0 – n)

Optimality of Proof Structures

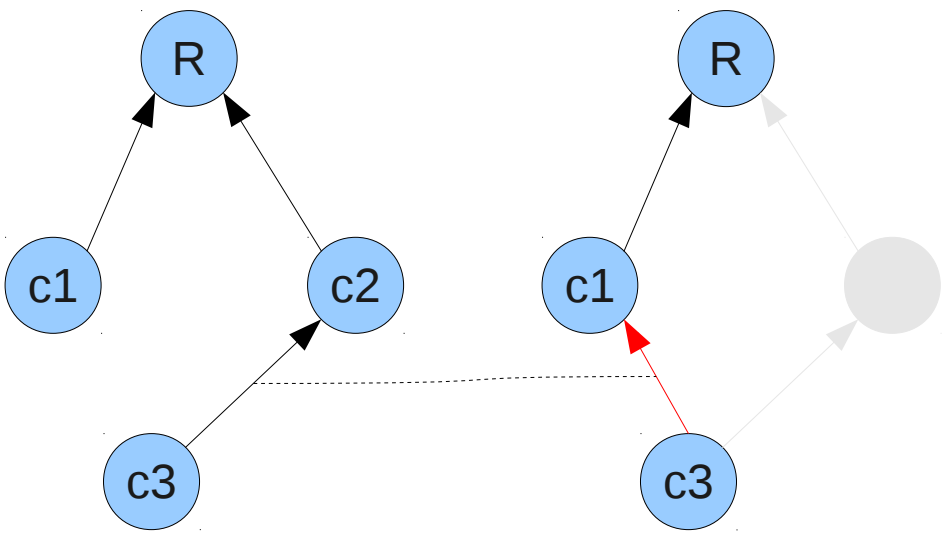
- Optimal PS: A PS that gives UpperBound not greater than any other PS for the same P
  - Hard to compare
  - Hard to construct
- C-Optimal PS (Counter-Optimal)
  - Less counters (no node merge is possible)
  - Less dependencies (no edge deletion is possible)
  - The generator must be able to find bounds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

Finding C-Optimal PS (1)

- Node Merging
  - Let c1 and c2 not depend on each other (transitively)
  - Replace C(E) = c2 with C(E) = c1
  - Merge c1 and c2 in the dependency Graph



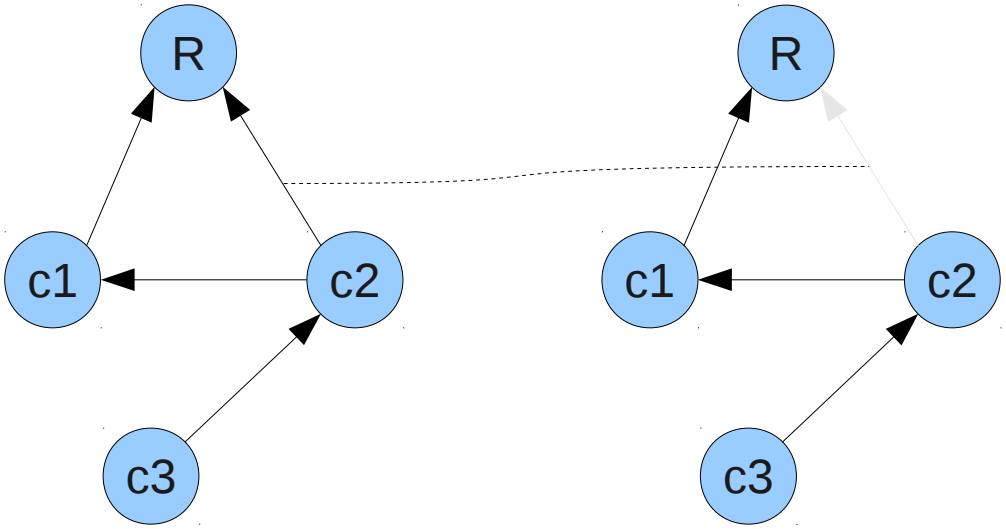
Back-edges in CFG



Dependency graph

Finding C-Optimal PS (2)

- Edge Deletion
  - Remove (c1, c2) from the dependency graph



Dependency graph

Finding C-Optimal PS (3)

- **while** some C(E) = null, **until** a fix-point
  - **for** every existing counter c
    - C(E) := c
    - **if** finding bounds succeeds
      - break
  - **if** finding bounds fails
    - C(E) := 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 Pos(e : Item, L : List) and Len(L : List)
- We instrument each use of Lists's methods with
  - e := L.Head()
    - Assume(e = null **implies** Len(L) = 0)
    - Assume(e <> null **implies** Len(L) > 0)
    - Pos(e, L) := 0
  - t := L.IsEmpty()
    - Assume(t = true **implies** Len(L) = 0)
    - Assume(t = false **implies** Len(L) > 0)
  - e1 := L.GenNext(e2)
    - Pos(e1, L) := Pos(e2, L) + 1
    - Assume(0 <= Pos(e2, L) < 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 <= Pos(e, L) – Pos(f, L) } { Pos(e, L) <= Len(L) }**
- **UpperBound** = Len(L) – Pos(f, L)

Bound Examples

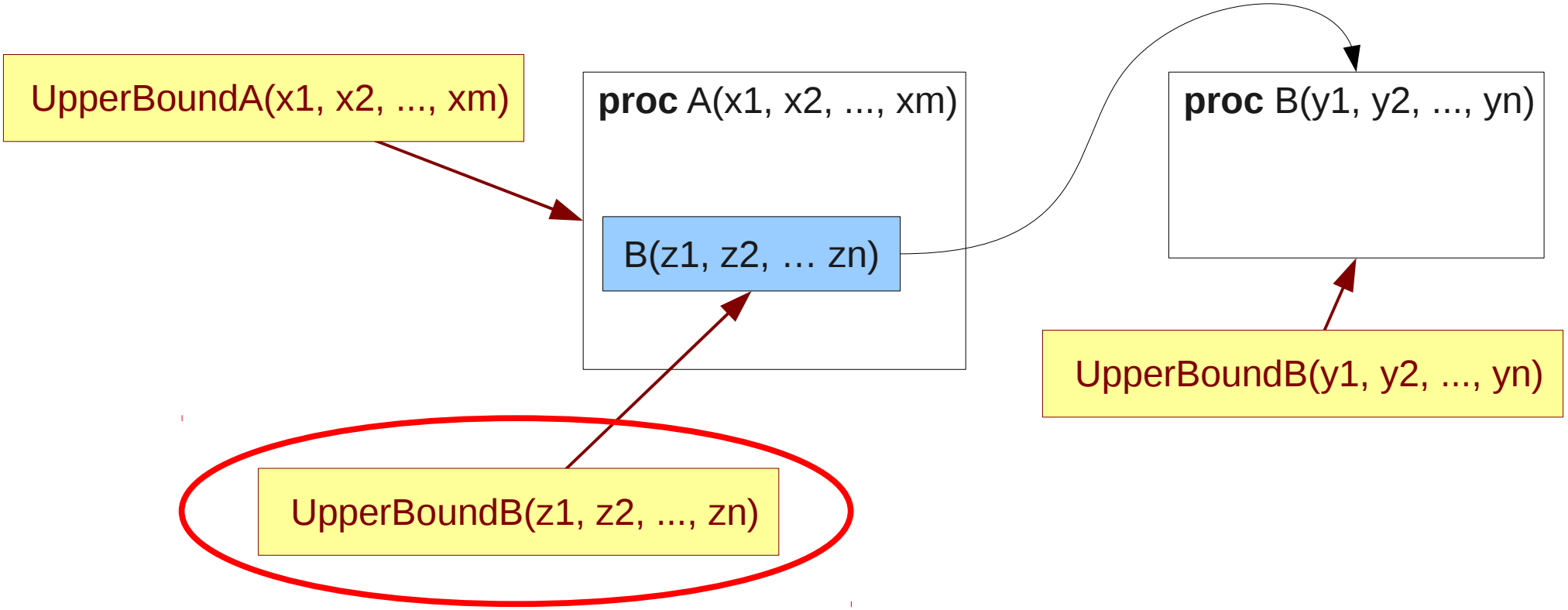
- **for** (; not L.IsEmpty(); L.RemoveHead())
  - {c <= Old(Len(L)) – Len(L)} {Len(L) >= 0}
  - UpperBound = Old(Len(L))
- **for** (e := L.Head(); e <> null;)
  - tmp := e;
  - e := L.GetNext();
  - **if** (...)
    - L.remove(tmp);
  - {c <= Pos(e, L) + Old(Len(L)) – Len(L)} {Pos(e,L) <= Len(L)}
  -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



Expressing Arguments

- Problem:
  - having an expression z
  - represent it's value as a function of (x1, ..., xm)
- Solution
  - Compute invariant for z (a bound)
  - Replace z with this bound in UpperBoundB(z)
- Example
  - **proc** B(y1, y2)
    - UpperBoundB(y1, y2) = y1 - 2*y2
  - **proc** A(x1, x2)
    - {z1 – z2 <= x1} {z2 >= x2}
    - B(z1, z2)
      - UpperBoundB(z1, z2) <= x1 - x2

$$\exists z_1, z_2 \left[(U \leq z_1 - 2 \cdot z_2) \wedge (z_1 - z_2 \leq x_1) \wedge (z_2 \geq x2) \right]$$

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
  - http://portal.acm.org/citation.cfm?doid=1480881.1480898