Constraint-based Invariant Inference over Predicate Abstraction

Saurabh Srivastava

University of Maryland, College Park

Sumit Gulwani
Ramarathnam Venkatesan

Microsoft Research, Redmond

Introduction

- Last decade has seen an engineering revolution in SAT solving.
- Can we bring the technology to program analysis?
  - This talk shows how to do that for predicate abstraction by using off-the-shelf SAT solvers, e.g. Z3
- We have developed constraint-based techniques:
  - Program Verification
  - Maximally-weak Precondition Inference
  - Inter-procedural Summary Computation
  - Inferring the Maximally-general Counterexamples to Safety (i.e. finding the best descriptions of bugs).

Introduction

- Last decade has seen an engineering revolution in SAT solving.

- Can we bring the technology to program analysis?
  - This talk shows how to do that for predicate abstraction by using off-the-shelf SAT solvers, e.g. Z3

- We have developed constraint-based techniques:
  - Program Verification
  - Maximally-weak Precondition Inference
  - Inter-procedural Summary Computation
  - Inferring the Maximally-general Counterexamples to Safety (i.e. finding the best descriptions of bugs).

Predicate Abstraction

- Given a fixed finite set of n predicates, associate with each predicate p_i a **boolean indicator** b_i.

- Sound over-approximation of the invariant at each program point represented by a boolean expression involving the indicators.

Boolean expression

$$\gamma\,(exp(b_1,\ldots,b_n)) = exp[p_1/b_1,\ \ldots,\ p_n/b_n]$$

$$\alpha\,(\psi) = \bigwedge\{exp(b_1,\ldots,b_n) \mid \psi \Rightarrow exp[p_1/b_1,\ \ldots,\ p_n/b_n]\}$$

$\alpha(\psi)$ in general, not computable

$$\alpha'\,(\psi) = \bigwedge_{i=1\ldots n}\{b_i \mid \psi \Rightarrow p_i\}$$

Constraint-based Invariant Inference

- Guess a DNF template: k disjuncts

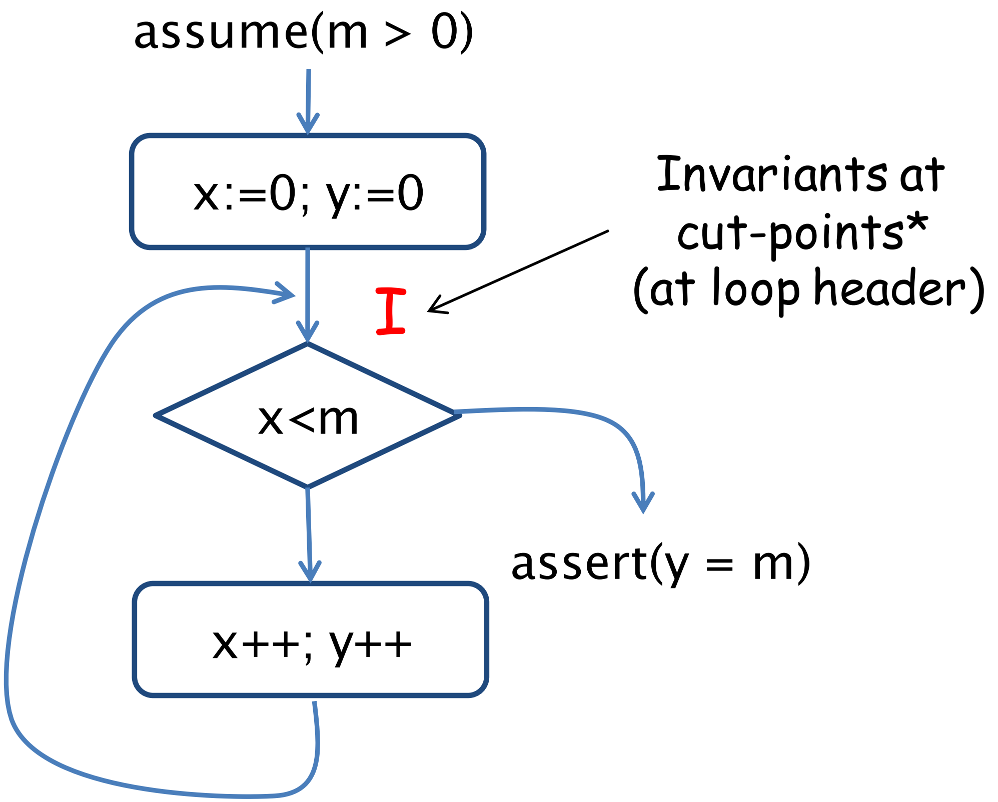
  $(\ldots)\vee(\ldots)\vee(\ldots)$: k=3

- Task: Fill out each disjunct with a boolean monomial (conjunction of indicator literals)

- Approach: Generate boolean constraints over indicators using the program semantics and directly solve using off-the-shelf solvers.

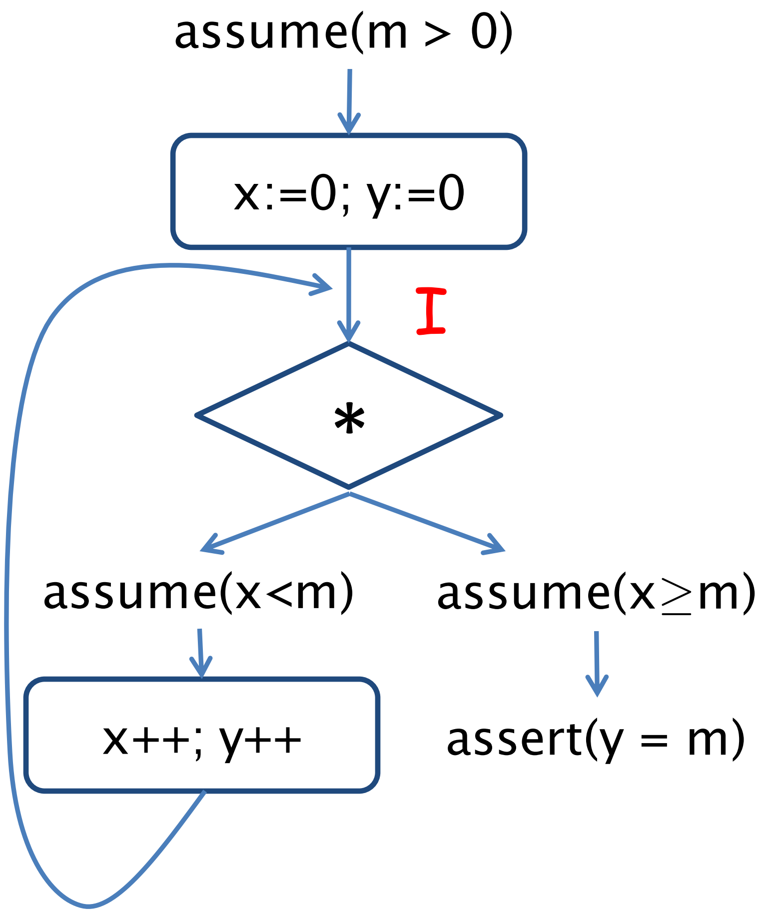
Example: Cut-points

```
loop (int m) {
  assume(m > 0)
  x:=0; y:=0;

  while (x<m) {
     x++; y++;
  }

  assume(y = m)
}
```

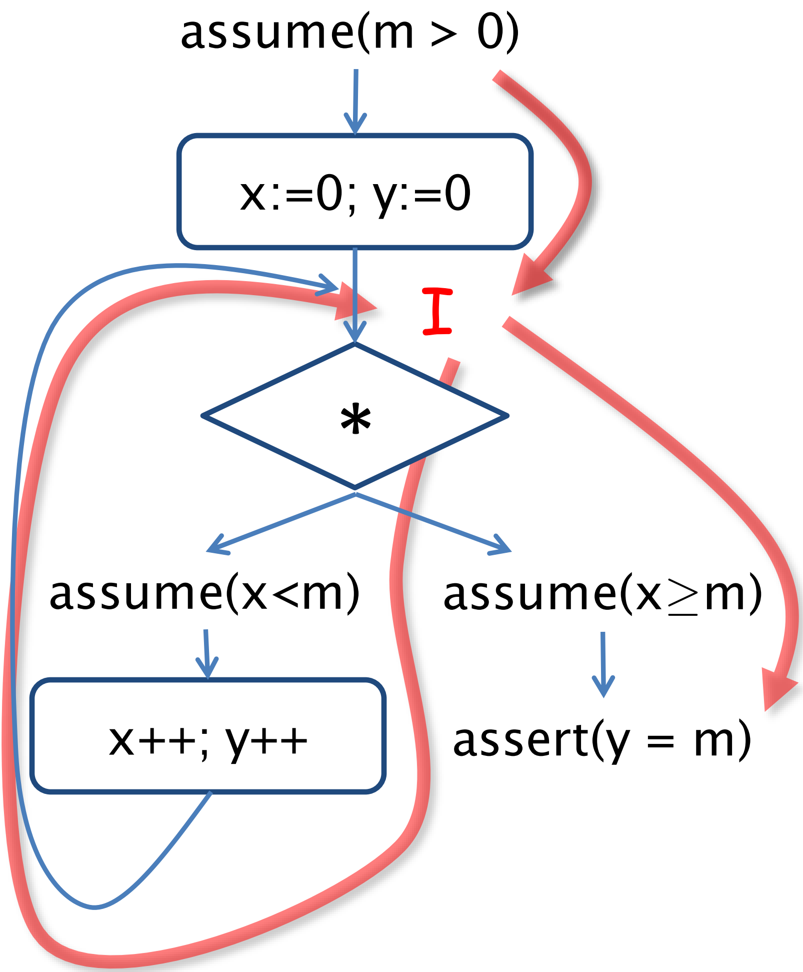


*Cut-set: Set of cut-points such that each cycle in CFG passes through at least one cut-point

Example: Simple paths and VCs



Example: Simple paths and VCs



- Verification condition induced by each simple path (sequence of stmt)
- VC computed using standard backwards weakest precondition operator ω:

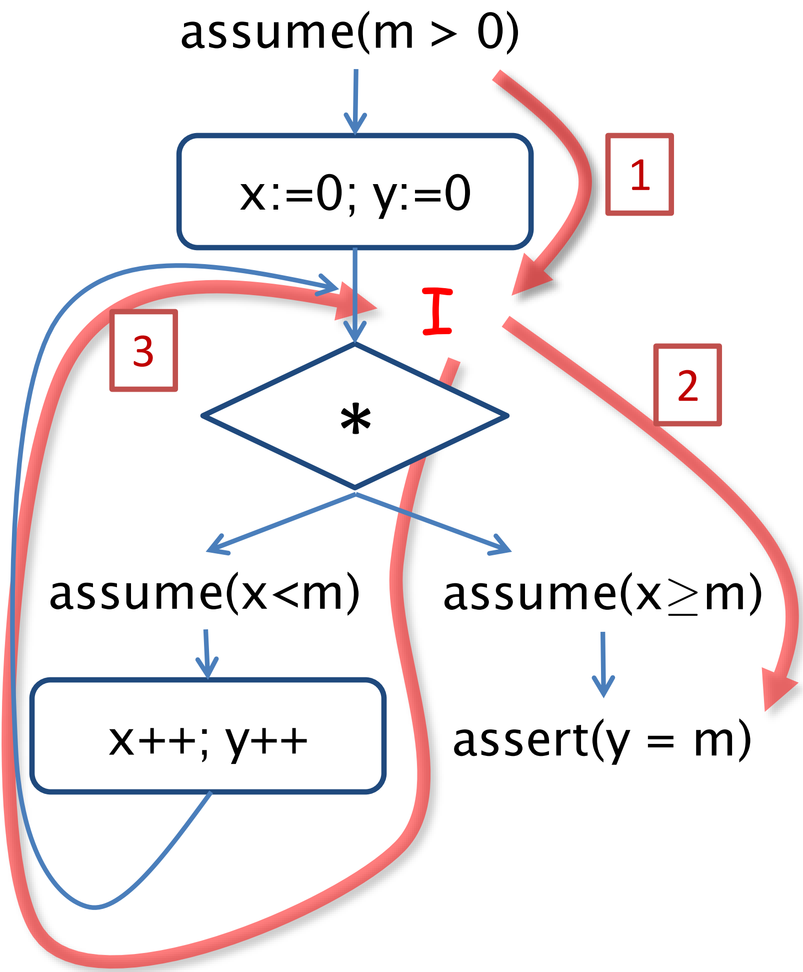
$$\omega(x:=e, \phi) = \phi\,[e/x]$$

$$\omega(\text{assume}(p), \phi) = p \Rightarrow \phi$$

$$\omega(\text{assert}(p), \phi) = p \wedge \phi$$

$$\omega(\tau_1;\tau_2, \phi) = \omega(\tau_1, \omega(\tau_2, \phi))$$

Example: Simple paths and VCs



1. $m>0 \Rightarrow I[y \rightarrow 0, x \rightarrow 0]$
2. $I \wedge x \geq m \Rightarrow y=m$
3. $I \wedge x<m \Rightarrow I[y \rightarrow y+1, x \rightarrow x+1]$

$\omega(x:=e, \phi) = \phi\,[e/x]$

$\omega(\text{assume}(p), \phi) = p \Rightarrow \phi$

$\omega(\text{assert}(p), \phi) = p \wedge \phi$

$\omega(\tau_1;\tau_2, \phi) = \omega(\tau_1, \omega(\tau_2, \phi))$

Example: Boolean Constraint Generation

Unknown invariant on the LHS; constrains how weak I can be

Unknown invariant on the RHS; constrains how strong I can be

1. $m>0 \Rightarrow I[y \rightarrow 0, x \rightarrow 0]$
2. $I \wedge x \geq m \Rightarrow y=m$
3. $I \wedge x<m \Rightarrow I[y \rightarrow y+1, x \rightarrow x+1]$

Unknown on both sides; combination of above cases

Example: Boolean Constraint Generation

Unknown invariant on the LHS; constrains how weak I can be

[2] $I \wedge x \geq m \Rightarrow y=m$

Unknown invariant on the RHS; constrains how strong I can be

[1] $m>0 \Rightarrow I[y \rightarrow 0,\ x \rightarrow 0]$

1: $y \leq m\ \wedge y \geq m$

2: $x<m$

3: $x \leq y \wedge y \leq m$

$x \leq y,\ \ x \geq y,\ \ x<y$

$x \leq m,\ x \geq m,\ x<m$

$y \leq m,\ y \geq m,\ y<m$

$0 \leq 0,\ \ 0 \geq 0,\ \ 0<0$

$0 \leq m,\ 0 \geq m,\ 0<m$

$0 \leq m,\ 0 \geq m,\ 0<m$

Maximally-weak ways of satisfying the constraint using the given preds. (Computed using Predicate Cover)

$\neg \left(x<y,\ x \geq m,\ y \geq m \right)$

$(b_{x<m}) \vee (b_{y \leq m} \wedge b_{y \geq m}) \vee (b_{x \leq y} \wedge b_{y \leq m})$

$\neg\, b_{x \geq m} \wedge \neg\, b_{x<y} \wedge \neg\, b_{y \geq m}$

Example: Solving using SAT

$\neg b_{x \geq m} \wedge \neg b_{x<y} \wedge \neg b_{y \geq m}$

$(b_{x<m}) \vee (b_{y \leq m} \wedge b_{y \geq m}) \vee (b_{x \leq y} \wedge b_{y \leq m})$

$(b_{y \leq m} \Rightarrow (b_{y<m} \vee b_{y \leq x})) \wedge \neg b_{x<m} \wedge \neg b_{y<m}$

Individual *local* computations

SAT Solver (fixed point computation)

tt: $b_{y \leq x}$; $b_{y \leq m}$; $b_{x \leq y}$

ff: rest

I: $y=x \wedge y \leq m$

```
loop (int m) {
  assume(m > 0)
  x:=0; y:=0;
  while (x<m) {
     x++; y++;
  }
  assert(y = m)
}
```

- ✓ Program Verification
- ➢ Maximally-weak Precondition Inference
- ➢ Inter-procedural Summary Computation

Maximally-weak preconditions

- Instead of the precondition **true** as in PV, treat precondition as an unknown PRE
- Generate constraints as for PV—now in terms of PRE and the unknowns invariant I's
- Solving these yields *a* precondition PRE, but not necessarily the maximally-weakest
- Iteratively, improve the current precondition T by adding the following constraint:

$$T \Rightarrow PRE \wedge \neg(PRE \Rightarrow T)$$

Context-sensitive Inter-procedural Analysis

- Compute context-sensitive procedure summaries as (A_i, B_i) pre/post pairs in assume-guarantee style reasoning
- Constraint generation
  - Procedure body (guarantee):

    assume(A_i); S; assert(B_i) P(x) { S; return y; }

  - Calls (assume):

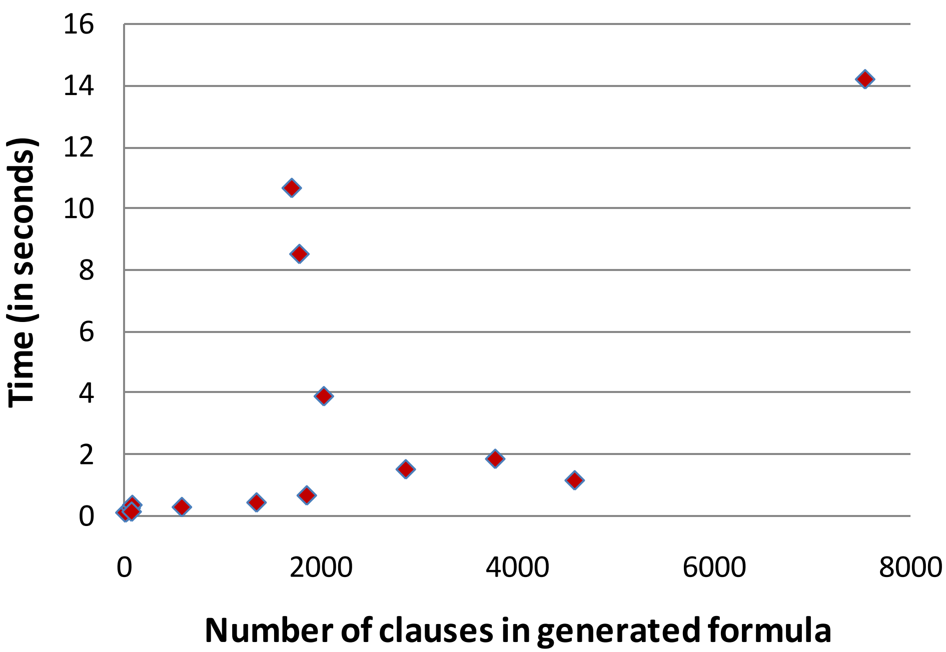
    assert(A_i[u/x]); assume(B_i[u/x, t/y]); v:=t v = P(u);

Experiments: Overview

- Our benchmarks are academic/small benchmark programs that demonstrate the feasibility of the technique
- We ran our tool in two modes: program verification and weakest precondition
- We are able to easily generate disjunctive invariants for which specialized techniques have been proposed earlier
- We collected three performance statistics:
  - Time for verification condition generation (weakest precondition over simple paths)
  - Time for boolean constraint generation (includes the predicate cover operation)
  - Time for SAT solving (fixed point computation)

Experiments: Results

- VC generation: 0.23sec
- SAT solving: 0.06sec
- Boolean formula generation:



- Overall time for invariant generation is low
- Predicate cover called on small formulas. Our unoptimised version performs reasonably

Related Work

- Constraint-based invariant inference:
  - Cousot / Sankarnarayanan et.al.: LIA using mathematical solvers
  - Beyer et.al.: LIA+UFS by compiling away UFS to LIA
  - Podelski et.al. / Bradley et.al.: Discover ranking functions

  We describe a reduction over the very successful domain of predicate abstraction

- Application of SAT to program analysis:
  - SATURN: bit accurate modeling of loop-free programs with complicated data structures
  - Bounded model checking etc.

  Use SAT for validation; in contrast, we use it for inference of invariants that are sound over-approximations

Conclusions

- Constraint-based techniques offer two advantages over iterative fixed-point techniques:
  - Goal directed (may buy efficiency)
  - Do not require widening (may buy precision)
- For predicate abstraction, we have shown how to reduce various program analysis problems to constraint solving.
- In addition to program verification, constraint-based encoding facilitates easy extensions to inter-procedural summary computation, maximally-weak preconditions, counter-examples to safety.

Future Work

- We are exploring extensions to quantifiers and other analysis problems as future work.

- We are exploring the scalability of this technique along two directions:
  - Encodings that yield simpler SAT instances, e.g. exploiting symmetry information for the case of disjunctive solutions
  - Reducing programmer burden by automatically inferring predicate sets and templates

- VS^3: Verification and Synthesis using SMT Solvers
  http://www.cs.umd.edu/~saurabhs/pacs/

Questions?

Best Description of Bugs

- Instrument

$x<m \wedge y \geq x$

```
Err (int m) {
   while (x<m) {
       x++; y++;
       assert (y<m);
   }
}
```

$(x<m \wedge y \geq x) \vee (error=1 \wedge y \geq x)$

- Run maximally-weak precondition