Program Verification

Plus: “Program Verification with Probabilistic Inference” by Sumit Gulwani and Nebojsa Jojic

What is Program Verification?

- Simple idea: Prove that a program behaves correctly, given some specification
- What kinds of specifications?

  Invariants: precondition and postcondition

- **Hoare Triple** – {A} P {B}

What is Program Verification?

- Simple idea: Prove that a program behaves correctly, given some specification
- What kinds of specifications?

  Invariants: precondition and postcondition

- **Hoare Triple** – {A} P {B}

What is Program Verification?

- Simple idea: Prove that a program behaves correctly, given some specification
- What kinds of specifications?

  Invariants: precondition and postcondition

- **Hoare Triple** – {A} P {B}

What is Program Verification?

- Simple idea: Prove that a program behaves correctly, given some specification
- What kinds of specifications?

  Invariants: precondition and postcondition

- **Hoare Triple** – {A} P {B}

Invariants and Program State

- Precondition and postcondition are special cases of program **invariants**, denoted φ
- A **program state** σ is a mapping of variables in the program to values
- Invariants **restrict the set of valid program states** at a specific point in execution

$\sigma \vDash \varphi$ means "σ is a valid state given φ" or "σ satisfies φ"

Validity of Hoare Triple

- **A program is correct w.r.t. invariants if the Hoare triple is valid**

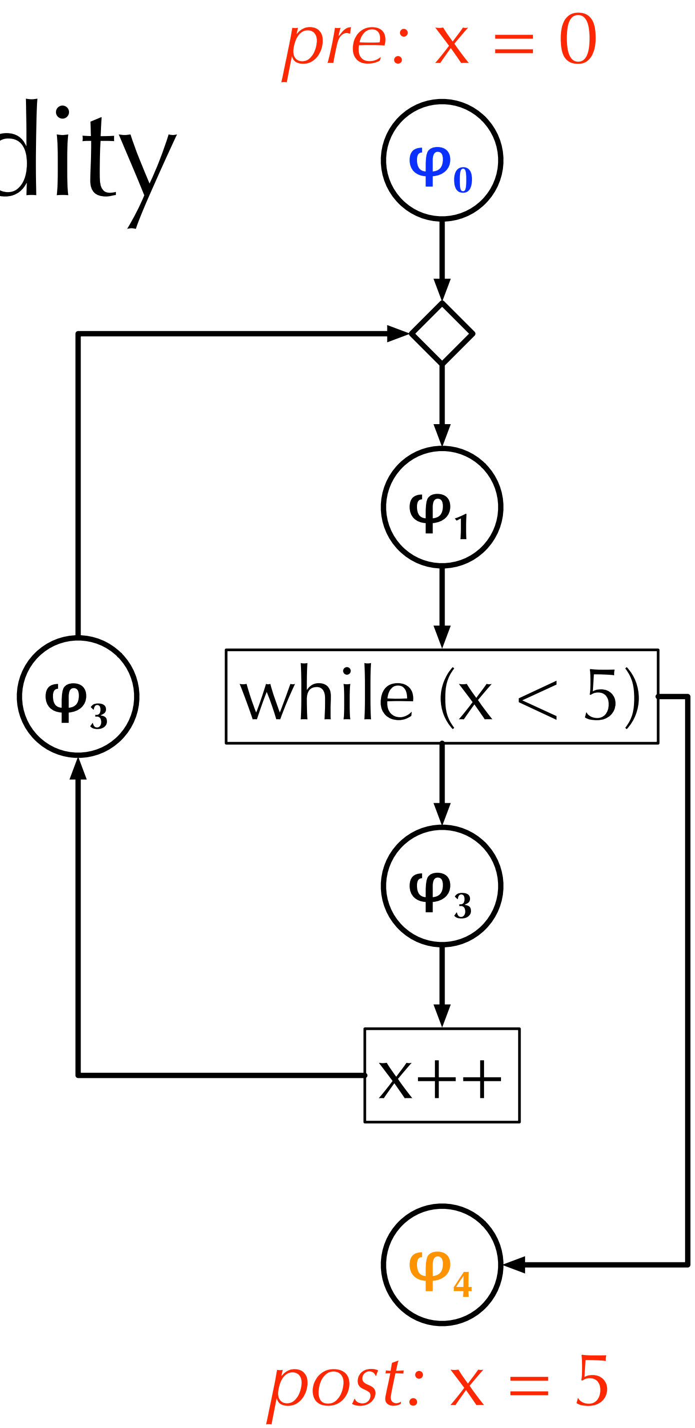
$$\models \{A\}\ P\ \{B\}$$

"For all σ, if $\sigma \models A$ then σ' is the state after executing P, and $\sigma' \models B$."

- Not feasible to look at every state. Can we prove this another way?

Proof of Validity

- Find φs such that each individual statement + invariants forms a valid Hoare triple
- Require *pre* ⇒ φ_0 and φ_4 ⇒ *post*
- How can we find these invariants?
- One option: backwards analysis "pushes" invariants backwards past statements.



Pushing Invariants
(the first one's always free)

- Given postcondition(s) for a statement **s**, find an invariant s.t. all states satisfying the invariant prior **s** satisfy the postcondition(s) after **s**
  - Many possible invariants (*e.g.* **false** trivially suffices). Choose the *weakest* one

  What does it mean for an invariant to be "**weak**" or "**strong**"?

Strengths and Weaknesses

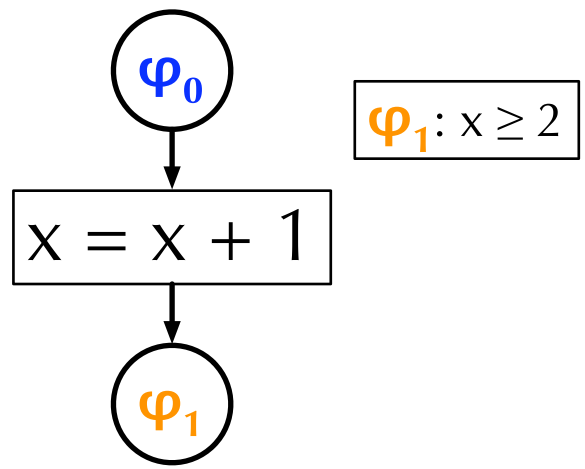
- φ' is weaker than φ if $\varphi \Rightarrow \varphi'$
- What does $\varphi \Rightarrow \varphi'$ mean?
  - For all $\sigma \vDash \varphi$, $\sigma \vDash \varphi'$
  - Matches our natural understanding of $\Rightarrow$
- Intuition: **the more valid program states an invariant allows, the weaker it is.**

Example

- φ_0 must be chosen so φ_1 is valid after assignment
- Many options (*e.g.* {x ≥ 3}). We choose the one which has the most valid states:

$$\varphi_0: x \geq 1$$

- Note, for all other φ that work, $\varphi \Rightarrow \varphi_0$
- We call φ_0 the "weakest precondition"



Backwards analysis

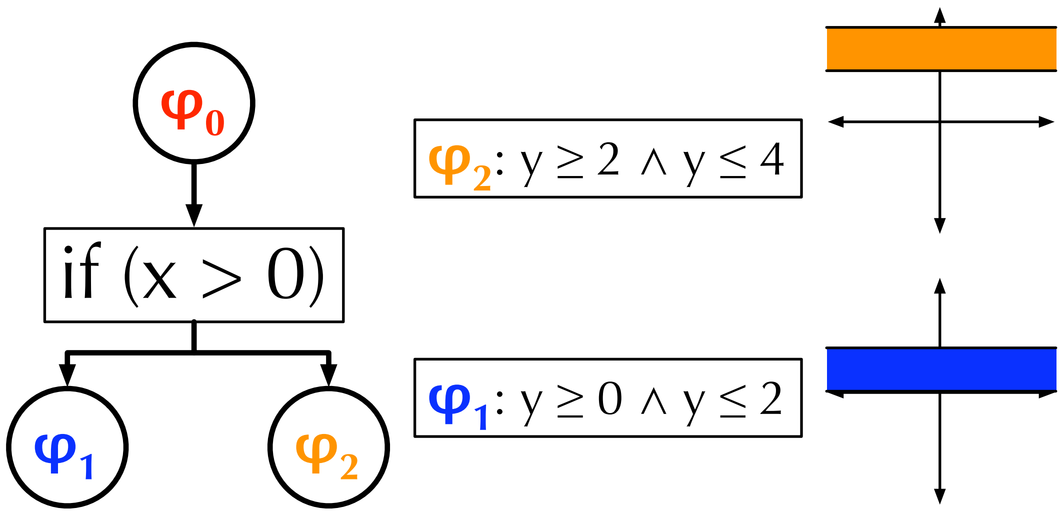
- Initialize all invariants to **true**
- Push invariants back until convergence
- Produces φ_0 at beginning of program. Must prove that precondition $\Rightarrow \varphi_0$
  - This is undecidable! (Thanks, Gödel...)
  - Solution: restrict domain of invariants

Underapproximation

- When domain of invariants is restricted, we must **underapproximate** invariant
  - Precondition we *want* may not be expressible in domain
  - We must choose stronger invariant (*i.e.* fewer valid states)
- This may preclude finding proof
  - Precondition may not imply φ_0

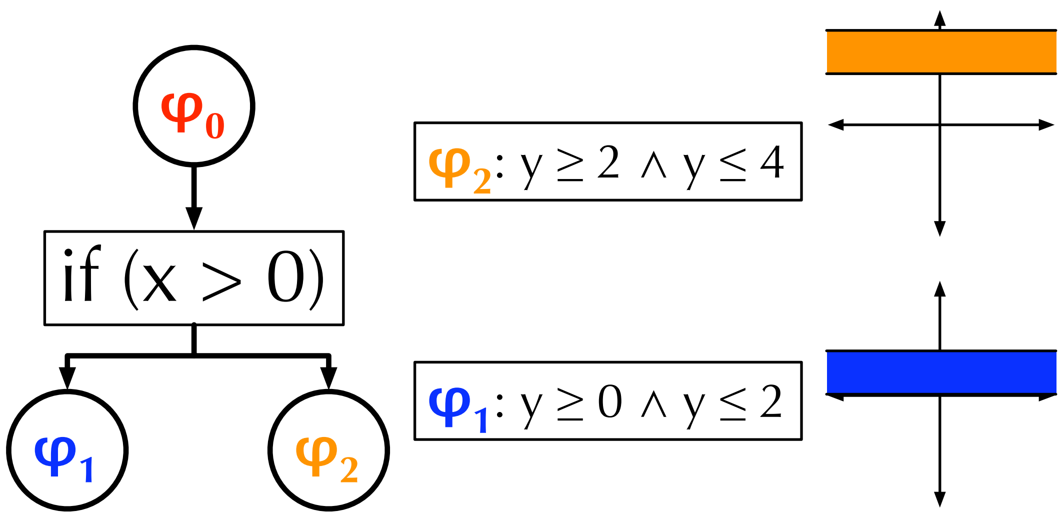
Example

- Domain: conjunctions of inequalities (*i.e.* convex polyhedra)



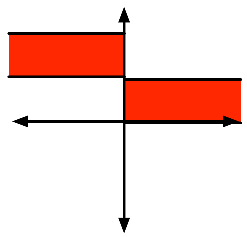
Example

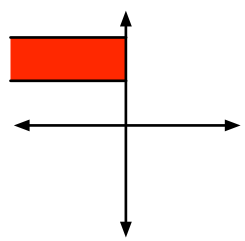
- Weakest φ_0: $((x > 0) \wedge \varphi_1) \vee (x \leq 0) \wedge \varphi_2))$



Example

φ_0: $(x > 0 \wedge y \geq 0 \wedge y \leq 2) \vee (x \leq 0 \wedge y \geq 2 \wedge y \leq 4)$



- This can't be expressed in abstract domain! Must choose different invariant
- Underapproximation  sound, but loses precision
- Some valid preconditions can't be verified
  - e.g. $\{x = 1 \wedge y = 1\}$

Wrapping up

- Similar procedure for forward analysis
  - Initialize to **false**, push forward using **strongest postcondition**
  - Show that final $\varphi \Rightarrow$ program's postcondition
  - May **overapproximate**
- Analysis produces **correctness proof**: $\vdash \{A\}\ P\ \{B\}$
- This is sound, but not complete:

$$\vdash \{A\}\ P\ \{B\} \Rightarrow\ \models \{A\}\ P\ \{B\}$$

On to the Paper!

Program Verification: Rethought

- Recall: a program is verified when a proof is found establishing the postconditions given the preconditions
- This is a global condition
- Alternate formulation: a proof is valid when all φs are **locally consistent**

Local Consistency

- Consider a program point π_k
- Weakest precondition of successors: pre(π_k)
- Strongest postcondition of predecessors: post(π_k)
  - Define pre(π_{exit}) to be postcondition of program and post(π_{entry}) to be its precondition.
- φ_k is **locally consistent** when:

$$\text{post}(\pi_k) \Rightarrow \varphi_k \wedge \varphi_k \Rightarrow \text{pre}(\pi_k)$$

Main Idea of Paper

- Randomly choose φs until all are locally consistent!
  - Deciding if φ is locally consistent does not require global knowledge
  - But may take unbounded time
- Apply **probabilistic inference** to converge on φs faster!

Quick Detour:
Need to Climb a Hill

Probabilistic Inference

- Given a **probability density function** (pdf) of K variables:

$$p(x_1, x_2, \ldots, x_K)$$

Can we find values for all x_is such that p is maximized?

Gibbs Sampling

- Pick arbitrary x_i and consider **conditional distribution function** (cdf):

$$p(x_i \mid x_1, \ldots, x_{i-1}, x_{i+1}, \ldots, x_k)$$

- Choose a value for x_i according to probabilities of cdf ("Draw a sample from cdf")
- Choose another x_i and continue
- Will converge to optimal values for variables

Analogy with Hill Climbing

- Classic AI search technique:
  - Pick a variable x_i and change it to improve target function
  - With some (small) probability, choose something other than best value for x_i
    - Avoids local maxima

Now back to your regularly scheduled program verification

Inconsistency Measure

- Define an **inconsistency measure, *M*** for invariants φ and φ'
  - Intuition: The closer φ is to being stronger than φ', the more consistent the two invariants are
  - $M(\varphi, \varphi') = 0$ iff $\varphi \Rightarrow \varphi'$ (no inconsistency)
  - As φ gets stronger, consistency increases
  - As φ' gets stronger, consistency decreases

Local Consistency as a Function

- Local inconsistency for a given φ at program point π_k

$$L(\varphi, \pi_k) = M(\text{post}(\pi_k), \varphi) + M(\varphi, \text{pre}(\pi_k))$$

- Note that when $L(\varphi, \pi_k) = 0$

$$\text{post}(\pi_k) \Rightarrow \varphi \wedge \varphi \Rightarrow \text{pre}(\pi_k)$$

so φ is locally consistent

Verification as Optimization

- Now have a real-valued measure of local consistency at each program point
- Construct function *f*

$$f(\varphi_0, \varphi_1, \ldots \varphi_K)$$

  using $L(\varphi_i, \pi_i)$ such that *f* is maximized when all φs are locally consistent

- Can apply Gibbs sampling to this function!

Operation of algorithm

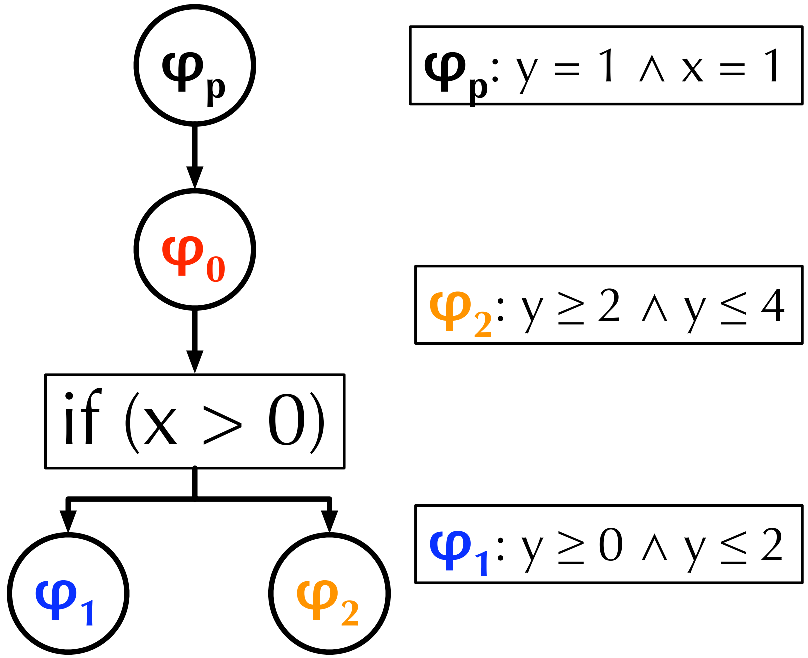
- Initialize all φs to $\perp$
- Pick a random program point π_k whose invariant φ_k is not locally consistent
- Choose φ to minimize inconsistency at π_k
  - But with some probability, choose other φ
- Update $\varphi_k = \varphi$
- Continue until no local inconsistency

Key Algorithm Features

- Only local decisions made at any point
  - Local inconsistency only related to small number of program points
- Uses both forward and backward information
  - *L* involves both predecessors and successors
  - Avoids precision issues of standard analyses

Example, take two

- Consider choosing appropriate invariant for φ_0

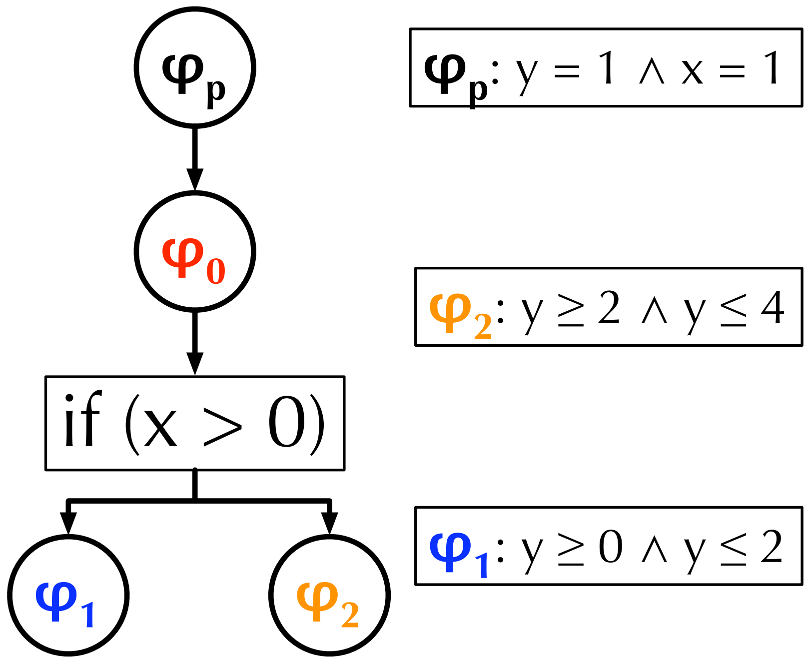


Example, take two

- Consider choosing appropriate invariant for φ_0

  $post(\pi_0) = \varphi_p$

  $pre(\pi_0) = ((x > 0) \wedge \varphi_1) \vee (x \leq 0) \wedge \varphi_2))$



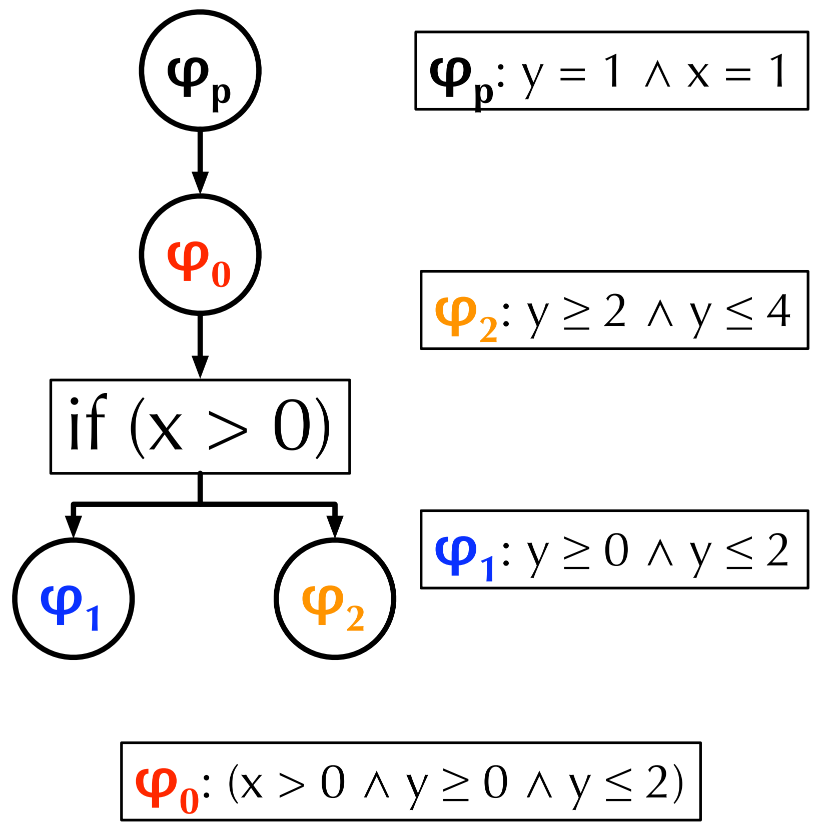
Example, take two

- Consider choosing appropriate invariant for φ_0

  $post(\pi_0) = \varphi_p$

  $pre(\pi_0) = ((x > 0) \wedge \varphi_1) \vee (x \leq 0) \wedge \varphi_2))$

- Desire to minimize inconsistency with both post and pre leads to correct choice of φ_0



Forward + Backward > Standing Still

- Essentially, analysis uses information from predecessors to “guide” its underapproximation (equivalently, uses information from successors to guide overapproximation)
- Produces better results than many existing analyses

Random Choices are Good

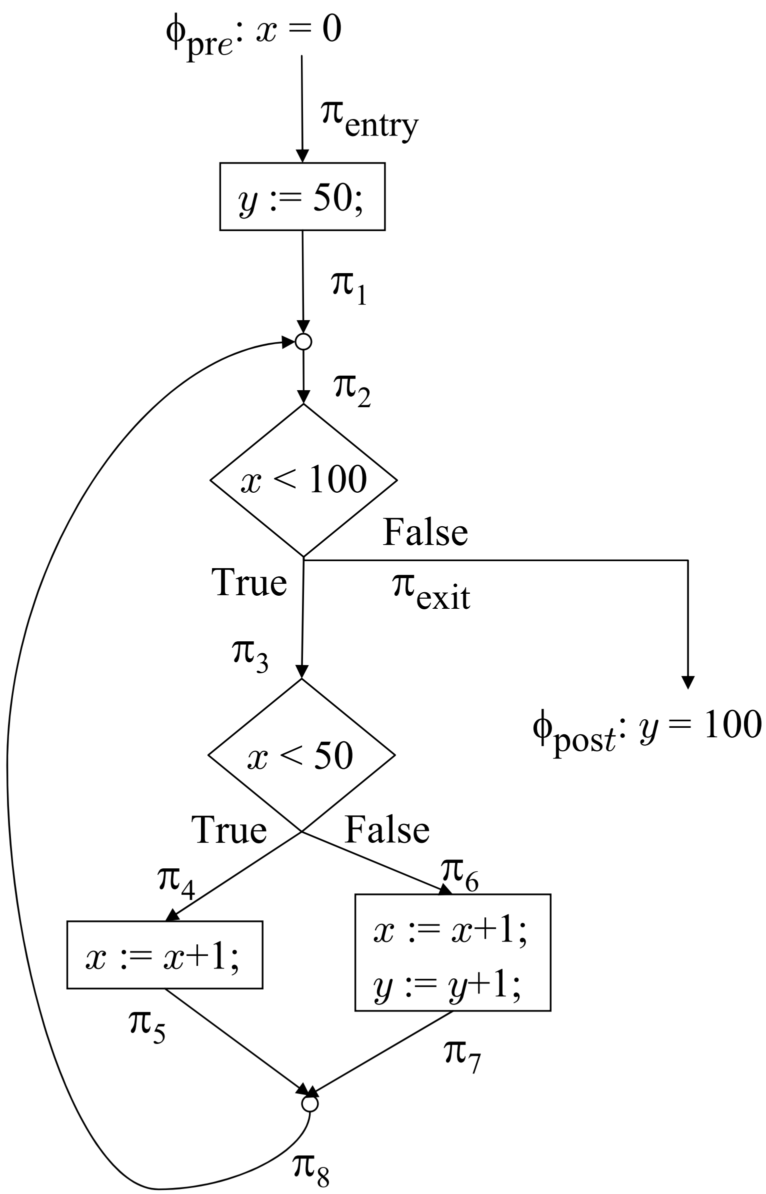
- Random choices
  - Which program point to update: Finding the proper invariants may require very specific sequence of updates. This is almost impossible to determine normally
  - What invariant to use: Given a set of equally inconsistent choices, random selection will eventually choose the right invariant
- Upshot: Randomness leads to proper result when there is no clear strategy

Some Results

- Abstract domain: Boolean combinations of difference constraints with ($m \times n$) template
  - m conjuncts, each with at most n disjuncts
- $M(\varphi, \varphi')$ where φ' is the conjunction of several clauses:

$$\mathcal{M}(\phi, \bigwedge_{i=1}^{m} C_i) = \sum_{i=1}^{m} \frac{1}{m} \times \mathcal{M}(\phi, C_i) \qquad \mathcal{M}(\bigvee_{j=1}^{k} D_j, C_i) = \sum_{j=1}^{k} \frac{1}{k} \times \mathcal{M}(D_j, C_i)$$

Test Program and Proof

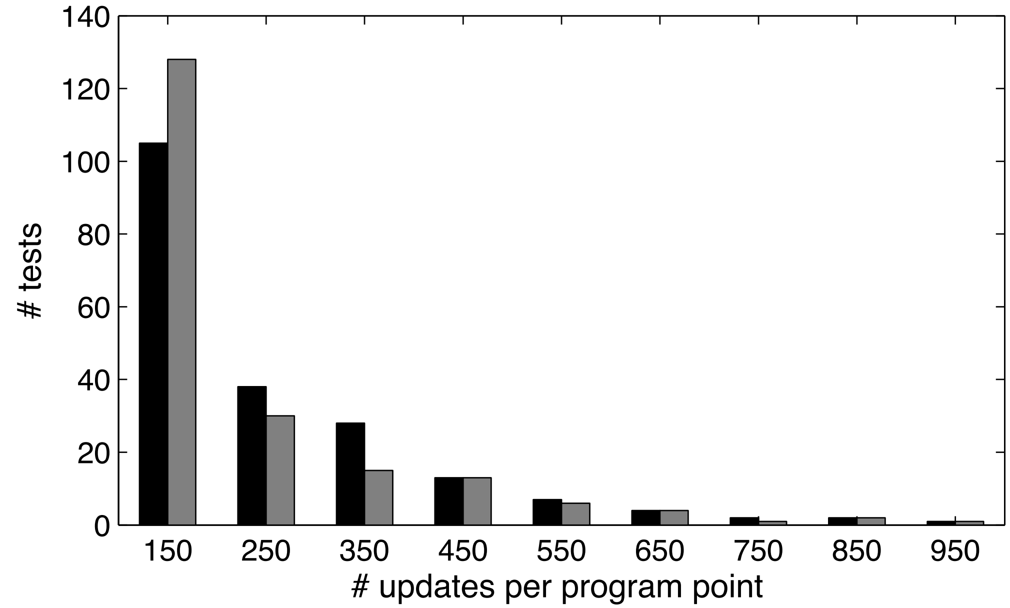


Program Point	Invariant
π_0	$x = 0$
π_1	$(y = 50) \wedge (x = 0)$
π_2	$(y = 50 \vee x \geq 50) \wedge (y = x \vee x < 50) \wedge (y = 100 \vee x < 100)$
π_3	$(y = 50 \vee x \geq 50) \wedge (y = x \vee x < 50) \wedge (y = 99 \vee x < 99)$
π_4	$(y = 50) \wedge (x < 50)$
π_5	$(y = 50) \wedge (x < 51)$
π_6	$(x \geq 50) \wedge (y = x \vee x < 50) \wedge (y = 99 \vee x < 99)$
π_7	$(x > 50) \wedge (y = x \vee x < 51) \wedge (y = 100 \vee x < 100)$
π_8	$(y = 50 \vee x \geq 50) \wedge (y = x \vee x < 50) \wedge (y = 100 \vee x < 100)$
π_9	$y = 100$

Existing techniques unable to verify this program!

How long does it take?

- Performed multiple runs of prover
- Histogram of tests which took a certain number of updates per π
- Black bar: all πs initialized to ⊥
- Gray bar: use previously found proof on slightly modified program



Discussion

- Could there be some benefit to a more directed search? (*e.g.* choosing which program point to update in a more systematic way)
- Is this randomized approach useful in other domains? Can it be applied to any dataflow/abstract interpretation problem?