Proving Programs Correct by Abstract Interpretation

Supratik Chakraborty
IIT Bombay

WEPL (POPL) 2015
Program Analysis: An Example

```c
int x = 0, y = 0, z;
read(z);
while (f(x, z) > 0) {
    if (g(z, y) > 10) {
        x = x + 1; y = y + 100;
    }
    else if (h(z) > 20) {
        if (x >= 4) {
            x = x + 1; y = y + 1;
        }
    }
}
```

IDEAS?
- Run test cases
- Get code analyzed by many people
- Convince yourself by ad-hoc reasoning

What is the relation between x and y on exiting while loop?
int x = 0, y = 0, z;
read(z);

while (f(x, z) > 0) {
    if (g(z, y) > 10) {
        x = x + 1; y = y + 100;
    }
    else if (h(z) > 20) {
        if (x >= 4) {
            x = x + 1; y = y + 1;
        }
    }
}

assert(x < 4 OR y >= 2);
Verification & Analysis: Close Cousins

- Both investigate relations between program variables at different program locations
- Verification: A (seemingly) special case of analysis
  - Yes/No questions
  - No simpler than program analysis
- Both problems undecidable (in general) for languages with loops, integer addition and subtraction
  - Exact algorithm for program analysis/verification that works for all programs & properties: an impossibility
  - But why care about arbitrary programs?
Hope for Real-Life Software

- Certain classes of analyses/property-checking of real-life software feasible in practice
  - Uses domain specific techniques, restrictions on program structure…
  - “Safety” properties of avionics software, device drivers, …
- A practitioner’s perspective

![Diagram showing relationships between Automation, “Large” Programs, and “Complex” Properties. The diagram indicates that currently, one can get any 2 out of 3.]

“Large” Programs

“Complex” Properties

Currently, can get any 2 out of 3

Automation
Some Driving Factors

- Compiler design and optimizations
  - Since earliest days of compiler design
- Performance optimization
  - Renewed importance for embedded systems
- Testing, verification, validation
  - Increasingly important, given criticality of software
- Security and privacy concerns
- Distributed and concurrent applications
  - Human reasoning about all scenarios difficult
Successful Approaches in Practical Software Verification

- Use of sophisticated abstraction and refinement techniques
  - Domain specific as well as generic
- Use of constraint solvers
  - Propositional, quantified boolean formulas, first-order theories, …
- Use of scalable symbolic reasoning techniques
  - Several variants of decision diagrams, combinations of decision diagrams & satisfiability solvers …
- Incomplete techniques that scale to real programs
Focus of today’s talk

Abstract Interpretation Framework

- Elegant unifying framework for several program analysis & verification techniques
- Several success stories
  - Checking properties of avionics code in Airbus
  - Checking properties of device drivers in Windows
  - Many other examples
    - Medical, transportation, communication …
- But, NOT a panacea
- Often used in combination with other techniques
Sequential Program State

Given sequential program P

- State: information necessary to determine complete future behaviour
  - (pc, store, heap, call stack)
  - pc: program counter/location
  - store: map from program variables to values
  - heap: dynamically allocated/freed memory and pointer relations thereof
  - call stack: stack of call frames
A simple program:

```c
void func(int a, int b)
{
    int x, y;
    L1: x = 0;
    L2: y = 1;
    L3: if (a >= b + 2)
        L4:   a = y;
        else
            L5:   b = x;
    L6: return;
}
```

State = (pc, store)
heap, stack unchanged within func
void func(int a, int b)
{
    int x, y;
    L1: x = 0;
    L2: y = 1;
    L3: if (a >= b + 2)
        L4:   a = y;
        else
        L5:   b = x;
    L6: return;
}
Programs as State Transition Systems

State: pc, x, y, a, b

void func(int a, int b)
{
    int x, y;
    L1: x = 0;
    L2: y = 1;
    L3: if (a >= b + 2)
        L4: a = y;
    else
        L5: b = x;
    L6: return;
}
Assertion Checking as Reachability

Path from an initial to an assertion violating state?

Absence of path: System cannot exhibit error

Presence of path: System can exhibit error

What happens with procedure calls/returns?
State Space: How large is it?

- State = (pc, store, heap, call stack)
  - pc: finite valued
  - store: finite if all variables have finite types
  - Every program statement effects a state transition
  - enum {wait, critical, noncritical} pr_state (finite)
  - int a, b, c (infinite)
  - bool *p, *q (infinite)
  - heap: unbounded in general
  - call stack: unbounded in general

- Bad news: State space infinite in general
Dealing with State Space Size

- Infinite state space
  - Difficult to represent using state transition diagram
  - Can we still do some reasoning?
- Solution: Use of abstraction
  - Naive view
    - Bunch sets of states together “intelligently”
    - Don't talk of individual states, talk of a representation of a set of states
    - Transitions between state set representations
  - Granularity of reasoning shifted
  - Extremely powerful general technique
    - Allows reasoning about large/infinite state spaces
Simple Abstractions

Group states according to values of variables and pc

```
void func(int a, int b)
{
    int x, y;
    L1: x = 0;
    L2: y = 1;
    L3: if (a >= b + 2)
        L4:   a = y;
        else
            L5:   b = x;
        L6: return;
}
```

Group states with same pc

State: pc, x, y, a, b

L1, 2, 7, 2, 0
L1, 3, 20, 8, 7
L1, -1, 10, 9, 1
Programs as State Set Transformers

Group states according to values of variables and pc

void func(int a, int b)  
{  int x, y;

L1: x = 0;
L2: y = 1;
L3: if (a >= b + 2) 
   L4:   a = y;
else
   L5:   b = x;
L6: return;

}  

Group states with same pc

\(a < 5\)

\(a \geq 5\)
Recall: Set of (potentially infinite) concrete states is an abstract state

Think of program as abstract state transformer

Programs as Abstr State Transformers

State: pc, x, y, a, b

L4: a = y

L4, -1, 10, 9, 1

L4, 2, 7, 2, 0

L6, -1, 10, 10, 1

L6, 2, 7, 7, 0

L6, 3, 20, 20, 7

L4, 3, 20, 8, 7
Programs as Abstr State Transformers

- Recall: Set of (potentially infinite) concrete states is an abstract state
- Think of program as abstract state transformer

Central problem: Compute $a_2$ from $a_1$ and prog stmt (abstract state transitions)

Abstract state $a_1$ → Program statement as abstract state transformer → Abstract state $a_2$

L4: $a = y$
A Generic View of Abstraction

- Every subset of concrete states mapped to unique abstract state
- Desirable to capture containment relations
- Transitions between state sets (abstract states)
Mathematical Foundations of Abstract Interpretation

- Set of concrete states: \( S \)
  - Concrete lattice \( C = (\mathcal{P}(S), \subseteq, \cup, \cap, S, \emptyset) \)
Mathematical Foundations of Abstract Interpretation

- Abstract lattice \( \mathbf{A} = (\mathcal{A}, \subseteq, \sqcup, \sqcap, \top, \bot) \)

- Abstraction function \( \alpha : \wp(S) \rightarrow \mathbf{A} \)
  
  - Monotone: \( S_1 \subseteq S_2 \Rightarrow \alpha(S_1) \sqsubseteq \alpha(S_2) \) for all \( S_1, S_2 \subseteq S \)
  
  - \( \alpha(S) = \top, \quad \alpha(\emptyset) = \bot \)

- Concretization function \( \gamma : \mathbf{A} \rightarrow \wp(S) \)
  
  - Monotone: \( a_1 \sqsubseteq a_2 \Rightarrow \gamma(a_1) \subseteq \gamma(a_2) \) for all \( a_1, a_2 \in \mathbf{A} \)
  
  - \( \gamma(\top) = S, \quad \gamma(\bot) = \emptyset \)
Mathematical Foundations of Abstract Interpretation

- \( \alpha \) and \( \gamma \) form a **Galois connection**
  - First view: \( S_1 \subseteq \gamma(\alpha(S_1)) \) for all \( S_1 \subseteq S \)
Mathematical Foundations of Abstract Interpretation

- $\alpha$ and $\gamma$ form a **Galois connection**
  - First view: $S_1 \subseteq \gamma(\alpha(S_1))$ for all $S_1 \subseteq S$
  - $\alpha(\gamma(a_1)) \subseteq a_1$ for all $a_1 \in \mathcal{A}$
Mathematical Foundations of Abstract Interpretation

- \( \alpha \) and \( \gamma \) form a **Galois connection**

  - Second (equivalent) view:
    \[
    \alpha(S_1) \subseteq a_1 \iff S_1 \subseteq \gamma(a_1) \text{ for all } S_1 \subseteq S, \ a_1 \in A
    \]
Computing Abstract State Transformers

- Concrete state set transformer function
  - Example:

$$S_1 = \{ (L4, x, y, a, b) | \ldots \} : \text{set of concr. states}$$

$$S_2 = \{ (L6, x, y, a', b) | \exists (L4, x, y, a, b) \in S_1 \land a' = y \}$$

$$= F^c(S1) : \text{set of concrete states}$$
Computing Abstract State Transformers

Abstract state transformer function

- Example:

\[ a_2 = \alpha(\gamma(a_1)) \]

ideally, but \[ F^A(a_1) \supseteq \alpha(F^C(\gamma(a_1))) \] often used
Computing Abstract State Transformers

- Abstract state transformer for if-then-else
  - Example:

```
L3: if (a >= b+2) goto L4 else goto L5
```

```
a2 = a1 \cap \alpha ((x, y, a, b) \mid a >= b+2)
a3 = a1 \cap \alpha ((x, y, a, b) \mid a < b+2)
a2 \in A
a3 \in A
```

```
pc in a2: L4
pc in a3: L5
```
Dealing with Loops

- Example: ....
  L1: while (a > b) do
  L2: <loop body>
  L9: end while

Given
F^A: abstr state transformer of loop body,
a: abstr state at L1 the first time L1 is reached

What is the abstract loop invariant at L1?
Dealing with Loops

Given
\(F^A: \) abstr state transformer of loop body,
\(a: \) abstr state at \(L1\) the first time \(L1\) is reached

What is the abstract loop invariant at \(L1\)?

\(a_{\text{cond}} = \alpha (\{s \mid s \text{ is a concrete state with } a > b\})\)

Current view of abstract loop invariant
Dealing with Loops

Given

\( F^A \) : abstr state transformer of loop body,
\( a \) : abstr state at L1 the first time L1 is reached

What is the abstract loop invariant at L1?

\[ a \text{cond} = \alpha ( \{ s \mid s \text{ is a concrete state with } a > b \} ) \]

Current view of abstract loop invariant
Dealing with Loops

Given
\( F^A : \) abstr state transformer of loop body,
\( a : \) abstr state at L1 the first time L1 is reached

What is the abstract loop invariant at L1?

\[ a\text{cond} = \alpha (\{s | s \text{ is a concrete state with } a > b\}) \]

Current view of abstract loop invariant

\[ \text{[Diagram]} \]
Dealing with Loops

Given
$F^A$: abstr state transformer of loop body,
$a$: abstr state at L1 the first time L1 is reached

What is the abstract loop invariant at L1?

$a \text{cond} = \alpha (\{s \mid s \text{ is a concrete state with } a > b\})$

Abstract loop invariant
Dealing with Loops

Given

$F^A$: abstr state transformer of loop body,

$a$: abstr state at L1 the first time L1 is reached

What is the abstract loop invariant at L1?

$a_{\text{cond}} = \alpha (\{s \mid s \text{ is a concrete state with } a > b\})$

Loop invariant at L1 is limit of the sequence:

$z_0 = a$
Dealing with Loops

Given

\( F^A \): abstr state transformer of loop body,
\( a \): abstr state at L1 the first time L1 is reached

What is the abstract loop invariant at L1?

\[ a \text{cond} = \alpha (\{ s \mid s \text{ is a concrete state with } a > b \}) \]

Loop invariant at L1 is limit of the sequence:
\[ z_0 = a \]
\[ z_1 = a \cup F^A (z_0 \cap \neg a\text{cond}) \]
Dealing with Loops

Given

\( F^A : \) abstract state transformer of loop body, 

\( a : \) abstract state at L1 the first time L1 is reached

What is the abstract loop invariant at L1?

\[ a \text{cond} = \alpha ( \{ s \mid s \text{ is a concrete state with } a > b \} ) \]

Loop invariant at L1 is limit of the sequence:

\[ z_0 = a \]

\[ z_1 = a \bigcup F^A (z_0 \bigcap \text{acond}) \]

……

\[ z_{i+1} = a \bigcup F^A (z_i \bigcap \text{acond}) \]
Dealing with loops

- Loop invariant at L1 is limit of the sequence:
  - $z_0 = a, \ldots, z_{i+1} = a \uparrow F^A (z_i \sqcap a\text{cond})$
  - The limit exists and is the least fixpoint of $g: \mathcal{A} \rightarrow \mathcal{A}$ where $g(z) = a \uparrow F^A (z \sqcap a\text{cond})$
- Difficult to compute if $A$ has infinite ascending chains
- Use an extrapolation (widen) operator $r$
  - $w_0 = z_0$, and $w_{i+1} = w_i \triangledown z_{i+1}$ for all $i \geq 0$
  - By definition of $\triangledown$,
    - Sequence of $w_i$’s stationary after finitely many $i$’s
    - Stationary value $w^*$ overapproximates limit of sequence of $z_i$’s
- Theory of abstract interpretation guarantees that $\gamma(w^*)$ overapproximates loop invariant at L1
Putting It All Together

- Given a program P and an assertion \( \varphi \) at location L
  - Choose an abstract lattice (domain) A with a \( \triangleright \) operator
  - Compute abstract invariant at each location of P
  - If abstract invariant at L is \( a_L \), check if \( \gamma(a_L) \) satisfies \( \varphi \)
  - The theory of abstract interpretation guarantees that
    \[ \gamma(a_L) \supseteq \text{concrete invariant at L} \]
A Simple Abstract Domain
Simplest domain for analyzing numerical programs

- Represent values of each variable separately using intervals
- Example:

L0:  x = 0; y = 0;
L1:  while (x < 100) do
    L2:  x = x+1;
    L3:  y = y+1;
L4:  end while

If the program terminates, does x have the value 100 on termination?
Interval Abstract Domain

- Abstract states: pairs of intervals (one for each of x, y)
  - [-10, 7], (-1, 20]
  - \(\sqsubseteq\) relation: Inclusion of intervals
  - [-10, 7], (-1, 20] \(\sqsubseteq\) [-20, 9], (-1, +\(\infty\))
  - \(\sqcup\) and \(\sqcap\): union and intersection of intervals
  - \([a, b] \sqcup x\ [c, d] = [e, f]\), where
    - \(e = a\) if \(c \geq a\), and \(e = -\infty\) otherwise
    - \(f = b\) if \(d \leq b\), and \(f = +\infty\) otherwise
  - \(\sqcup y\) similarly defined, and \(\sqcup\) is simply \((\sqcup x, \sqcup y)\)
  - \(\perp\) is empty interval of x and y
  - \(\top\) is \((-\infty, +\infty), (-\infty, +\infty)\)
Analyzing our Program

L0: \( x = 0; y = 0; \)

L1: while \((x < 100)\) do

\[ L2: \quad x = x+1; \]

\[ L3: \quad y = y+1; \]

L4: end while
Some Concluding Remarks

- Abstract interpretation: a fundamental technique for analysis of programs
- Choice of right abstraction crucial
- Often getting the right abstraction to begin with is very hard
  - Need automatic refinement techniques
- Very active area of research