Abstract Interpretation and Program Verification

Supratik Chakraborty
IIT Bombay
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?
Program Verification: An Example

```
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);
```

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

INVARIANT or PROPERTY
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
- This doesn’t reduce the importance of proving programs correct
  - Can we solve this in special (real-life) cases?
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
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, Horn clauses ...
- 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
int 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 (a-b);
}
```

State = (pc, store) heap, stack unchanged within func

```
L1, 2, 7, 2, 0
L2, 0, 7, 2, 0
L3, 0, 1, 2, 0
L4, 0, 1, 2, 0
L6, 0, 1, 1, 0
```
int 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 (a-b);
}
Programs as State Transition Systems

State: pc, x, y, a, b

```c
int 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 (a - b);
}
```
Specifying Program Properties

Pre-condition:
{ a + b >= 0 }
int func(int a, int b)
{ int x, y;
  L1: x = 0;
  L2: y = 1;
  L3: if (a >= b + 2)
      // assert (a-b <= 1);
      L4:   a = y;
  else
      L5:   b = x;
  L6: return (a-b);
}

Post-condition:
{ ret_val <= 1  }

State: pc,  x, y,  a, b
Specifying Program Properties

State: pc, x, y, a, b

Pre-condition:
{ a + b >= 0 }  
int func(int a, int b)  
{ int x, y; 
  L1: x = 0; 
  L2: y = 1; 
  L3: if (a >= b + 2) // assert (a-b <= 1); 
      L4: a = y; 
  else 
      L5: b = x; 
  L6: return (a-b); 
}  

Post-condition:
{ ret_val <= 1 }
Assertion Checking as Reachability

Path from initial to 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

```
int 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 (a-b);
}
```

State: pc, x, y, a, b

Group states with same pc

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

Group states according to values of variables and pc

![Diagram showing program states based on conditions](image-url)

```c
int 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 (a-b);
}
```

- a < 5
- a >= 5

Group states with same pc
Programs as Abstr State Transformers

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

State: pc, x, y, a, b

L4: a = y

Program statement as concrete state transformer

L4, 2, 7, 2, 0

L4, -1, 10, 9, 1

L4, 3, 20, 8, 7

L6, 2, 7, 7, 0

L6, -1, 10, 10, 1

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)
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)
Pre-condition:
\{ a + b >= 0 \}

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

Post-condition:
\{ ret_val <= 1 \}
Pre-condition: \{ a + b \geq 0 \}

int func(int a, int b)
{
    int x, y;
    L1: x = 0;
    L2: y = 1;
    L3: if (a \geq b + 2) // assert (a-b \leq 1);
        L4:   a = y;
    else
        L5:   b = x;
    L6: return (a-b);
}

Post-condition: \{ ret_val \leq 1 \}

The Game Plan

How do we choose the right abstraction?
Is there a method beyond domain expertise?
Can we learn from errors in abstraction to build better (refined) abstractions?
Can refinement be automated?
Supratik Chakraborty, IIT Bombay

Pre-condition: \( \{ a + b \geq 0 \} \)

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

Post-condition: \( \{ \text{ret_val} \leq 1 \} \)

The Game Plan

Abstract state spaces can be infinite. What can we do to make abstract analysis practical? Finite ascending chains what beyond?

Abstract analysis engine
Desirable Properties of Abstraction

- Suppose $S_1 \subseteq S_2$ : subsets of concrete states
  - Any behaviour starting from $S_1$ can also happen starting from $S_2$

- If $\alpha(S_1) = a_1, \alpha(S_2) = a_2$ we want this monotonicity in behaviour in abstract state space too
  - Need ordering of abstract states, similar in spirit to $S_1 \subseteq S_2$
Structure of Concrete State Space

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

[Diagram with nodes labeled:
- Powerset of \( S \)
- Partial order
- Least upper bound
- Greatest lower bound
- Bottom element
- Top element]
Structure of Abstract State Space

- Abstract lattice $\mathcal{A} = (\mathcal{A}, \subseteq, \cup, \cap, \top, \bot)$

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

- Concretization function $\gamma : \mathcal{A} \to \wp(S)$
  - Monotone: $a_1 \subseteq a_2 \Rightarrow \gamma(a_1) \subseteq \gamma(a_2)$ for all $a_1, a_2 \in \mathcal{A}$
  - $\gamma(\top) = S, \quad \gamma(\bot) = \emptyset$
A Simple Abstract Domain
Supratik Chakraborty, IIT Bombay

Interval 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: intervals of values of x, pc implicit
  - [-10, 7]: \( \{ (x, y) \mid -10 \leq x \leq 7 \} \)
  - \((\infty, 20]\): \( \{ (x, y) \mid x \leq 20 \} \)

- \(\sqsubseteq\) relation: Inclusion of intervals
  - \([-10, 7] \sqsubseteq [-20, 9]\)

- \(\sqcup\) and \(\sqcap\): union and intersection of intervals
  - \([-10, 9] \sqcup [-20, 7] = [-20, 9]\)
  - \([-10, 9] \sqcap [-20, 7] = [-10, 7]\)

- \(\bot\) is empty interval of x

- \(\top\) is \((\infty, +\infty)\)
Interval Abstract Domain

- Abstract states: intervals of values of x, pc implicit
  
  \([-10, 7]: \{ (x, y) | -10 \leq x \leq 7 \} \)
  
  \((-\infty, 20]: \{ (x, y) | x \leq 20 \} \)

- \(\sqsubseteq\) relation: Inclusion of intervals
  
  \([-10, 7] \sqsubseteq [-20, 9]\)

- \(\sqcup\) and \(\sqcap\): union and intersection
  
  \([-10, 9] \sqcup [-20, 7] = [-20, 9]\)
  
  \([-10, 9] \sqcap [-20, 7] = [-10, 7]\)

- \(\bot\) is empty interval of x

- \(\top\) is \((-\infty, +\infty)\)

\[\alpha( \{(L1, 1, 3), (L1, 2, 4), (L1, 5, 7)\} ) = [1, 5]\]

\[\alpha( \{(L1, 5, 7), (L1, 7, 6), (L1, 9, 10)\} ) = [5, 9]\]

\[\alpha( \{(L1, 5, 7)\} ) = [5, 5]\]
Interval Abstract Domain

- Abstract states: pairs of intervals (one for $x$, $y$), pc implicit
  - $(\[-10, 7\], (-\infty, 20])$
  - $\subseteq$ relation: Inclusion of intervals
    - $(\[-10, 7\], (-\infty, 20]) \subseteq (\[-20, 9\], (-\infty, +\infty))$
  - $\sqcup$ and $\cap$: union and intersection of intervals
    - $(\[-10, 9\], (-\infty, 20]) \cap (\[-20, 7\], [3, +\infty)) = (\[-10, 7\], [3, 20])$
    - $(\[-10, 9\], (-\infty, 20]) \sqcup (\[-20, 7\], [3, +\infty)) = (\[-20, 9\], (-\infty, +\infty))$
- $\bot$ is empty interval of $x$ and $y$
- $\top$ is $(\(-\infty, +\infty\), (-\infty, +\infty))$
Desirable Properties of $\alpha$ and $\gamma$

For all $S_1 \subseteq C$ and $S_1 \subseteq \gamma(\alpha(S_1))$. 

Set of concrete states $C$ \hspace{2cm} Set of abstract states $A$

$S_1$ \hspace{1cm} $\gamma$ \hspace{1cm} $\alpha$
Desirable Properties of $\alpha$ and $\gamma$

\[ S_1 \subseteq \gamma(\alpha(S_1)) \] \[ \forall S_1 \subseteq C \]
\[ \alpha(\gamma(a_1)) \subseteq a_1 \] \[ \forall a_1 \in A \]

$\alpha$ and $\gamma$ form a Galois connection

Supratik Chakraborty, IIT Bombay
Desirable Properties of $\alpha$ and $\gamma$

- $\alpha$ 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 Transitions

Set of concrete states

Concrete state c1

L4: a = y

Concrete state c2

Set of abstract states

Abstract state a1

L4: a = y

Abstract state a2

Abstraction (\(\alpha\))

Concretization (\(\gamma\))
Computing Abstract State Transitions

- Concrete state set transformer function
  - Example:

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

\[
S_2 = \{ (L6, x, y, a', b) | (L4, x, y, a, b) \in S_1, \ a' = y \}
\]

\[= F^C(S_1) : \text{set of concrete states}\]

\*Monotone concrete state set transformer function for stmt at L4\*
Computing Abstract State Transitions

- Abstract state transformer function
  - Example:

\[
\text{Set of concrete states}
\]

\[
\begin{align*}
\alpha & = \gamma(\mathcal{F}^C(\gamma(a_1))) \\
\mathcal{F}^A(a_1) & \supseteq \alpha(\mathcal{F}^C(\gamma(a_1))) \text{ often used}
\end{align*}
\]
Example Abstr State Transition

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

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

\[\]

L2: \( x = x+1; \)

L3: \( y = y+1; \)

L4: end while

Abstract states: pairs of intervals (one for \(x, \ y\)), pc implicit

\[
\frac{ly' = ly + 1}{uy' = uy + 1}
\]

\([lx, \ ux] , \ [ly, \ uy]\)

\(F^A(a1) \sqsupseteq \alpha(F^C(y(a1)))\)
Example Abstr State Transition

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

Abstract states: pairs of intervals (one for \( x, y \)), pc implicit

\[
F^A(a1) \equiv \alpha(F^C(y(a1)))
\]

\[
y = y+x;
\]

\[
([lx, ux], [ly, uy]) \rightarrow ([lx, ux], [ly', uy'])
\]

\[
ly' = ly + lx
\]

\[
uy' = uy + ux
\]
Computing Abstract State Transitions

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

  ![Diagram](image)

  \[
  \text{acond} = \alpha( (x, y, a, b) \mid a \geq b+2 )
  \]

  \[
  \text{acondb} = \alpha( (x, y, a, b) \mid a < b+2 )
  \]

  L3: if \((a \geq b+2)\) goto L4 else goto L5

  \[
  a2 = a1 \cap \text{acond}
  \]

  \[
  \text{pc in a2: L4}
  \]

  \[
  a3 = a1 \cap \text{acondb}
  \]

  \[
  \text{pc in a3: L5}
  \]
Dealing with Loops

Abstract pre-cond: a0

L0: \( a = 0; \) \( b = 0; \)

L1: …… ;

L7: while (a > b) do

L8: ….. ;

L19:…… ;

L20: end while

L21: ……;

L100: ……;
Dealing with Loops

L0: \( a = 0; \ b = 0; \)
L1: \( \ldots \ldots; \)
L7: while \( (a > b) \) do
    L8: \( \ldots\ldots; \)
    L19: \( \ldots\ldots; \)
L20: end while
L21: \( \ldots\ldots; \)
L100: \( \ldots\ldots; \)

Abstract state: \( a_1 = F_{a0}^A(a0) \)
Dealing with Loops

L0: \( a = 0; \ b = 0; \)

L1: \( \ldots \ldots; \)

L7: while \((a > b)\) do

L8: \( \ldots \ldots; \)

L19: \( \ldots \ldots; \)

L20: end while

L21: \( \ldots \ldots; \)

L100: \( \ldots \ldots; \)

Abstract state: \( a_{7} = F_{1..7}^{A}(a_{1}) \)
Dealing with Loops

L0:  a = 0; b = 0;
L1:  ......;
L7:  while (a > b) do
    L8:  ......;
    L19: ......;
L20: end while
L21:  ......;
L100: ......;

Abstract state a20? Can’t be computed as $F^A_{8..19}$ (a7 □ acond)
Loop may iterate 0,1,2,... times

$\alpha(...)$ = acond

Loop Body
Dealing with Loops

L0: \( a = 0; \ b = 0; \)

L1: \( \ldots \ldots \); \( \alpha(\text{not } \ldots) = acondb \)

L7: while \((a > b)\) do

L8: \( \ldots \ldots \); \( \alpha(acondb) \)

L100: \( \ldots \ldots \);

L100: \( \ldots \ldots \);

Calculate abstract loop invariant \( a7^* \) at L7. Whenever L7 is reached in program, corresponding abstract state  

Abstract state \( a20 = (a7^* \quad acondb) \)
Dealing with Loops

L0: \(a = 0; b = 0;\)

L1: 

L7: while \((a > b)\) do

L8: 

L19: 

L20: end while

L21: 

L100: 

Abstract state: \(a_{21} = a_{20}\)
Dealing with Loops

L0:   a = 0; b = 0;
L1:    ...... ;
L7:    while (a > b) do
       L8:    ..... ;
L19:   ..... ;
L20:   end while
L21:   ..... ;
L100:  ..... ;

Abstract state:
\[ a_{100} = F^A_{21..100}(a_{21}) \]

Loops can be handled if we know how to compute abstract loop invariants
Computing Abstract Loop Invariant

- Example: ....
  L7: while (a > b) do
    L8: .......;
    L19: .......;
  L20: end while

Given
F^A : abstr state transformer of loop body L8...L19
a : abstr state at L7 the first time L7 is reached

What is the abstract loop invariant at L7?
Computing Abstract Loop Invariant

Given
\( F^A : \) abstr state transformer of loop body,
\( a : \) abstr state at L7 the first time L7 is reached

What is the abstract loop invariant at L7?

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

Current view of abstract loop invariant
Computing Abstract Loop Invariant

Given

\( F^A \) : abstr state transformer of loop body,
\( a \) : abstr state at L7 the first time L7 is reached

What is the abstract loop invariant at L7?

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

Current view of abstract loop invariant

Supratik Chakraborty, IIT Bombay
Computing Abstract Loop Invariant

Given

\( F^A : \text{abstr state transformer of loop body} \),
\( a : \text{abstr state at L7 the first time L7 is reached} \)

What is the abstract loop invariant at L7?

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

Current view of abstract loop invariant

Recall: Meet-over-paths
Computing Abstract Loop Invariant

Given

\( F^A : \) abstr state transformer of loop body,
\( a : \) abstr state at L7 the first time L7 is reached

What is the abstract loop invariant at L7?

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

Abstract loop invariant

How do we calculate this effectively without knowing bound of loop iterations?
Abstract Loop Invariant: Another view

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

Successive views of the loop invariant at L7:
\[ z_0 = a \]
Abstract Loop Invariant: Another view

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

Successive views of the loop invariant at L7:
- \( z_0 = a \)
- \( z_1 = a \sqcap F^A (z_0 \nabla a\text{cond}) \)
Abstract Loop Invariant: Another View

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

Successive views of loop invariant at L7:
\[ z_0 = a \]
\[ z_1 = a \sqcap F^A (z_0 \sqcap a_{\text{cond}}) \]
\[ z_2 = a \sqcap F^A (z_1 \sqcap a_{\text{cond}}) \]
Abstract Loop Invariant: Another View

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

Successive views of loop invariant at L7:

\[ z_0 = a \]
\[ z_1 = a \cup F^A (z_0 \cap \text{acond}) \]
\[ z_2 = a \cup F^A (z_1 \cap \text{acond}) \]

\ldots\ldots

\[ z_{i+1} = a \cup F^A (z_i \cap \text{acond}) \]
Abstract Loop Invariant: Another View

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

Successive views of loop invariant at L7:

\[ z_0 = a = a \sqcup \mathcal{F}^A (\perp \sqcap \text{acond}) \]
\[ z_1 = a \sqcup \mathcal{F}^A (z_0 \sqcap \text{acond}) = g(\perp) \]
\[ z_2 = a \sqcup \mathcal{F}^A (z_1 \sqcap \text{acond}) = g(g(\perp)) \]

\[ z_{i+1} = a \sqcup \mathcal{F}^A (z_i \sqcap \text{acond}) = g(\ldots g(\perp) \ldots) \]

\[ z_0 \sqsubseteq z_1 \sqsubseteq z_2 \sqsubseteq \ldots \]

Reasonable requirements:

\[ \mathcal{F}^A (\perp) = \perp \]

If \( a_1 \sqsubseteq a_2 \) then \( \mathcal{F}^A (a_1) \sqsubseteq \mathcal{F}^A (a_2) \)

\[ g(z) = a \sqcup \mathcal{F}^A (z \sqcap \text{acond}) \]

\[ g(\ ) \text{ monotone} \]
Abstract Loop Invariant: Another View

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

Successive views of loop invariant at L7:

\[
\begin{align*}
z_0 &= g(\bot) \\
z_1 &= g(g(\bot)) \\
z_2 &= g(g(g(\bot))) \\
&\vdots \\
z_i &= g(\ldots g(\bot)\ldots)
\end{align*}
\]

Abstract loop invar = \( \lim_{i \to \infty} g^{(i)} (\bot) \)

Reasonable requirements:

\( F^A(\bot) = \bot \)

If \( a_1 \sqsubseteq a_2 \) then \( F^A(a_1) \sqsubseteq F^A(a_2) \)

\[
g(z) = a \sqcup F^A(z \cap a\text{cond})
\]

\( g(\_\_) \) monotone
Abstract Loop Invariant: Another View

Abstract loop invar = \( \lim_{i \to \infty} g^{(i)}(\bot) \)

= smallest \( a^* \) s.t. \( g(a^*) = a^* \)

= “least fixed point” of \( g(\ ) \)

Reasonable requirements:

\( F^A(\bot) = \bot \)

If \( a1 \sqsubseteq a2 \) then \( F^A(a1) \sqsubseteq F^A(a2) \)

\( g(z) = a \sqcup F^A(z \sqcap acond) \)

\( g(\ ) \text{ monotone} \)
Abstract Loop Invariant: Least Fixed Point View

Abstract loop invar $a^*$ computable if $\mathcal{A}$ has no infinite ascending chains

What if there are infinite ascending chains? Can we at least compute an overapprox of $a^*$?

Observe the sequence

$$g(\bot) \subseteq g^2(\bot) \subseteq \ldots \subseteq g^{(i)}(\bot) \text{ upto i terms}$$

and extrapolate ("informed guess") to a proposed overapprox of $a^*$

Special extrapolation (widen) operator $\triangledown$
Abstract Loop Invariant: Widen Operator

\[ \nabla: A \times A \rightarrow A \]

Current estimate of limit

Revised estimate of limit

Next element in sequence

Current estimate of limit
Abstract Loop Invariant: Widen Operator

\[ \nabla: A \times A \to A \]

Required properties of \( \nabla \)

For every \( a_1, a_2 \) in \( A \)
\[ a_1 \nabla a_2 \supseteq a_1 \quad \text{and} \quad a_1 \nabla a_2 \supseteq a_2 \]

For every \( a_0 \subseteq a_1 \subseteq a_2 \subseteq \ldots \), the sequence
\[ z_0 = a_0 \]
\[ z_1 = z_0 \nabla a_1 \]
\[ z_2 = z_1 \nabla a_2 \]
\[ \ldots \]
\[ z_{i+1} = z_i \nabla a_{i+1} \]

stabilizes, i.e.
There exists an \( i \geq 0 \) s.t. \( z_i = z_{i+1} = z_{i+2} = \ldots \)

Stabilized value \( z^* \supseteq \) limit of \( a_0, a_1, a_2, \ldots \).
Abstract Loop Invariant: Widen Operator

∇: A x A → A

Compute $g(\bot)$, $g^2(\bot)$, ... $g^{(k)}(\bot)$ for parameter $k > 0$

Define $a_0 = g^{(k)}(\bot)$
$a_1 = g(z_0)$
$a_2 = g(z_1)$

........
$a_i = g(z_{i-1})$

Fact: $g^{(k+j)}(\bot) \subseteq a_j \subseteq a_{j+1}$ for all $j \geq 0$

Recall $g$: $A \rightarrow A$ is monotone
Abstract Loop Invariant: Widen Operator

\( \nabla: \mathbb{A} \times \mathbb{A} \rightarrow \mathbb{A} \)

Compute \( g(\perp) \), \( g^2(\perp) \), \( \ldots \) \( g^{(k)}(\perp) \) for parameter \( k > 0 \)

Define \( a_0 = g^{(k)}(\perp) \) \( z_0 = a_0 \)
\( a_1 = g(z_0) \) \( z_1 = z_0 \uplus a_1 \)
\( a_2 = g(z_1) \) \( z_2 = z_1 \uplus a_2 \)
\( \ldots \)
\( a_i = g(z_{i-1}) \) \( z_i = z_{i-1} \uplus a_i \)

Fact: \( g^{(k+j)}(\perp) \subseteq a_j \subseteq a_{j+1} \) for all \( j \geq 0 \)

If \( z_i = z_{i+1} \), then
\( a_{j+1} = a_{i+1} \) for all \( j \geq i \)
\( z_j = z_i \) for all \( j \geq i \)

Can detect when sequence stabilizes
Abstract Loop Invariant: Widen Operator

\[ \nabla: A \times A \rightarrow A \]

Compute \( g(\perp), g^2(\perp), \ldots g^{(k)}(\perp) \) for parameter \( k > 0 \)

Define \( a_0 = g^{(k)}(\perp) \)

\[ z_0 = a_0 \]
\[ a_1 = g(z_0) \]
\[ z_1 = z_0 \nabla a_1 \]
\[ a_2 = g(z_1) \]
\[ z_2 = z_1 \nabla a_2 \]
\[ \ldots \]
\[ a_i = g(z_{i-1}) \]
\[ z_i = z_{i-1} \nabla a_i \]

Stabilized value \( z^* \) overapproximates \( g^{(i)}(\perp) \) for all \( i \geq 0 \)

Abstract loop invariant

In fact, \( g^{(r)}(z^*) \) also overapproximates \( g^{(i)}(\perp) \) for all \( r \geq 0 \)
Another View of Widening

Supratik Chakraborty, IIT Bombay

Least fixed point

Post-fixed points $g(x) \subseteq x$

Fixed points $g(x) = x$

Pre-fixed points $x \subseteq g(x)$
Another View of Widening

Post-fixed points $g(x) \subseteq x$

Fixed points $g(x) = x$

Pre-fixed points $x \subseteq g(x)$

Fixed points $g(x) = x$
Another View of Widening

\[ z^* = z^* \sqcup g(z^*) \]
implies
\[ g(z^*) \sqsubseteq z^* \]

\[ z^* \] is a
post-fixed point

\[ g^k(\bot) \]
\[ g^{k+1}(\bot) \]

\[ z^* \]
\[ z_1 \]
\[ z_2 \]
\[ z_3 \]
\[ z_m \]

Post-fixed points
\[ g(x) \sqsubseteq x \]

Fixed points
\[ g(x) = x \]

Pre-fixed points
\[ x \sqsubseteq g(x) \]

Supratik Chakraborty, IIT Bombay
Another View of Widening

$z^* = z^* \vee g(z^*)$ implies $g(z^*) \sqsubseteq z^*$

$z^*$ is a post-fixed point

$g^{(r)}(z^*)$ is a post-fixed point and $\text{lfp} \sqsubseteq g^{(r)}(z^*)$

$z^* = z^* \sqcup g(z^*)$
implies $g(z^*) \sqsubseteq z^*$

$z^*$ is a post-fixed point

$g(z^*)$ is a post-fixed point and $\text{lfp} \sqsubseteq g^{(r)}(z^*)$

$g(\bot)$
$g^{(k)}(\bot)$

Pre-fixed points $x \sqsubseteq g(x)$

Fixed points $g(x) = x$

Post-fixed points $g(x) \sqsubseteq x$

Supratik Chakraborty, IIT Bombay
Putting It All Together

- Given a program $P$ and an assertion $\varphi$ at location $L$
  - Choose an abstract lattice (domain) $A$ with a $\triangledown$ 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
    
    - $\gamma(a_L) \supseteq \text{concrete invariant at } L$
Interval 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)
  - $[-10, 7]$, $(-\infty, 20]$
  - $\sqsubseteq$ relation: Inclusion of intervals
    - $[-10, 7]$, $(-\infty, 20] \sqsubseteq [-20, 9]$, $(-\infty, +\infty)$
  - $\cup$ and $\cap$: union and intersection of intervals
    - $[a, b] \sqcup [c, d] = [e, f]$, where
      - $e = a$ if $c \geq a$, and $e = -\infty$ otherwise
      - $f = b$ if $d \leq b$, and $f = +\infty$ otherwise
    - $\sqcap$ 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: \(x = x+1;\)

L3: \(y = y+1;\)

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