Overview

• Program static analysis
• Abstract interpretation
• Static analysis techniques

What is static analysis?

• The analysis to understand computer software without executing programs
  – Simple coding style
    • Empty statement, EqualsHashcode
  – Complex property of the program
    • the program's implementation matches its specification
    • "Given program P and specification S, does P satisfy S ?"
• Can be conducted on source code or object code

Has anyone done static analysis?

• Code review
• ...

Why static analysis?

• Program comprehension
  – Is this value a constant?
• Bug finding
  – Is a file closed on every path after all its access?
• Program optimization
  – Constant propagation

An Informal Introduction to Abstract Interpretation

Patrick Cousot[2]
Modified by Na Meng
<table>
<thead>
<tr>
<th>Semantics &amp; Safety</th>
<th>Undecidability</th>
</tr>
</thead>
</table>
| • The **concrete semantics** of a program formalizes (is a mathematical model of) the set of all its possible executions in all possible execution environments  
• **Safety**: No possible execution in any possible execution environment can reach an erroneous state | • The concrete semantics of a program is undecidable  
– Given an arbitrary program, can you prove that it halts or not on any possible input?  
– Turing proved no algorithm can exist that always correctly decides whether, for a given arbitrary program and its input, the program halts when run with that input |

<table>
<thead>
<tr>
<th>Abstract Semantics</th>
<th>Why is Testing/Debugging insufficient?</th>
</tr>
</thead>
</table>
| • A sound approximation (superset) of the concrete semantics  
• It covers all possible concrete cases  
• If the abstract semantics is proved to be safe, then so is the concrete semantics  
• **Abstract interpretation**  
  – abstract semantics + proof of safe properties | • Only consider a subset of the possible executions  
• No correctness proof  
• No guarantee of full coverage of concrete semantics |

<table>
<thead>
<tr>
<th>Static Analysis Techniques</th>
<th>Model Checking</th>
</tr>
</thead>
</table>
| • Model checking  
• Theorem proving  
• Data flow analysis | • The abstract semantics is modeled as a finite state machine of the program execution  
• The model can be manually defined or automatically computed  
• Each state is enumerated exhaustively to automatically check whether this model meets a given specification |
import java.util.Random;
public class Rand {
    public static void main (String[] args)
    {
        Random random = new Random(42); // (1)
        int a = random.nextInt(2); // (2)
        System.out.println("a=");
        int b = random.nextInt(3); // (3)
        System.out.println("b=");
        int c = a/(b+a-2); // (4)
        System.out.println("c=");
    }
}

An Example [3]

Another Example [9]

• Consider a system: simple microwave oven
  • States of the system correspond to values of 3 boolean variables:
    • Either door is closed or not closed
    • Either microwave is running or it is stopped
    • Either the food in the microwave is warm or it is cold

• Using Temporal Logic, one can say
  • Specification: microwave does not heat the food up until the door is closed
  • \( \Rightarrow \neg \text{hot holds until closed} \)
  • Formula \( f = \neg \text{hot} \cup \neg \text{closed} \)
• Given \( f \) and model, model checking can return whether or not the model satisfies \( f \)
• If not, a counterexample is returned, showing a path of execution whereby the system fails to satisfy the formula

Model Checking

• Model microwave as a simple transition system

  • \text{hot}, \text{closed}

Advantages of Model Checking

• No proofs
• Procedure is completely automatic.
• Fast (linear in size of model and in size of specification)
• Counterexamples
• Logic is very expressive: allows for easy modeling of real-world protocols
Disadvantages of Model Checking

- There can be too many states to enumerate (state explosion)
- Abstract model creation puts burden on programmers
- The model may be wrong
  - If verification fails, is the problem in the model or the program?

Solutions to Space Explosion

- 1987: Ken McMillan developed a symbolic model checking approach where the system was represented using Binary Decision Diagrams
  - Data structure for representing boolean functions
  - Concise representations for transition systems, fast manipulation
  - Good for synchronous systems
- Partial Order Reduction: reduce number of states that must be explicitly enumerated
  - Good for asynchronous systems

Today’s Model Checkers

- Can handle systems with between 100 and 300 state variables
- Systems with 10^{120} reachable states have been checked!
- Using appropriate abstraction techniques, systems with an essentially unbounded number of states can be checked

Theorem Proving [4]

- Soundness
  - If the theorem is valid then the program meets specification
  - If the theorem is provable then it is valid

Verification Condition Generation

- State predicates/assertions: Boolean functions on program states
  - E.g., $x = 8$, $x < y$, true, false
- You can deduce verification condition predicates from known predicates at a given program location

- From programs and specs to theorems
  - Verification condition generation
- From theorems to proofs
  - Theorem provers
Hoare Triples [6]

• For any predicates P and Q and program S,

{P} S {Q}

says that if S is started in a state satisfying P, then it terminates in Q

– E.g., \{true\} \(x := 12\) (\(x = 12\)), \(x := x + 1\) (\(x \leq 40\))

Precise Triples

• If \{P\} S R or \{Q\} S R hold, then does \{P \lor Q\} S R hold?

• Weakest preconditions

– The most general precondition \{P \lor Q\}, is the

 weakest precondition on the initial state

 ensuring that execution of S terminates in a

 final state satisfying R.

\(\forall x, P \Rightarrow wp(S, R)\)

– E.g., \(x < 13\) \(x = x + 3\) (\(x > 13\)) vs. \(x > 10\) \(x = x + 3\) (\(x > 13\)), which precondition is weaker?

Example: Does the program satisfy the specification?

• Specification

 requires true (precondition)

 ensures \(c = a \lor b\) (postcondition)

• Program

```java
bool or(bool a, bool b) {
    if (a)
        c := true;
    else
        c := b;
    return c
}
```

Theorem Proving

• Step 1

– Given the post condition, infer the weakest precondition of the program

• Step 2

– Verify whether the given precondition can infer the weakest precondition

• If so, the program satisfies the specification

• Otherwise, it does not

Precise Triples

• If \{P\} S (Q \land R) holds, then do \{P\} S Q and \{P\} S R hold?

• Strongest postcondition

– The most precise postcondition \(Q \land R\),

 which implies any postcondition satisfied by

 the final state of any execution \(x\) of S

\(\forall x, sp(S, P) \Rightarrow Q\)

– E.g., \{true\} \(x := 12\) (\(x = 12\)) vs. \{true\} \(x := 12\)

\(x > 0\), which postcondition is stronger?

Weakest Precondition Rules

• \(wp(x := E, B) = B[E/x]\)

• \(wp(s1; s2, B) = wp(s1, wp(s2, B))\)

• \(wp(if \ E \ then \ s1 \ else \ s2, B) = (E \Rightarrow wp(s1, B) \land (~E \Rightarrow wp(s2, B))\)

• \(wp(\text{assert} \ E, B) = E \land B\)

• What is the WP of our example program?

– \(wp(S) = (a \Rightarrow true = a \lor b) \land (~a \Rightarrow b = a \lor b)\)
• Conjecture to be proved:
  – true⇒(a⇒true=a∨b)(a⇒b=a∨b)

Data Flow Analysis [5]

Peter Lee
Modified by Na Meng

Data Flow Analysis
• A technique to gather information about the possible set of values calculated at various points in a computer program

How to do data flow analysis?
• Set up data-flow equations for each node of the control flow graph
  \[ \text{out}_i = \text{trans}(\text{in}_i) \]
  \[ \text{in}_i = \text{join}_{prev}(\text{out}_i) \]

  • Solve the equation set iteratively, until reaching a fixpoint: all in-states do not change
    for \( i \leftarrow 1 \) to \( N \)
    initialize node \( i \)
    while (sets are still changing)
      for \( i \leftarrow 1 \) to \( N \)
        recompute sets at node \( i \)

Work List Iterative Algorithm

for \( i \leftarrow 1 \) to \( N \)
  initialize node \( i \)
  add node \( i \) to worklist
while (worklist is not empty)
  remove a node \( n \) from worklist
  calculate out-state based on in-state
  if out-state is different from the original value
    worklist = worklist U succ(n)

Directions of Data Flow Analysis
• Forward
  – Calculate output-states based on input-states
• Backward
  – Calculate input-states based on output-states
An Example [7]

• What variable definitions reach the current program point?

```plaintext
1: int N = input()
2: initialize array A[N + 1]
3: call check(N)
4: int I = 1
5: while (I < N) {
6:     A(I) = A(I) + I
7:     I = I + 1 }
8: print A(N)
```

Reaching Definition

• A definition at program point \( d \) reaches program point \( u \) if there is a control-flow path from \( d \) to \( u \) that does not contain a definition of the same variable as \( d \)

Reaching Definition Equations

• Forward analysis

\[
\text{out}_b = \text{gen}_b \cup (\text{in}_b - \text{kill}_b)
\]

\[
\text{in}_b = \bigcup_{p \in \text{pred}(b)} \text{out}_p
\]

• \( \text{gen}_b \): variable definitions generated by \( b \)
• \( \text{kill}_b \): definitions killed at \( b \) by redefinitions of the variable(s)
• initialization: \( \text{in} = \{ \} \)

Using Reaching Definition

• Detection of uninitialized variables

```plaintext
int x;
if (...) x = 1;
...
a = x;
```

Using Reaching Definition

• Loop-invariant Code Motion
  – Consider an expression inside a loop. If all reaching definitions are outside of the loop, then move the expression out of the loop

Revisit the Example[7]

• What variable definitions are or will be actually used?

```plaintext
1: int N = input()
2: initialize array A[N + 1]
3: call check(N)
4: int I = 1
5: while (I < N) {
6:     A(I) = A(I) + I
7:     I = I + 1 }
8: print A(N)
```
Live-out Variables

- A variable \( v \) is **live-out** of statement \( n \) if \( v \) is used along some control path starting at \( n \). Otherwise, we say that \( v \) is **dead**
  - "What variables' definitions are actually used?"

Liveness Analysis Equations

- Backward analysis

  \[
  \begin{align*}
  \text{in}_b &= \text{out}_b - \text{kill}_b \cup \text{gen}_b \\
  \text{out}_b &= \bigcup_{v \in \text{dom}_v} (\text{in}_v)
  \end{align*}
  \]

- \( \text{gen}_b \): variables used by \( b \)
- \( \text{kill}_b \): if \( v \) is defined without using \( v \), all its prior definitions are killed
- initialization: \( \text{out} = \{ \} \)

Using Liveness Analysis

- Dead code elimination
  - Suppose we have a statement defining a variable, whose value is not used, then the definition can be removed without any side effect

Available Expression

- An expression \( e \) is **available** at statement \( n \) if
  - it is computed along every path from entry node to \( n \), and
  - no variable used in \( e \) gets redefined between \( e \)'s computation and \( n \)

Using Available Expressions

- Common-subexpression elimination

  - What are the available expressions at 5?
  - What direction is the analysis?
  - How to define \( \text{gen}_b \) and \( \text{kill}_b \)?
  - What are the equations for \( \text{in}_b \) and \( \text{out}_b \)?
Our analyses so far

<table>
<thead>
<tr>
<th>union</th>
<th>intersection</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reaching definitions</td>
<td>Available expressions</td>
</tr>
<tr>
<td>Live variables</td>
<td></td>
</tr>
</tbody>
</table>

Questions

• Does work list iterative algorithm always terminate?

Lattices

• A lattice \( L \) is a (possibly infinite) set of values, along with \( \cup \) and \( \cap \) operations
  - \( \forall x, y \in L \exists \text{unique } w \text{ and } z \text{ such that } x \cup y = w \text{ and } x \cap y = z \)
  - \( \forall x, y \in L \) \( x \cup y \) and \( x \cap y \) are both elements of \( L \)
  - \( \forall x, y, z \in L \) \( (x \cup y) \cup z = x \cup (y \cup z) \) and \( (x \cap y) \cap z = x \cap (y \cap z) \)
  - \( \exists \perp, \top \in L \), \( \forall x \in L \), \( x \cap \perp = \perp \) and \( x \cup \top = \top \)

Monotonic Functions

• The join and meet operators induce a partial order on the lattice elements
  - \( x \leq y \text{ if and only if } x \cap y = x \text{ and } x \cup y = y \)
  - reflexive, anti-symmetric, transitive
• For a lattice \( L \), a function \( f: L \rightarrow L \) is monotonic if for all \( x, y \in L \)
  - \( x \leq y \Rightarrow f(x) \leq f(y) \) or \( x \leq y \Rightarrow f(x) \geq f(y) \)

Reaching definition is monotonic

\[ \text{out}_b = \text{gen}_b \cup (\text{in}_b \setminus \text{kill}_b) \]

• Proof (for single-variable single-block programs) by contradiction:
  - Suppose \( \text{in}_b = \{1\}, \text{out}_b = \{0\} \), where 1 means there is a variable definition, 0 means no definition, then \( \text{gen}_b = \{0\}, \text{kill}_b = \{1\} \).
  - However, \( \text{kill}_b = \{1\} \) only if the block \( b \) has a redefinition of the variable, which means \( \text{gen}_b = \{1\} \)

• Therefore, after limited number of iterations \( (N^* (E+1) \) at worst case), every definition is propagated to every node
• Therefore, we can find a fixpoint \( p \), such that \( f(p) = p \)
In dataflow analysis, we require that all flow functions be monotone and have only finite-length effective chains.

Ingredients of a Dataflow Analysis

- Flow direction
- Transfer function
- Meet operator (Join function)
- Dataflow information
  - Set of definitions, variables, and expressions
  - Initialization
  - How about concrete data values?

Inter-procedural Analysis [8]

Stephen Chong
Imported by Na Meng

Procedures

- So far we have looked at **intra-procedural** analysis: analyzing a single procedure
- **Inter-procedural** analysis uses calling relationships among procedures
  - Connect intra-procedural analysis results via call edges
  - Enable more precise analysis information

Inter-procedural CFG

```c
void main() {
    x = 7;
    r = p(x);
    x = r;
    z = p(x+10);
}
int p(int a) {
    int y;
    if (a < 9)
        y = 0;
    else
        y = 1;
    return y;
}
```

Imprecision

- Dataflow facts from one call site can "taint" results at other call sites
  - Is \( z \) a constant?
Inlining

- Make a copy of the callee’s CFG at each call site.

Exponential Size Increase

- How about recursive function calls?
  - p(int n) {... p(n - 1) ...}
- The exponential increase makes analysis infeasible.

Context Sensitivity

- Make a finite number of copies
- Use context information to determine when to share a copy
  - Different decisions achieve different tradeoffs between precision and scalability
- Common choice: approximation call stack

An Example

Procedure Summaries

- In practice, people don’t construct a single global CFG and then perform dataflow
- Instead, construct and use procedure summaries
- Summarize effect of callees on callers
  - E.g., is there any side effect on callers?
- Summarize effect of callers on callees
  - E.g., is any parameter constant?

Other Contexts

- Object/pointer sensitivity
  - What is the type of a given object and what are the corresponding possible method targets?
  - What is the value of a given object’s field?
**Pointer Analysis**

- What is the points-to set of p?
  ```
  int x = 3;
  int y = 0;
  int* p = unknown() ? &x : &y;
  ```
- Alias analysis
  - Decide whether separate memory references point to the same area of memory
  - Can be used interchangeably with pointer analysis (points-to analysis)

**Flow Sensitivity**

- Flow insensitive analysis
  - Perform analysis without caring about the statement execution order
  - E.g., analysis of c1;c2 will be the same as c2;c1
  - Address-taken, Steensgaard, Andersen
- Flow sensitive analysis
  - Observes the statement execution order

---

**An Example**

1. a = &b
2. b = &c
3. f = &d
4. d = &e
5. a = f

- After 5, both *a and *f point to d

**AddressTaken**

- Assume that variables whose addresses are taken may be referenced by all pointers
  - Address-taken variables: b, c, d, e
  - A single alias pointer set: {a, b, f, d}

**Steensgaard**

- Constraints
  - p = &x: x ∈ points-to(p)
  - p = q: points-to(p) ⊆ points-to(q)
  - p = *q ∀a ∈ points-to(q), points-to(p) = points-to(a)
  - *p = q ∀b ∈ points-to(p), points-to(b) = points-to(q)
  
**Andersen**

- Subset Constraints
  - p = &x: x ∈ points-to(p)
  - p = q: points-to(q) ⊆ points-to(p)
  - p = *q ∀a ∈ points-to(q), points-to(a) ⊆ points-to(p)
  - *p = q ∀b ∈ points-to(p), points-to(q) ⊆ points-to(b)

---

**Address Taken**

1. a = &b
2. b = &c
3. f = &d
4. d = &e
5. a = f
Flow-sensitive Pointer Analysis

\[ \text{out}_i = \text{gen}_i \cup (\text{in}_i - \text{kill}_i) \]
\[ \text{in}_i = \bigcup_{j \in \text{pred}(i)} (\text{out}_j) \]

- x = y: **strong update**
  - kill—clear pts(x)
  - gen—add pts(y) to pts(x)

- \*x = y:
  - If x definitely points to a single concrete memory location z, pts(z) = y (strong update)
  - If x may point to multiple locations, then **weak update** by adding y to pts of all locations

Reference

- [9] Model Checking and Software Verification, [https://courses.cs.washington.edu/courses/csep590/03su/Lectures/lecture1.ppt](https://courses.cs.washington.edu/courses/csep590/03su/Lectures/lecture1.ppt)