Program Static Analysis

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, Equals-Hashcode
  – 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
Semantics & Safety

- 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

Undecidability

- 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

Abstract Semantics

- 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

Why is Testing/Debugging insufficient?

- Only consider a subset of the possible executions
- No correctness proof
- No guarantee of full coverage of concrete semantics

Static Analysis Techniques

- Model checking
- Theorem proving
- Data flow analysis

Model Checking

- 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
An Example [3]

```java
import java.util.Random;
public class Rand {
    public static void main(String[] args) {
        Random random = new Random(42); // (i)
        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=");
    }
}
```

Is there any Divide-by-Zero error?

Model Checking

1. Random random = new Random()
2. int a = random.nextInt(2)
3. int b = random.nextInt(3)
4. int c = a/(b+a-2)

What if there is any loop?

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

Model microwave as a simple transition system

- 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

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

• From programs and specs to theorems
  – Verification condition generation
• From theorems to proofs
  – Theorem provers

• 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
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 < 40\} \rightarrow x := x + 1 \{x \leq 40\},

Precise Triples

- If \{P\} S \{Q \land R\} holds, then does \{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?

Precise Triples

- If \{P\} S \{R\} or \{Q\} S \{R\} hold, then does \{P \lor Q\} S \{R\} hold?
- Weakest preconditions
  
  - The most general preconditions \(P \lor Q\) is the “weakest” preconditions 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

```c
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

Weakest Precondition Rules

- \(WP(x := E, B) = B[E/x]\)
- \(WP(s1; s2, B) = WP(s1, WP(s2, B))\)
- \(WP(if E then s1 else s2, B) = (E \Rightarrow WP(s1, B)) \land (\neg E \Rightarrow WP(s2, B))\)
- \(WP(assert E, B) = E \land B\)
- What is the WP of our example program?
  
  \(WP(S) = (a \Rightarrow true = a) \land \neg a \Rightarrow b = a \lor b)\)
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
  \[ \text{out}_i = \text{trans}(\text{in}_i) \]
  \[ \text{in}_i = \text{join}_{\text{pred}}(\text{out}_i) \]
• Solve the equation set iteratively, until reaching a fixpoint: all in-states do not change
  \[ \text{for } i = 1 \text{ to } N \]
  \[ \text{initialize node } i \]
  \[ \text{while (sets are still changing) for } i = 1 \text{ to } N \]
  \[ \text{recompute sets at node } i \]

Work List Iterative Algorithm

\[ \text{for } i = 1 \text{ to } N \]
\[ \text{initialize node } i \]
\[ \text{add node } i \text{ to worklist} \]
\[ \text{while (worklist is not empty) remove a node } n \text{ from worklist} \]
\[ \text{calculate out-state based on in-state} \]
\[ \text{if out-state is different from the original value worklist = worklist U succ(n)} \]
An Example [7]

• What variable definitions reach the current program point?

```
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
  
  \[
  \begin{align*}
  \text{out}_b &= \text{gen}_b \cup (\text{in}_b - \text{kill}_b) \\
  \text{in}_b &= \bigcup_{p \in \text{pred}(b)} \text{out}_p
  \end{align*}
  \]

  - \( \text{gen}_b \): 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

```
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?

```
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
  
  $in_b = out_b - kill_b \cup gen_b$
  
  $out_b = \cup_{j \in succs} (in_j)$

- $gen_b$: variables used by $b$
- $kill_b$: if $v$ is defined without using $v$, all its prior definitions are killed
- initialization: $out = \emptyset$

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

---

1: $c = a + b$
2: $d = a * c$
3: $e = d * d$
4: $i = 1$
5: $f[i] = a + b$
6: $c = c * 2$
7: if $c < d$ goto 10
8: $g[i] = d * d$
9: goto 11
10: $g[i] = a * c$
11: $i = i + 1$
12: if $i <= 10$ goto 5

- What are the available expressions at 5?
- What direction is the analysis?
- How to define $gen_b$ and $kill_b$?
- What are the equations for $in_b$ and $out_b$?
Our analyses so far

<table>
<thead>
<tr>
<th>Backward</th>
<th>Forward</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reaching definitions</td>
<td>Available expressions</td>
</tr>
<tr>
<td>Live variables</td>
<td>union</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
  - $\forall x, y \in L, \exists$ unique $w$ and $z$ such that $x \cup y = w$ and $x \cap y = z$
  - $\forall x, y \in L, (x \cup y) \cup z = x \cup (y \cup z)$
  - $\exists \bot, \top \in L, \forall x \in L, x \cap \bot = \bot$ and $x \cup \top = \top$

Monotonic Functions
- The join and meet operators induce a partial order on the lattice elements
  - $x \leq y$ if and only if $x \cap y = x$
  - 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

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

- 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.

```
Entry main
x := 7
Call p(x)
Entry p
x < 9
y := 0
return y
Exit p
y := 1
Return p(x)
```

**Exponential Size Increase**

- How about recursive function calls?
  - \( p(\text{int } n) \{ \ldots p(n - 1) \ldots \} \)
- 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**

- Context insensitivity
- Context sensitive, 1-stack depth
- Context sensitive, 2-stack depth

**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 \( p \)?
  
  ```c
  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 \( d \)

Address Taken

- 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
  - \( p = \&x: x \in \text{pts-to}(p) \)
  - \( p = q: \text{pts-to}(p) = \text{pts-to}(q) \)
  - \( p = *q \quad \forall a \in \text{pts-to}(q), \text{pts-to}(p) = \text{pts-to}(a) \)
  - \( *p = q \quad \forall b \in \text{pts-to}(p), \text{pts-to}(b) = \text{pts-to}(q) \)

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

  - Points-to set: \( \text{pts}(a) = \text{pts}(f) = \{b, d\} \)

Andersen

- Subset Constraints
  - \( p = \&x: x \in \text{pts-to}(p) \)
  - \( p = q: \text{pts-to}(q) \subseteq \text{pts-to}(p) \)
  - \( p = *q \quad \forall a \in \text{pts-to}(q), \text{pts-to}(a) \subseteq \text{pts-to}(p) \)
  - \( *p = q \quad \forall b \in \text{pts-to}(p), \text{pts-to}(q) \subseteq \text{pts-to}(b) \)

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

  - Points-to set: \( \text{pts}(a) = \{b, d\} \), \( \text{pts}(f) = \{d\} \)
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 \(\text{pts}(x)\)
  - gen—add \(\text{pts}(y)\) to \(\text{pts}(x)\)
• \(*x = y\): 
  - If \(x\) definitely points to a single concrete memory location \(z\), \(\text{pts}(z) = y\) (strong update)
  - If \(x\) may point to multiple locations, then **weak update** by adding \(y\) to \(\text{pts}\) 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)