Program Static Analysis

Overview

- Program static analysis
- Abstract interpretation
- Data flow analysis
  - Intra-procedural
  - Inter-procedural

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

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

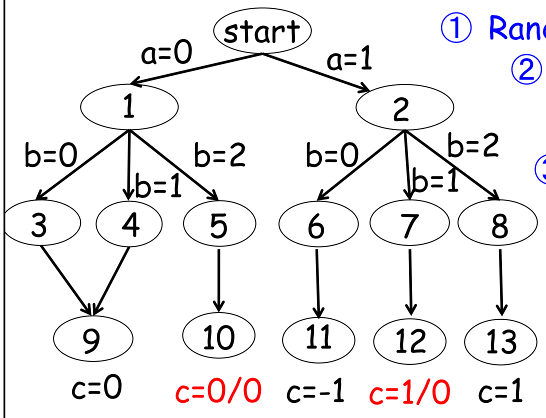
An Example [3]

```
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=" + a);
        ... ...
        int b = random.nextInt(3);          // (3)
        System.out.println("  b=" + b);

        int c = a/(b+a - 2);                // (4)
        System.out.println("    c=" + 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?

Limitations of Model Checking

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

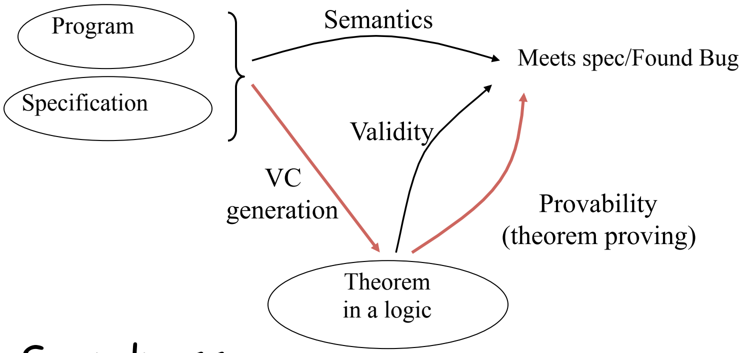
An axiomatic approach [4]

- Add auxiliary specifications to the program to decompose the verification task into a set of local verification tasks
- Verify each local verification problem

Limitations

- Auxiliary spec is burden on programmers
- Auxiliary spec might be incorrect
- If verification fails, is the problem with the auxiliary specification or the program?

Theorem Proving



Program

Specification

Semantics

Meets spec/Found Bug

Validity

VC generation

Provability (theorem proving)

Theorem in a logic

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

- From programs to theorems
  - Verification condition generation
- From theorems to proofs
  - Theorem provers

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

Hoare Triples [6]

- For any predicates P and Q and program S,

postcondition

$$\{P\}\ S\ \{Q\}$$

precondition

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} x := x+1 {x ≤ 40}

Precise Triples

- If {P} S {Q ∧ R} holds, then do {P} S {Q} and {P} S {R} hold?
- Strongest postcondition
  - The most precise postcondition (Q ∧ 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 v Q} S {R} hold?
- Weakest preconditions
  - The most general precondition {P v 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 v b (postcondition)
- Program

```
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 that if 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 => WP(s1, B)) ∧ (¬E => WP(s2, B))
- WP(assert E, B) = E ∧ B
- What is the WP of our example program?
  - – WP(S) = (a=>true=a∨b)∧(¬a=>b=a∨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

$$out_b = trans(in_b)$$

$$in_b = join_{p \in pred_b}(out_p)$$

- 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 ← 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)
```

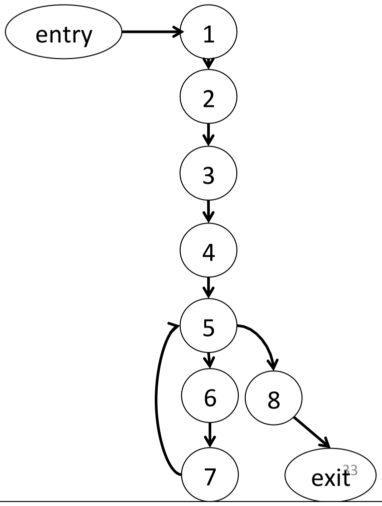
Directions of Data Flow Analysis

- 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?

```
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

$$out_b = gen_b \cup (in_b - kill_b)$$
$$in_b = \cup_{p \in pred_b}(out_p)$$

- gen_b: variable definitions generated by b
- $kill_b$: definitions killed at b by redefinitions of the variable(s)
- initialization: in = {}

Using Reaching Definition

- Constant propagation

```
int x = 5;
int y = 7;
int z = x + y;
int w = x – y;
```

- Detection of uninitialized variables

```
int x;
if (…)
   x = 1;
…
a = x;
```

Using Reaching Definition

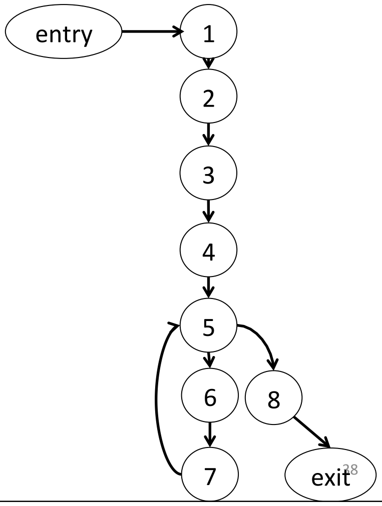
- Loop-invariant Code Motion
  - Consider an expression inside a loop. If all reaching definitions are outside of the loop, then move the expression out of the loop

---

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

$$in_b = out_b - kill_b \cup gen_b$$
$$out_b = \cup_{p \in succ_b}(in_p)$$

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

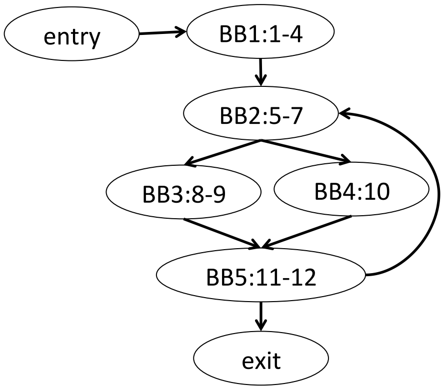
Using Liveness Analysis

- 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**

```
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?

Using Available Expressions

- Common-subexpression elimination

Our analyses so far

	union	intersection
forward	Reaching definitions	Available expressions
backward	Live variables	

Questions

- Does work list iterative algorithm always terminate?

Lattices

- A lattice L is a (possibly infinite) set of values, along with ∪ and ∩ operations
  - $\forall x, y \in L, \exists$ unique w and z such that $x \cup y = w$ and $x \cap y = z$
  - $\forall x, y \in L, x \cup y = y \cup x$ and $x \cap y = y \cap x$
  - $\forall x, y, z \in L, (x \cup y) \cup z = x \cup (y \cup z)$ and $(x \cap y) \cap z = x \cap (y \cap z)$
  - $\exists \bot, \mathrm{T} \in L,$ such that $\forall x \in L, x \cap \bot = \bot$ and $x \cup \mathrm{T} = \mathrm{T}$

Monotonic Functions

- The join and meet operators induce a **partial order** on the lattice elements
  - $x \subseteq y$ if and only if $x \cap y = x$
  - reflexive, anti-symmetric, transitive
- For a lattice L, a function f: L->L is **monotonic** if for all x, y $\in L$
  - $x \subseteq y \Rightarrow f(x) \subseteq f(y)$ or $x \subseteq y \Rightarrow f(x) \supseteq f(y)$

Reaching definition is monotonic

$$out_b = gen_b \bigcup (in_b - kill_b)$$

- Proof (for single-variable single-block programs) by contradiction:
  - Suppose $\mathrm{in}_b = \{1\}, out_b = \{0\}$, where 1 means there is a variable definition, 0 means no definition, then $gen_b = \{0\}, kill_b = \{1\}$.
  - However, $kill_b = \{1\}$ only if the block **b** has a redefinition of the variable, which means $gen_b = \{1\}$

- 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?

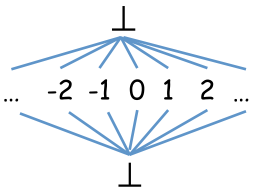
Constant Propagation

```
1: c = 3
2: d = 5
3: e = c + d
4: f = input()
5: if (f>0)
6:    e = 0
7: g = d + e
```

- What is the value of variables at 4 and 7?
- How do you define the data flow analysis?

Constant-propagation Analysis

- For a single-variable program
  - Direction: forward
  - Transfer function: $out_b = gen_b \cup (in_b - kill_b)$
  - Dataflow value: elements in CP-lattice
  - Meet operator(Join function): CP-lattice ∪
  - Initialization: ⊥



⊥ means "uninitialized variable"
⊥ means "not a constant"

Inter-procedural Analysis [8]

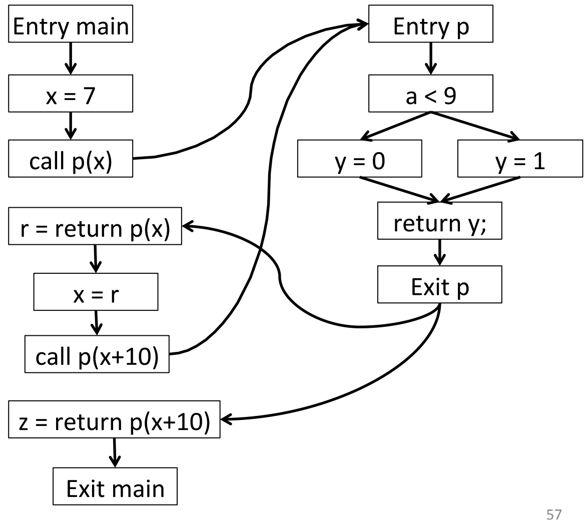
Stephen Chong
Imported by Na Meng

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

```
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;
}
```

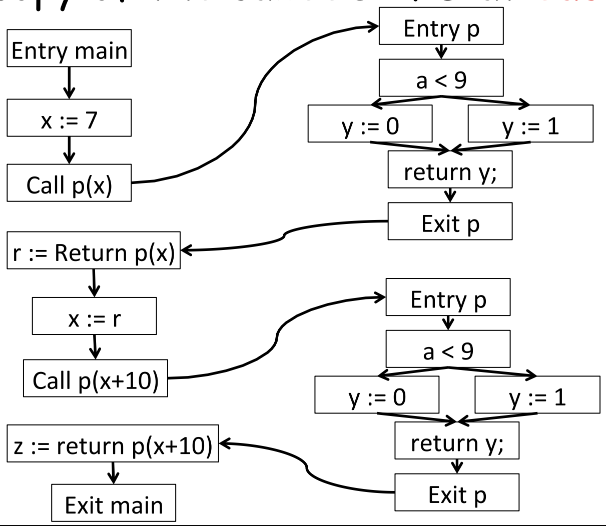


Imprecision

- 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



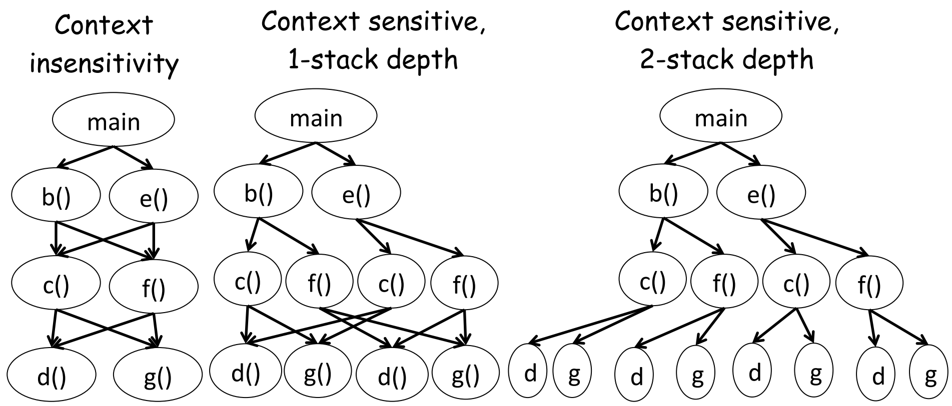
Exponential Size Increase

- 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, Anderson
- 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
```

$a \xrightarrow{1} b \xrightarrow{2} c$

$f \xrightarrow{4} d \xrightarrow{3} e$

- 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}

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

$a \xrightarrow{1} b \xrightarrow{2} c$

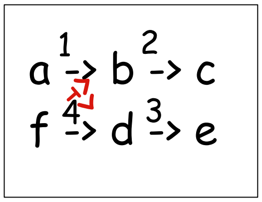
$f \xrightarrow{4} d \xrightarrow{3} e$

Steensgaard

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

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

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



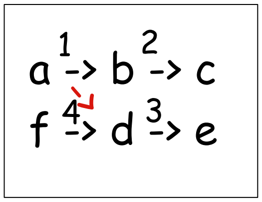
69

Andersen

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

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

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



70

Flow-sensitive Pointer Analysis

$$out_b = gen_b \bigcup (in_b - kill_b)$$

$$in_b = \bigcup_{p \in pred_b} (out_p)$$

- 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 **week update** by adding y to pts of all locations

Reference

[1] Static program analysis, https://en.wikipedia.org/wiki/Static_program_analysis
[2] Patrick Cousot, A Tutorial on Abstract Interpretation, http://homepage.cs.uiowa.edu/~tinelli/classes/seminar/Cousot--A%20Tutorial%20on%20AI.pdf
[3] Software Model Checking Example, http://javapathfinder.sourceforge.net/sw_model_checking.html
[4] Automated Theorem Proving, https://courses.cs.washington.edu/courses/cse599f/06sp/lectures/atp1.ppt
[5] Peter Lee, Classical Dataflow Optimizations, http://www.cs.cmu.edu/afs/cs/academic/class/15745-s06/web/handouts/04.pdf
[6] K. Rustan M. Leino, Hoare-style program verification, http://research.microsoft.com/en-us/um/people/leino/papers/cse503-Leino-Lecture0.ppt.

Reference

[7] Kathryn S. McKinley, Data Flow Analysis and Optimizations, http://www.cs.utexas.edu/users/mckinley/380C/lecs/03.pdf

[8] Stephen Chong, Interprocedural Analysis, http://www.seas.harvard.edu/courses/cs252/2011sp/slides/Lec05-Interprocedural.pdf