Announcements

- HW2 out
- Post question on
  - Setup
  - Starter code, generic class analysis framework and fixpoint iteration algorithm
  - Soot

Outline of Today’s Class

- Analysis scope and approximation
  - Class analysis, intro
  - Class Hierarchy Analysis (CHA)
  - Rapid Type Analysis (RTA)
  - Program analysis frameworks: Soot, Ghidra
  - The XTA analysis family (next time)
  - Other analyses: 0-CFA and PTA (next time)

Analysis Scope

- **Intraprocedural analysis**
  - Scope is the CFG of a single subroutine
  - Assumes no call and returns in routine, or models calls and returns
  - What we did so far

- **Interprocedural analysis**
  - Scope of analysis is the ICFG (Interprocedural CFG), which models flow of control between routines

- **Whole-program analysis**
  - Usually, assumes entry point “main”
  - Application code + libraries
    - Intricate interdependences, e.g., Android apps

- **Modular analysis**
  - Scope of analysis is the application code (or library) without any dependencies
  - ... or a library that depends on other missing libraries
Approximations
- Once we tackle the "whole program" maintaining a solution per program point (i.e., \(\text{in}(j)\) and \(\text{out}(j)\) sets) becomes too expensive
- Dimensions of approximation
  - Transfer function space
  - Lattice
  - Context sensitivity
  - Flow sensitivity

Context Sensitivity
- So far, we studied **intraprocedural analysis**
- Once we extend to **interprocedural analysis** the issue of "context sensitivity" comes up
- Interprocedural analysis can be context-insensitive or context-sensitive
  - Today, we'll see some context-insensitive analyses
  - Next week, a lot more on context-sensitive analysis

Context Insensitivity
- Context-insensitive analysis makes one big CFG; reduces the problem to standard dataflow, which we know how to solve
- Treats implicit assignment of actual-to-parameter and return-to-left_hand_side as explicit ones
  - E.g., \(x = \text{id}(y)\) where \(\text{id}\): \(\text{int id}(\text{int } p) \{ \text{return } p; \}\)
  - adds \(p = y\)
  - and \(x = \text{ret}\)
- Can be flow-sensitive or flow-insensitive

Flow Sensitivity
- **Flow-sensitive vs. flow-insensitive** analysis
- Flow-sensitive analysis maintains the CFG and computes a solution per each node in CFG (i.e. each program point)
  - Standard dataflow analysis is flow-sensitive
- For large programs, maintaining CFG and solution per program point does not scale

Flow Insensitivity
- Flow-insensitive analysis discards CFG edges and computes a **single solution** \(S\)
- A "declarative" definition, i.e., specification:
  - Least solution \(S\) of equations \(S = f(S) \lor S\)
- Points-to analysis is an example where such a solution makes sense
Flow Insensitivity

- An “operational” definition. A worklist-like algorithm:
  \[ S = f_1(\text{InitialValue}), \quad W = \{ 2, \ldots, n \} \] 
  while \( W \neq \emptyset \) do 
  remove \( j \) from \( W \) 
  \[ S = f_j(S) \cup S \] 
  if \( S \) changed then 
  \[ W = W \cup \{ k \mid k \text{ is a "successor" of } j \} \] 
  \}

- Note that “successors” here does not refer to successor nodes in the CFG, but nodes \( k \) whose transfer function \( f_k \) may contribute to \( S \) as a result of the change by \( j \).

Your Homework

- A bunch of flow-insensitive, context-insensitive analyses for Java
  - RTA, XTA, and optionally other
  - Simple property space
  - Simple transfer functions
    - E.g., in fact, RTA gets rid of most CFG nodes, processes just 2 kinds of nodes
  - Millions of lines of code in seconds

Outline of Today’s Class

- Analysis scope and approximation
- Class analysis, intro
- Class Hierarchy Analysis (CHA)
- Rapid Type Analysis (RTA)
- Program analysis frameworks: Soot, Ghidra
- The XTA analysis family (next time)
- Other analyses: 0-CFA and PTA (next time)

Class Analysis

- Problem statement: What are the classes of objects that a (Java) reference variable may refer to?
- Class Hierarchy Analysis (CHA)
- Rapid Type Analysis (RTA)
- XTA (next time)
- 0-CFA (next time)
- Points-to Analysis (PTA) (next time)

Applications of Class Analysis

- Call graph construction
  - At virtual call \( r.m() \), what methods may be called? (Assuming \( r \) is of static type \( A \).)

- Virtual call resolution
  - If analysis proves that a virtual call has a single target, it can replace it with a direct call
  - An OOPSLA ’96 paper by Holzle and Driesen reports that C++ programs spend 5% of their time in dispatch code. For “all virtual”, it is 14%
public class AndExp extends BoolExp {
    private BoolExp left;
    private BoolExp right;

    public AndExp(BoolExp left, BoolExp right) {
        this.left = left;
        this.right = right;
    }

    public boolean evaluate(Context c) {
        return left.evaluate(c) && right.evaluate(c);
    }
}

left: {Constant}
right: {OrExp}

public class OrExp extends BoolExp {
    private BoolExp left;
    private BoolExp right;

    public OrExp(BoolExp left, BoolExp right) {
        this.left = left;
        this.right = right;
    }

    public boolean evaluate(Context c) {
        return left.evaluate(c) || right.evaluate(c);
    }
}

left: {VarExp}
right: {VarExp}

main() {
    Context theContext;
    BoolExp x = new VarExp("X");
    BoolExp y = new VarExp("Y");
    BoolExp exp = new AndExp(
            new Constant(true), new OrExp(x, y));
    theContext.assign(x, true);
    theContext.assign(y, false);
    boolean result = exp.evaluate(theContext);
    exp: {AndExp}
}

At runtime, exp can refer to an object of class AndExp, but it cannot refer to objects of class OrExp, Constant or VarExp!

Class Hierarchy Analysis (CHA)

- Attributed to Dean, Grove and Chambers:
  - Jeff Dean, David Grove, and Craig Chambers, "Optimization of OO Programs Using Static Class Hierarchy Analysis", ECOOP’95
  - Simplest way of inferring information about reference variables: we simply look at class hierarchy

Class Hierarchy Analysis (CHA)

- In Java, if a reference variable r has type A, r can refer only to objects that are concrete subclasses of A. Denoted by $\text{SubTypes}(A)$
  - Note: refers to Java subtype, not true subtype
  - Note: $\text{SubTypes}(A)$ notation due to Tip and Palsberg (OOPSLA’00)
  - Given a virtual call site r.m(), we can find what methods may be called based on
    - the static type of r, and
    - class hierarchy information
Example

```java
public class A {
    public static void main() {
        A a;
        D d = new D();
        E e = new E();
        if (...) a = d; else a = e;
        a.m();
    }
}
```

```java
public class B extends A {
    public void foo() {
        G g = new G();
    }
}
```

... If no other creation sites or calls in the program

Example

```java
public class A {
    public static void main() {
        A a;
        D d = new D();
        E e = new E();
        if (...) a = d; else a = e;
        a.m();
    }
}
```

```java
public class B extends A {
    public void foo() {
        G g = new G();
    }
}
```

... RTA starts at main. Records that D and E are instantiated. At call a.m() looks at all CHA targets. Expands only into target C.m()! Never reaches B.foo(), never records G as being instantiated.

CHA as Reachability Analysis

R denotes the set of reachable methods

1. \( \text{main} \in R \) // meaning: add \text{main} to \( R \)
2. for each method \( m \) in \( R \), each virtual call \( y.n(z) \) in \( m \), each class \( C \) in \( \text{SubTypes}(\text{StaticType}(y)) \) and \( n' \), where \( n' = \text{resolve}(C,n) \) \( n' \in R \) (Practical concerns: must consider direct calls too!)

Rapid Type Analysis (RTA)

Due to Bacon and Sweeney

- David Bacon and Peter Sweeney, “Fast Static Analysis of C++ Virtual Function Calls”, OOPSLA ’96

- Expands on CHA

- Improves on CHA

- Expands calls only if it has seen an instantiated object of the appropriate type!
RTA

R is the set of reachable methods
I is the set of instantiated types

1. main ∈ R

2. for each method m in R and each new site new C in m
   C ∈ I

3. for each method m in R, each virtual call y.n(z) in m, each class C in SubTypes(StaticType(y)) ∩ I, and n', where n' = resolve(C,n)
   n' ∈ R

Comparison

class A {
  public :
    virtual int foo() { return 1; };
};
class B: public A {
  public :
    virtual int foo() { return 2; };
    virtual int foo(int i) { return i+1; };
};
void main() {
  B* p = new B;
  int result1 = p->foo(1);
  int result2 = p->foo();
  A* q = p;
  int result3 = q->foo();
}

CHA resolves result2 to B.foo(); however, it does not resolve result3.
RTA resolves result3 to B.foo() because only B has been instantiated.

Outline of Today’s Class

- Analysis scope and approximation
- Class analysis, intro
- Class Hierarchy Analysis (CHA)
- Rapid Type Analysis (RTA)
- Program analysis frameworks: Soot, Ghidra
- The XTA analysis family (next time)
- Other analyses: 0-CFA and PTA (next time)

Soot: a framework for analysis and optimization of Java/Dalvik bytecode

- https://sable.github.io/soot/
- History
- Overview of Soot
  - From Java bytecode/Dalvik bytecode to typed 3-address code (Jimple)
  - 3-address code analysis and optimization
  - From Jimple to Java/Dalvik
- Jimple
- Analysis
Overview of Soot

Class files/APK

JIMPLIFY

ANALYSIS/OPTIMIZATION

Optimized jimple

Some IR

Class files/APK

Advantages of Jimple and Soot

- Jimple
  - Typed local variables
  - 16(!) simple 3-address statements (1 operator per statement). Bridges gap between analysis abstraction to analysis implementation
- Soot provides
  - Itraprocedural dataflow analysis framework
  - Points-to analysis
  - Context-sensitive analysis framework
  - Android taint analysis

Jimple

Run soot: java soot.Main –jimple A (need paths)

public class A extends java.lang.Object {
  public void <init>() {
    A r0;
    r0 := @this: A;
    specialinvoke r0.<java.lang.Object: void <init>()>();
    return;
  }
  ...
}

... (continues on next slide)

Java:

public class A {
  main(String[] args) {
    A a = new A();
    a.m();
  }
  public void m() {
  }
}

Jimple:

public class A {
  main(String[] args) {
    A a = new A();
    a.m();
  }
  public void m() {
  }
}

... (continues on next slide)

Soot Abstractions. Look up API!

- Abstracts program constructs
- Some basic Soot classes and interfaces
  - SootClass
  - SootMethod
    - SootMethod sm; sm.isMain(), sm.isStatic(), etc.
  - Local
    - Local l; ... l.getType()
  - InstancelInvokeExpr
    - Represents an instance (as opposed to static) invoke expression
    - InstancelInvokeExpr iie; ... receiver = iie.getBase();
4 Kinds of Calls

- Constructor/Super Call:
  ```java
  A a = new A();
  $r1 = new A; specialinvoke $r1.<A: void <init>()>();
  a.m();
  virtualinvoke r2.<A: void m>();
  ```

- Virtual Call:
  ```java
  sm();
  staticinvoke <A: void sm>();
  ```

- Static Call:
  ```java
  interfaceinvoke r0.<pack2.X: void m>();
  ```

- Interface Call:
  ```java
  x.m();
  ```

1. We should not need to worry about dynamicInvoke. (Soot does support it.)

An Overview of Homework

- Syntax
  - Assignment stmt: x = y
  - Field read stmt: x = y.f
  - Field write stmt: x.f = y
  - Array read stmt: x = y[i]
  - Array write stmt: x[i] = y
  - Allocation stmt: x = new A;
  - Direct call: x = sm(args) or x = y.m(args)
  - Virtual call: x = y.m(args)

For RTA, we only care about the last 3