Dataflow Analysis: Class Analysis

Announcements
- HW2 out
- I’ve added a post on the forum on some common setup issues
- Post question on
  - Setup
  - Starter code, generic analysis framework and fixpoint iteration algorithm
  - Soot

Outline of Today’s Class
- Analysis scope and approximation
  - Class analysis
  - Class Hierarchy Analysis (CHA)
  - Rapid Type Analysis (RTA)
  - The XTA analysis family
  - Other analyses: 0-CFA and PTA (next time)
- A brief overview of Soot

Outline of Today’s Class
- Reading
  - Jeff Dean, David Grove, and Craig Chambers, “Optimization of OO Programs Using Static Class Hierarchy Analysis”, ECOOP’95
  - David Bacon and Peter Sweeney, “Fast Static Analysis of C++ Virtual Function Calls”, OOPSLA ’96
  - Frank Tip and Jens Palsberg, “Scalable Propagation-Based Call Graph Construction Algorithms”, OOPSLA ’00

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

Analysis Scope
- Whole-program analysis
  - Usually, assumes entry point “main”
  - Application code + libraries
    - Intricate interdependences, e.g., Android apps
- Modular analysis
  - Scope either a library without entry point
  - or application code with missing libraries
  - … or a library that depends on other missing libraries
Approximations

- Once we tackle the "whole program" maintaining a solution per program point (i.e., in(j) and 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 = id(y) where id: int id(int p) { return p; }
    adds p = y and x = 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:

\[
S = 0, W = \{1, 2, \ldots, n\} /* all nodes */ \\
\text{while } W \neq \emptyset \text{ do } \\
\text{remove } j \text{ from } W \\
S = f_j(S) \cup S \\
\text{if } S \text{ changed then} \\
W = W \cup \{k | k \text{ is “successor” of } j\} \\
\]

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

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
- 0-CFA (next time)
- Points-to Analysis (PTA) (next time)

Boolean Expression Hierarchy

```java
public abstract class BoolExp { 
    public boolean evaluate(Context c); 
}

public class Constant extends BoolExp { 
    private boolean constant; 
    public boolean evaluate(Context c) { 
        return constant; 
    } 
}

public class VarExp extends BoolExp { 
    private String name; 
    public boolean evaluate(Context c) { 
        return c.lookup(name); 
    } 
}

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);
    }
}

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, simply look at class hierarchy!

In Java, if a reference variable r has type A, r can refer only to objects that are concrete subclasses of A. Denoted by SubTypes(A)

- Note: refers to Java subtype, not true subtype
- Note: SubTypes(A) notation due to Tip and Palsberg (OOPSLA'00)
- At virtual call site r.m(), we can find what methods may be called based on the 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();
    }
}
public class B extends A {
    public void foo() {
        G g = new G();
    }
}...
```

...// no other creation sites or calls in the program
Example

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();
    }
}

Example

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();
    }
}

Example

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();
    }
}

CHA as Reachability Analysis

- \( R \) denotes the set of reachable methods

1. \( \text{main} \in R \)
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
- Improves on CHA
- Expands calls only if it has seen an instantiated object of the appropriate type!

Example

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();
    }
}

Example

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();
    }
}

Example

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();
    }
}

RTA

- \( R \) is the set of reachable methods
- \( I \) is the set of instantiated types

1. \( \text{main} \in R \)
2. For each method \( m \in R \) and each new site \( \text{new C} \) in \( m \)
   \( C \in I \)
3. for each method \( m \in R \), each virtual call \( y.n(z) \) in \( m \), each class \( C \) in \( \text{SubTypes} (\text{StaticType}(y)) \cap I \), and \( n' \), where \( n' = \text{resolve}(C, n) \), add \( n' \) to \( R \).
Practical Concerns

- Multiple parameters
- Direct calls
  - either static invoke calls or special invoke calls
- Array reads and writes!
- Static fields

- See Tip and Palsberg for more

The Big Picture

- All fit into our monotone dataflow framework!
- Flow-insensitive, context-insensitive
- Least solution of \( S = f(S) \lor S \)
- Algorithms differ mainly in “size” of \( S \)
- RTA: only 2 kinds of statements; Lattice?
- XTA: expands to all statements; Lattice?
- 0-CFA: all statements; Lattice?
- PTA (Points-to analysis): all statements; Lattice elements are points-to graphs

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

History

- https://sable.github.io/soot/
- Started by Prof. Laurie Hendren at McGill
  - First paper on Soot came in 1999
  - Patrick Lam
  - Ondřej Lhoták
  - Eric Bodden
  - and other...
- Now actively developed by Eric Bodden and his group
Advantages of Jimple and Soot

- **Jimple**
  - Typed local variables
  - 16(!) simple 3-address statements (1 operator per statement). Bridges gap from analysis abstraction to analysis implementation

- **Soot** provides
  - Intraprocedural dataflow analysis framework
  - Points-to analysis
  - Context-sensitive analysis framework
  - Android taint analysis

### Jimple

- **Run soot:** `java soot.Main -jimple A (need paths)`

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

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

### 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()
  - **InstanceInvokeExpr**
    - Represents an instance (as opposed to static) invoke expression
    - `InstanceInvokeExpr iie; ... receiver = iie.getBase();`

### 4 Kinds of Calls

- **Constructor/Super Call:**
  - `A a = new A();` → `sr1 = new A; specialinvoke sr1.<A: void <init>()>();`

- **Virtual Call:**
  - `a.m();` → `virtualinvoke sr2.<A: void m()>()`

- **Static Call:**
  - `sm();` → `staticinvoke A: void sm();`

- **Interface Call:**
  - `x.m();` → `interfaceinvoke r0.<pack2.X: void m()>()`

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

- Catch up: points-to analysis
- Interprocedural Analysis
- Context sensitivity