Dataflow Analysis: Classical Analysis for Object-oriented Programs
Announcements

- Quiz 2 today

- HW 2 out
  - We’ve added some posts covering some common setup issues
  - Post question on
    - Setup, please do set this up as soon as possible!
    - Starter code, class analysis framework and worklist algorithm
    - Soot
Outline of Today’s Class

- Analysis scope and approximation
- Class analysis
- Class Hierarchy Analysis (CHA)
- Rapid Type Analysis (RTA)

- Class analysis framework

- The XTA analysis family (next time)
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

- **Intra**procedural analysis
  - Scope is the CFG of a single routine
  - Assumes no calls/returns in routine, or modeling of calls/returns
  - What we did so far

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

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

- **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., \texttt{in(j)} and \texttt{out(j)} sets) becomes too expensive.

- Approximations
  - Transfer function space
  - Lattice
  - Context sensitivity
  - Flow 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

- In our Java homework, we’ll see some context-insensitive analyses
- Next week we’ll talk more about 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 assignment.

- E.g., \( x = \text{id}(y) \) where \( \text{id}: \) int \( \text{id}(\text{int } p) \) \{ return \( p; \) \} adds \( p = y \) // flow of values from arg to param and \( x = \text{ret} \) // flow of return to left_hand_side

Can be flow-sensitive or flow-insensitive.
int id(int p) {
    return p;
}

a = 5;
2: b = id(a);
x = b*b;
c = 6;
5: d = id(c);
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 $S = f_j(S) \lor S$

- Points-to analysis is an example where such a solution makes sense!
Flow Insensitivity

An “operational” definition. A worklist-like algorithm:

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

“successor” is not CFG successor nodes, but more generally, nodes \( k \) whose transfer function \( f_k \) may be affected as a result of the change in \( S \) by \( j \).
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 (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 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);
    }
}

left: {VarExp}
right: {VarExp}
A Client of the Boolean Expression Hierarchy

```java
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);
}
```

At runtime, `exp` can refer to an object of class AndExp, but it cannot refer to objects of class OrExp, Constant or VarExp!
Call Graph Example (Partial)

main
  \(\text{exp.evaluate}\)
  AndExp.evaluate
    \(\text{left.evaluate}\)
    \(\text{right.evaluate}\)
    \(\text{Constant.evaluate}\)
    \(\text{OrExp.evaluate}\)
      \(\text{left.evaluate}\)
      \(\text{right.evaluate}\)
      \(\text{VarExp.evaluate}\)
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 --- just 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 \( \text{SubTypes}(A) \)

- Note: refers to Java subtype, not true subtype
- Note: \( \text{SubTypes}(A) \) 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
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
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();
    }
}

public class C {

}

Example

SubTypes(A) = { A, B, C, D, E, G }
SubTypes(B) = { B, G }
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();
    }
}

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();
    }
}

Example

```

\[a: \text{SubTypes(StaticType(a))} = \text{SubTypes(A)} = \{A, B, C, D, E, G\}\]
CHA as Reachability Analysis

$\mathbf{R}$ denotes the set of reachable methods

1. $\text{main} \in \mathbf{R}$

2. for each method $\mathbf{m} \in \mathbf{R}$, each virtual call $\text{y.n}(\text{z})$ in $\mathbf{m}$, each class $\mathbf{C}$ in $\text{SubTypes} (\text{StaticType}(\text{y}))$ and $\mathbf{n'}$, where $\mathbf{n'} = \text{resolve}(\mathbf{C}, \mathbf{n})$

   $\mathbf{n'} \in \mathbf{R}$

   (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

```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();
    }
}
```

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.
RTA

\( R \) is the set of \textit{reachable methods}
\[ I \] is the set of \textit{instantiated types}

1. \( \text{main} \in R \)

2. for each method \( m \in R \) and each \textit{new site} 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 \( \text{SubTypes(StaticType}(y) \rangle \cap I \), and \( n' \), where \( n' = \text{resolve}(C,n) \)

\[ n' \in R \]
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
- Class Hierarchy Analysis (CHA)
- Rapid Type Analysis (RTA)
- Class analysis framework
- The XTA analysis family (next time)
Class Analysis Framework
The End