Advanced Program Analyses for Object-oriented Systems

Dr. Barbara G. Ryder
Rutgers University
http://www.cs.rutgers.edu/~ryder
http://prolangs.rutgers.edu/
July 2007

Lecture 2 - Outline

- Type-based call graph construction
- Dimensions of analysis precision
  - Properties and characterizations of an analysis
  - Choices and consequences (precision & cost)
- Reference analysis of OO programs
  - Points-to analyses
    - Flow sensitivity, context sensitivity, field sensitivity
  - Client analyses: Side effects, finding thread/method-local objects, synchronization removal, efficient heap layout
Running Example

cf Tip & Palsberg, OOPSLA’00

```
static void main(){
    B b1 = new B();
    A a1 = new A();
    f(b1);
    g(b1);
}
static void f(A a2){
    a2.foo();
}
static void g(B b2){
    B b3 = b2;
    b3 = new C();
    b3.foo();
}
```

class A {
    foo(){..}
}
class B extends A{
    foo() {...}
}
class C extends B{
    foo() {...}
}
class D extends B{
    foo() {...}
}

CHA Example

```
static void main(){
    B b1 = new B();
    A a1 = new A();
    f(b1);
    g(b1);
}
static void f(A a2){
    a2.foo();
}
static void g(B b2){
    B b3 = b2;
    b3 = new C();
    b3.foo();
}
```

class A {
    foo(){..}
}
class B extends A{
    foo() {...}
}
class C extends B{
    foo() {...}
}
class D extends B{
    foo() {...}
}

Cone(Declared_type(receiver))
**CHA Example - Call Graph**

```java
class A {
    foo() { .. }
}
class B extends A {
    foo() { .. }
}
class C extends B {
    foo() { .. }
}
class D extends B {
    foo() { .. }
}

static void main() {
    B b1 = new B();
    A a1 = new A();
    f(b1);
    g(b1);
}
static void f(A a2) {
    a2.foo();
}
static void g(B b2) {
    B b3 = b2;
    b3 = new C();
    b3.foo();
}
```

**CHA Characteristics**

- Ignores program flow
- Calculates types that a reference variable can point to
- Uses 1 abstract reference variable per class throughout program
- Uses 1 abstract object to represent all possible instantiations of a class

---

J. Dean, D. Grove, C. Chambers, *Optimization of OO Programs Using Static Class Hierarchy*, ECOOP'95
RTA Example

cf Tip & Palsberg, OOPSLA’00

```java
static void main(){
    B b1 = new B();
    A a1 = new A();
    f(b1);
    g(b1);
}
static void f(A a2){
    a2.foo();
}
static void g(B b2){
    B b3 = b2;
    b3 = new C();
    b3.foo();
}

class A {
    foo(){..}
}
class B extends A{
    foo() {..}
}
class C extends B{
    foo() {..}
}
class D extends B{
    foo() {..}
}
```

Call Graph
RTA Characteristics

- Only analyzes methods reachable from main(), on-the-fly
- Ignores classes which have not been instantiated as possible receiver types
- Uses 1 abstract reference variable per class throughout program
- Uses 1 abstract object to represent all possible instantiations of a class

D. Bacon and P. Sweeney, "Fast Static Analysis of C++ Virtual Function Calls", OOPSLA'96

Experimental Comparison
C++ Programs

Bacon and Sweeney, OOPSLA'96

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Lines</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>sched</td>
<td>5,712</td>
<td>RS/6000 Instruction Timing Simulator</td>
</tr>
<tr>
<td>ixx</td>
<td>11,157</td>
<td>IDL specification to C++ stub-code translator</td>
</tr>
<tr>
<td>lcom</td>
<td>17,278</td>
<td>Compiler for the “L” hardware description language</td>
</tr>
<tr>
<td>hotwire</td>
<td>5,335</td>
<td>Scriptable graphical presentation builder</td>
</tr>
<tr>
<td>simulate</td>
<td>6,672</td>
<td>Simula-like simulation class library and example</td>
</tr>
<tr>
<td>idi</td>
<td>30,288</td>
<td>SunSoft IDL compiler with demo back end</td>
</tr>
<tr>
<td>taldict</td>
<td>11,854</td>
<td>Taligent dictionary benchmark</td>
</tr>
<tr>
<td>deltablue</td>
<td>1,250</td>
<td>Incremental dataflow constraint solver</td>
</tr>
<tr>
<td>richards</td>
<td>606</td>
<td>Simple operating system simulator</td>
</tr>
</tbody>
</table>

Table 1: Benchmark Programs. Size is given in non-blank lines of code
Data Characteristics
Bacon and Sweeney, OOPSLA'96

- Frequency of execution matters
  - Direct calls were 51% of static call sites but only 39% of dynamic calls
  - Virtual calls were 21% of static call sites but were 36% of dynamic calls

- Results they saw differed from previous studies of C++ virtuals
  - Stresses importance of benchmarks in empirical work

Findings
Bacon and Sweeney, OOPSLA'96

- RTA was better than CHA on virtual function resolution, but not on reducing code size
  - Inference is that call graphs constructed have same node set but not same edge set!

- Claim both algorithms cost about the same because the dominant cost is traversing the CFGs of methods and identifying call sites
  - Implemented CHA with reachability calculation

- Claim that RTA is good enough for call graph construction so that more precise analyses are not necessary for this task
Type-based vs Flow-based Reference Analysis

- Uses only class hierarchy
- Same points-to set for every reference of a type
- Always flow- and context- insensitive
- Inexpensive
- Okay for call graph construction, but too imprecise for some other applications

- Uses reference assignments
- Distinguishes points-to sets of different references of same type
- Can be flow/context-sensitive or insensitive
- May be expensive
- Related to points-to approaches for C
- Okay for side-effect and dependence calculations

Dimensions of Precision

- Independent characteristics of a reference analysis which determines its precision
- Different combinations of these dimensions have already been explored in algorithms
- Need to understand what choices are available to design new analyses of appropriate precision for clients

Dimensions of Precision

1. Program representation - Call graph
   - Use type-based approximation
   - Lazy, on-the-fly construction
     - Only explore methods which are statically reachable from the main()
     - Especially important for OOPLs use of libraries

Dimensions of Precision

2. Object Representation
   - Use one abstract object per class (CHA, RTA)
   - Group object instantiations by creation site
   - Finer-grained object naming

3. Field Sensitivity
   - May or may not distinguish fields of objects; field-sensitive, field-based, field-insensitive
Field Sensitivity

• Field-insensitive
  - Does not distinguish between fields of an abstract object

• Field-based
  - Collapses all same-named fields into an abstract representative

• Field-sensitive
  - Distinguishes between different fields of an abstract object

Spark Experiments

• Precision measure incorporated unreachable dereferences and unique object reference targets
  - Precision of fb:fs was 57.7:60.0 on average
  - Time cost was very similar
  - Space cost of fb:fs was 86.6:138.4 on average

• Lesson learned: sometimes less precision is okay - need to know the client of the points-to info

Lhotak and Hendren, “Scaling Java Points-to Analysis Using SPARK”, CC’03
Dimensions of Precision

4. Reference representation
- Use one abstract reference per class (CHA, RTA)
- Use one abstract reference for each class per method (XTA)
- Represent reference variables or fields by their names program-wide (FieldSens)

5. Flow sensitivity
- Analyses which capture the sequential order of execution of program statements

1. A s,t;
2. s = new A(); //o₁
3. t = s;
4. s = new A(); //o₂

flow-sensitive:
at 2., s refers to o₁
at 3., s,t refer to o₁
at 4., s refers to o₂
t refers to o₁
flow-insensitive:
s,t refer to {o₁ o₂}
Dimensions of Precision

6. Context sensitivity
- Analyses which distinguish different calling contexts of the same method
- Differ by how they represent calling context
  - Call string
  - Functional approach
- 1-CFA, example of call string approach
- ObjSens, example of functional approach

Dimensions of Precision

7. Directionality
- How flow in reference assignments (r=s) is interpreted by the analysis
  - Symmetric (Unification): r and s have same points-to set after the assignment
  - Directional (Inclusion): r's points-to set includes s's points-to set after the assignment
Reference Analyses

• Vary according to the choices taken in each of the dimensions
• Flow-based analyses based on ideas from points-to analysis of C pointers
• XTA: a flow-based analysis close to type-based analysis in cost, but with better precision because it incorporates flow of types into methods

XTA Analysis

• Calculates set of classes that reach a method, incorporating (limited) flow
• Uses an on-the-fly constructed call graph
• Uses one abstract object per class with distinct fields (field-sensitive)
• Uses one abstract reference per class in each method

Tip and Palsberg, “Scalable Propagation-based Call Graph Construction Algorithms”, OOPSLA’00
Example of XTA

```java
{A,B}
class A {
  foo() {...
}
}
class B extends A{
  foo() {...
}
}
class C extends B{
  foo() {...
}
}
class D extends B{
  foo() {...
}
}

static void main(){
  B b1 = new B();
  A al = new A();
  f(b1);
  g(b1);
}

static void f(A a2){
  a2.foo();
}

static void g(B b2){
  B b3 = b2;
  b3 = new C();
  b3.foo();
}
```

Java Program Dataset

cf Tip & Palsberg, OOPSLA'00

<table>
<thead>
<tr>
<th>benchmark</th>
<th># classes</th>
<th># methods</th>
<th>#fields (reference-typed)</th>
<th># virtual call sites</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hanoi</td>
<td>44</td>
<td>373</td>
<td>520 (107)</td>
<td>235</td>
</tr>
<tr>
<td>Ice Browser</td>
<td>76</td>
<td>761</td>
<td>590 (253)</td>
<td>922</td>
</tr>
<tr>
<td>mBird</td>
<td>2050</td>
<td>17,945</td>
<td>6739 (4284)</td>
<td>3,259</td>
</tr>
<tr>
<td>Cindy</td>
<td>468</td>
<td>4,449</td>
<td>3075 (1677)</td>
<td>5,635</td>
</tr>
<tr>
<td>CindyApplet</td>
<td>468</td>
<td>4,449</td>
<td>3075 (1677)</td>
<td>5,635</td>
</tr>
<tr>
<td>Suite Sheet</td>
<td>588</td>
<td>5,590</td>
<td>4326 (1412)</td>
<td>4,450</td>
</tr>
<tr>
<td>eSuite Chart</td>
<td>733</td>
<td>8,302</td>
<td>5448 (2141)</td>
<td>8,074</td>
</tr>
<tr>
<td>javaReg 1.43</td>
<td>161</td>
<td>2,108</td>
<td>1525 (971)</td>
<td>3,482</td>
</tr>
<tr>
<td>B.OAT</td>
<td>282</td>
<td>2,677</td>
<td>1255 (541)</td>
<td>6,663</td>
</tr>
<tr>
<td>JAX 8.3</td>
<td>308</td>
<td>2,754</td>
<td>1252 (579)</td>
<td>3,836</td>
</tr>
<tr>
<td>javac</td>
<td>210</td>
<td>1,512</td>
<td>1107 (406)</td>
<td>3,651</td>
</tr>
<tr>
<td>Res. System</td>
<td>2,332</td>
<td>21,495</td>
<td>12487 (3334)</td>
<td>23,640</td>
</tr>
</tbody>
</table>
Findings

• Measures precision improvements over RTA
  - Given that reference r can point to an RTA-calculated set of types program-wide, then XTA reduces the size of this set by 88%, on average, per method.
• The reachable methods set (i.e., call graph nodes) is minimally reduced over that of RTA
• The number of edges in the call graph is significantly reduced by XTA over RTA (.3%-29% fewer, 7% on average)

Findings, cont.

• Data gives comparison restricted to those calls that RTA found to be polymorphic and how these analyses can improve on that finding.
  - Claim that the reduction in edges are for those calls that RTA found to be polymorphic, and often call sites become monomorphic after analysis
• Bottom line: Improved call graph construction
Data of Findings

similar cf Tip & Palsberg, OOPSLA’00

better

<table>
<thead>
<tr>
<th>benchmark</th>
<th>reached</th>
<th>RTA mono</th>
<th>poly</th>
<th>reached</th>
<th>XTA mono</th>
<th>poly</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hanoi</td>
<td>34.0%</td>
<td>61.6%</td>
<td>4.4%</td>
<td>34.0%</td>
<td>62.7%</td>
<td>3.3%</td>
</tr>
<tr>
<td>ice Browser</td>
<td>4.0%</td>
<td>91.4%</td>
<td>4.7%</td>
<td>4.0%</td>
<td>91.6%</td>
<td>4.5%</td>
</tr>
<tr>
<td>mBird</td>
<td>14.2%</td>
<td>73.4%</td>
<td>12.3%</td>
<td>17.4%</td>
<td>70.9%</td>
<td>11.7%</td>
</tr>
<tr>
<td>Cindy</td>
<td>49.3%</td>
<td>45.0%</td>
<td>5.7%</td>
<td>49.4%</td>
<td>45.5%</td>
<td>5.0%</td>
</tr>
<tr>
<td>CindyApplet</td>
<td>72.0%</td>
<td>24.6%</td>
<td>3.4%</td>
<td>72.3%</td>
<td>24.5%</td>
<td>3.2%</td>
</tr>
<tr>
<td>eSuite Sheet</td>
<td>28.1%</td>
<td>68.4%</td>
<td>3.5%</td>
<td>28.2%</td>
<td>69.1%</td>
<td>2.8%</td>
</tr>
<tr>
<td>eSuite Chart</td>
<td>13.3%</td>
<td>76.6%</td>
<td>10.1%</td>
<td>15.7%</td>
<td>78.0%</td>
<td>8.3%</td>
</tr>
<tr>
<td>javaFig 1.43</td>
<td>9.1%</td>
<td>87.1%</td>
<td>3.9%</td>
<td>9.7%</td>
<td>87.2%</td>
<td>3.1%</td>
</tr>
<tr>
<td>BLOAT</td>
<td>6.6%</td>
<td>82.4%</td>
<td>11.1%</td>
<td>7.0%</td>
<td>82.2%</td>
<td>10.8%</td>
</tr>
<tr>
<td>JAX 6.3</td>
<td>18.7%</td>
<td>75.9%</td>
<td>5.4%</td>
<td>18.9%</td>
<td>76.8%</td>
<td>4.3%</td>
</tr>
<tr>
<td>javac</td>
<td>3.0%</td>
<td>77.6%</td>
<td>19.4%</td>
<td>3.0%</td>
<td>77.7%</td>
<td>19.3%</td>
</tr>
<tr>
<td>Res System</td>
<td>18.1%</td>
<td>72.0%</td>
<td>9.9%</td>
<td>18.2%</td>
<td>74.0%</td>
<td>7.9%</td>
</tr>
<tr>
<td>AVERAGE</td>
<td>7.8%</td>
<td>7.0%</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Fully Incorporating Flow

- Need to incorporate intraprocedural flow of data into and out of reference variables (i.e., assignments)
- Model parameter / argument associations as assignments
- FieldSens based on Andersen’s points-to analysis for C
  - Flow-insensitive, context-insensitive, field-sensitive, inclusion constraints
Rules of Algorithm

- 4 types of reference assignment statements with points-to effects
  - Allocation: p=new X()
    - Adds o_x to Pts(p)
  - Copy: p = q
    - Pts(p) ⊇ Pts(q) (i.e., If o ∈ Pts(q), then o ∈ Pts(p))
  - Field store: p.f = q
    - If o ∈ Pts(p) and r ∈ Pts(q), then r ∈ Pts(o.f)
  - Field load: p = q.g
    - If o ∈ Pts(q) and oo ∈ Pts(o.g) then oo ∈ Pts(p)

Points-to Graph

- Nodes represent reference variables and fields, or objects
- Edges represent possible run-time points-to relations
  - Sometimes labeled with object field names
- Uses: call graph construction, side effect analysis, memory optimizations
```java
static void main(){
    B b1 = new B();
    A a1 = new A();
    f(b1);
    g(b1);
}
static void f(A a2){
    a2.foo();
}
static void g(B b2){
    B b3 = b2;
    b3 = new C();
    b3.foo();
}
```

Points-to Graph summarizes reference/object relationships

cf Tip & Palsberg, OOPSLA’00

```
static void main(){
    B b1 = new B();
    A a1 = new A();
    f(b1);
    g(b1);
} 
static void f(A a2){
    a2.foo();
}
static void g(B b2){
    B b3 = b2;
    b3 = new C();
    b3.foo();
}
```

class A {
    foo(){..}
}

```java
class B extends A{
    foo() {..}
}
```
FieldSens Characteristics

- Only analyzes methods reachable from main()
- Keeps track of individual reference variables and fields
- Groups objects by their creation site
- Incorporates reference value flow in assignments and method calls

Rountev, A. Milnova, B. Ryder, "Points-to Analysis for Java Using Annotated Constraints", OOPSLA'00;
Lhotak and Hendren, "Scaling Java Points-to Analysis using SPARK", CC'03

Clients of FieldSens

Rountev et. al, OOPSLA'00

- Devirtualization
- Side effect analysis
  - What objects can have their values changed through a reference assignment?
- Memory optimizations
  - What objects can be stored on the stack frame (local to a method, or thread)?
### Benchmarks Used

**Rountev et al., OOPSLA’00**

<table>
<thead>
<tr>
<th>Program</th>
<th>User Class</th>
<th>Size (Kb)</th>
<th>Whole-program Method</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>proxy</td>
<td>38</td>
<td>66</td>
<td>563</td>
<td>32813.0</td>
</tr>
<tr>
<td>compress</td>
<td>22</td>
<td>76.1</td>
<td>568</td>
<td>3344.0</td>
</tr>
<tr>
<td>db</td>
<td>14</td>
<td>70.7</td>
<td>565</td>
<td>3343.0</td>
</tr>
<tr>
<td>jts-6.1</td>
<td>21</td>
<td>53.6</td>
<td>574</td>
<td>3343.0</td>
</tr>
<tr>
<td>echo</td>
<td>17</td>
<td>86.5</td>
<td>577</td>
<td>3344.0</td>
</tr>
<tr>
<td>ratrace</td>
<td>35</td>
<td>115.9</td>
<td>582</td>
<td>3444.0</td>
</tr>
<tr>
<td>nttt</td>
<td>35</td>
<td>115.9</td>
<td>582</td>
<td>3444.0</td>
</tr>
<tr>
<td>jtrace-1.21</td>
<td>64</td>
<td>185.2</td>
<td>618</td>
<td>3583.0</td>
</tr>
<tr>
<td>jtrace-1.2</td>
<td>64</td>
<td>185.2</td>
<td>618</td>
<td>3583.0</td>
</tr>
<tr>
<td>jtrace-1.2</td>
<td>64</td>
<td>185.2</td>
<td>618</td>
<td>3583.0</td>
</tr>
<tr>
<td>javaup0.10</td>
<td>33</td>
<td>127.4</td>
<td>583</td>
<td>3564.0</td>
</tr>
<tr>
<td>rabbit-2</td>
<td>62</td>
<td>187.4</td>
<td>615</td>
<td>3579.0</td>
</tr>
<tr>
<td>jack</td>
<td>67</td>
<td>191.5</td>
<td>613</td>
<td>3574.0</td>
</tr>
<tr>
<td>jflex-1.2</td>
<td>34</td>
<td>188.5</td>
<td>608</td>
<td>3692.0</td>
</tr>
<tr>
<td>emacs</td>
<td>169</td>
<td>414.8</td>
<td>415</td>
<td>3973.0</td>
</tr>
<tr>
<td>mpgauto</td>
<td>62</td>
<td>170.8</td>
<td>600</td>
<td>3534.0</td>
</tr>
<tr>
<td>jtrace-1.0</td>
<td>72</td>
<td>272.0</td>
<td>620</td>
<td>6978.0</td>
</tr>
<tr>
<td>saberc2.9</td>
<td>312</td>
<td>492.3</td>
<td>863</td>
<td>5154.0</td>
</tr>
<tr>
<td>javac</td>
<td>182</td>
<td>614.2</td>
<td>720</td>
<td>3470.0</td>
</tr>
<tr>
<td>creature</td>
<td>65</td>
<td>259.7</td>
<td>626</td>
<td>5881.0</td>
</tr>
<tr>
<td>mindterm1.1</td>
<td>120</td>
<td>461.1</td>
<td>686</td>
<td>4120.0</td>
</tr>
<tr>
<td>scott-1.1</td>
<td>677</td>
<td>1000.4</td>
<td>1214</td>
<td>5660.0</td>
</tr>
<tr>
<td>muffin-0.2</td>
<td>245</td>
<td>665.2</td>
<td>824</td>
<td>5253.0</td>
</tr>
<tr>
<td>javac-1.0</td>
<td>63</td>
<td>592.6</td>
<td>615</td>
<td>4198.0</td>
</tr>
</tbody>
</table>

### Side Effect Analysis

**Rountev et al., OOPSLA’00**

[Graph showing percentage of all indirect accesses across different programs]
Synchronization Removal

- Java library methods are often synchronized for use in multi-threaded applications
- If program is single-threaded or threads do not share data, then this is unnecessary
  - Use *escape analysis* to find objects which escape the thread that creates them
  - If none found, then no need for synchronization
  - Can use results of points-to analysis to estimate objects that escape a method or thread
Thread-local & method-local new sites

(a) Object allocation sites

<table>
<thead>
<tr>
<th>Program</th>
<th>Objects</th>
<th>Thread-local</th>
<th>Method-local</th>
</tr>
</thead>
<tbody>
<tr>
<td>compress</td>
<td>456</td>
<td>99.3%</td>
<td>39.0%</td>
</tr>
<tr>
<td>db</td>
<td>154325</td>
<td>0.3%</td>
<td>0.01%</td>
</tr>
<tr>
<td>mtrt</td>
<td>6457298</td>
<td>99.9%</td>
<td>85.0%</td>
</tr>
<tr>
<td>jlex</td>
<td>7350</td>
<td>50.9%</td>
<td>31.6%</td>
</tr>
<tr>
<td>jack</td>
<td>1340919</td>
<td>86.7%</td>
<td>77.0%</td>
</tr>
<tr>
<td>jess</td>
<td>7902221</td>
<td>17.9%</td>
<td>17.9%</td>
</tr>
<tr>
<td>mpegaudio</td>
<td>2025</td>
<td>12.4%</td>
<td>12.4%</td>
</tr>
<tr>
<td>sablecc</td>
<td>420494</td>
<td>24.9%</td>
<td>13.7%</td>
</tr>
<tr>
<td>javac</td>
<td>3738777</td>
<td>27.6%</td>
<td>21.2%</td>
</tr>
<tr>
<td>javacc</td>
<td>43265</td>
<td>65.7%</td>
<td>45.8%</td>
</tr>
</tbody>
</table>

Run-time Payoff

(b) Run-time objects
Imprecision of Context
Insensitivity

class Y extends X{ ... }

class A{
    X f;
    void m(X q)
    { this.f=q; }
}

A a = new A();//\textcolor{green}{o_1}
a.m(new X());//\textcolor{green}{o_2}
A aa = new A();//\textcolor{green}{o_3}
aa.m(new Y());//\textcolor{green}{o_4}