Dimensions of Precision in Reference Analysis of Object-Oriented Programming Languages

Dr. Barbara G. Ryder

Presented by: Ming-Ho Yee
September 23, 2016

- This was the keynote at CC in 2003 (http://people.cs.vt.edu/ryder/CC03InvitedNew.pdf)
  - It surveys the research, how it has evolved, and classifies some common aspects (dimensions)
- Dr. Ryder is a Professor Emerita at Virginia Tech– she retired earlier this month

Introduction

- Object-oriented (OO) languages have become mature and popular
- Call graphs are useful for OO program analysis
  - But call graph construction is related to reference analysis
- Dimensions of a reference analysis affect precision and cost

2

- The paper was written in 2003, and OO languages were (and still are) popular, and there was a lot of active research
  - Java was first released in 1995
- We've already seen that call graphs are useful for analyzing OO programs, e.g. to inline methods, see if endOfWorld() is called
  - We've also seen that pointer analysis and call graphs are related
  - To construct a call graph, you need to know what receiver objects a variable may point to
  - And to determine what variables point to, you need to know the call graph
- This paper examines reference analysis
  - There is some set of objects, and a reference variable or field may point to any of those objects
  - The analysis goal is to determine information about that set
- There different dimensions and aspects of an analysis
  - We've also seen that there is a trade-off between precision and cost

Reference Analysis

"Determine information about the set of objects to which a reference variable or field may point during program execution."

- Applications
  - Tools: compiler optimizations, test harnesses, refactoring
  - Analyses: side-effect, escape, def-use
- Choosing the right cost/precision trade-off is very important

3

- Reference analysis is a general class of analyses
  - Reference variables may point to some set of objects
  - Want to determine information about that set
- Reference analysis is closely related to call graph construction
- There are many important applications (tools and other analyses) that require reference analysis
  - Compiler optimizations, test harnesses, code refactoring
  - Side-effect analysis, escape analysis, def-use analysis
- Different analyses have different requirements in terms of cost and precision
  - The task is to determine which tradeoffs are acceptable

Dimensions of Precision Analysis

- Flow sensitivity
- Context sensitivity
- Program representation
- Object representation
- Field sensitivity
- Reference representation
- Directionality

- The paper discusses 7 dimensions of varying precision analysis
  - Each one will be discussed in more detail
- We’ve already seen some examples, especially in the first paper of this course
  - Call graph construction algorithms (CHA, RTA, XTA)
  - In general, more precise algorithms also have higher cost

Flow Sensitivity

An analysis is *flow-sensitive* if it accounts for the order of execution of statements in a program. Otherwise, an analysis is *flow-insensitive*.

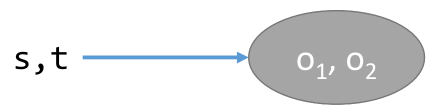
- Flow-sensitive analyses are more precise, but expensive
- Methods in OO programs are generally small
  - Flow-sensitivity probably not that useful
  - Context-sensitivity probably more useful

5

- An analysis is flow-sensitive if the order of execution of statements matters
  - Otherwise it is flow-insensitive
- Clearly, this improves precision, but it’s also more expensive
- OO methods are usually so
  - So it seems like flow-sensitivity isn’t that useful
  - Context sensitivity is probably more useful

Example: Flow Insensitive Analysis

```
A s, t;
s = new A(); // o1
t = s;
s = new A(); // o2
```



Example from Dr. Ryder's presentation

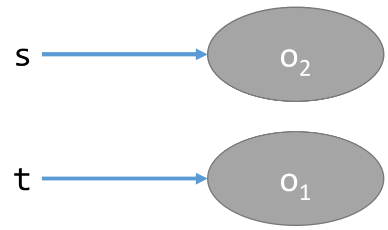
6

- s points to object o1
- t points to whatever s points to (o1)
- s then points to o2
- So t also points to o2

- If the order mattered, the analysis would notice that s no longer points to o1
    - Could jumble the statements around and still get the same result

Example: Flow Sensitive Analysis

```
A s, t;
s = new A(); // o1
t = s;
s = new A(); // o2
```



Example from Dr. Ryder's presentation

- s points to o1
- t points to whatever s points to (o1)
- s = new A() is a kill assignment
    - Previous points-to information is overwritten
    - s no longer points to o1 because it points to o2

- Order matters: s = new A() came last

Context Sensitivity

An analysis is *context-sensitive* if it distinguishes between different calling contexts. Otherwise, an analysis is *context-insensitive*.

- Context-sensitive analyses are more precise, but expensive
- Call string vs functional approach
- An area of active research
  - OO method calls would probably benefit from context-sensitive analyses

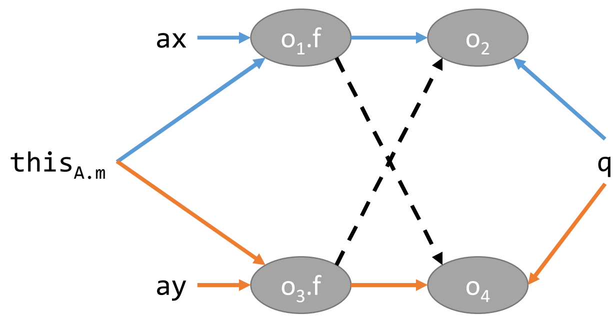
8

- An analysis is context-sensitive if it distinguishes between different calling contexts
  - Otherwise, it is context-insensitive
- Again, you get more precision but with higher cost
- Two common approaches: call string and functional
- There was a lot of research in this area
  - Indicates there is interest in using context-sensitive analyses

Example: Context Sensitivity

```
class Y extends X {…}
class A {
    X f;
    void m(X q)
      { this.f = q; }
}

A ax = new A(); // o1
ax.m(new X());  // o2
A ay = new A(); // o3
ay.m(new Y());  // o4
```



Example from Dr. Ryder's presentation

9

- ax points to o1
- Call ax.m
    - m's this points to o1 and parameter q points to o2
    - Assign this.f = q, so o1 points to o2
- Now a similar sequence
- ay points to o3
- Call ay.m
    - M's this points to o3
    - Assign this.f = q, so o3 points to o4
- But a context-insensitive analysis can't tell that m was called on ax or ay
    - So this.f for both objects could point to o2 or o4

Program Representation (Calling Structure)

- Program calling structure is related to reference analysis solution
- Two approaches:
  - Approximate call graph and then do the reference analysis
  - Interleave reference analysis with call graph construction
- Lazy approach is preferred
  - Only reachable methods are in the call graph
  - Excluding unused methods improves cost and precision

10

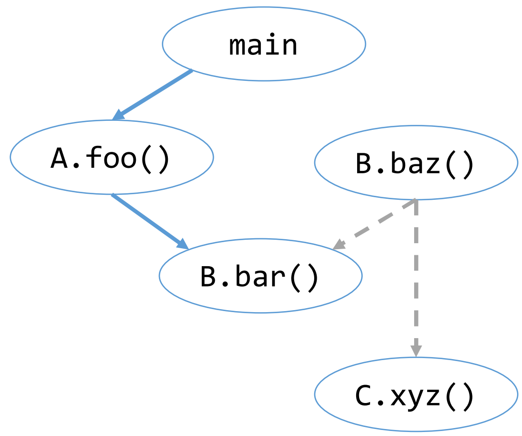
- As we’ve seen many times, building a call graph is related to a reference analysis
- We could build the graph first, and then run the reference analysis
- Or we could do both at the same time, constructing the call graph on-the-fly
  - The call graph algorithms we saw in the first paper did this
  - It starts from main, explores reachable methods, adds them to call graph, and continues building
- Lazy approach is preferred (Gove, Chambers TOPLAS ‘01)
  - Unreachable methods are ignored, which improves cost and precision
  - Better for handling library methods

Example: Program Representation

```
class A {
    public void foo(B b)
      { b.bar(); }
}

class B {
    public void bar() {}
    public void baz()
      { bar(); C.xyz(); }

public static void main(…) {
    A a = new A();
    a.foo(new B());
}
```



11

- If we build the call graph lazily, we see that main calls A.foo()
    - And A.foo() calls B.bar()
- If we just tried building the call graph by looking at all the methods
    - We would also see that B.baz() calls B.bar()
    - And B.baz() calls C.xyz()
    - C.xyz() could call a lot of other methods
- But all of this is irrelevant, since neither are reachable from main

Object Representation

- Two common approaches for elements in the analysis solution
  - One abstract object for all instantiations of a class
  - One abstract object for each creation site of a class
- There are also other, more precise approaches

- One abstract object per class might be OK for call graph construction
  - May not be precise enough for other analyses

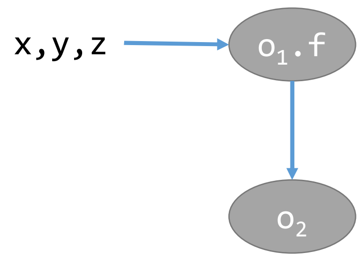
12

- There are two common approaches for representing objects in the analysis solution
  - One abstract object per class, i.e. one abstract object for all instantiations
  - One abstract object for each creation site
- Already, we can see there will be a difference in precision and cost
- There are also more precise approaches
- One abstract object per class might be fine for call graph construction (and is what was used in the first paper)
  - But more sophisticated analyses may require more precision

Example: One Abstract Object Per Class

```
class A {
    public B f;
    public void foo() {}
}
class B {}

A x = new A(); // o1
A y = new A(); // o1
A z = x;       // o1
y.f = new B()  // o2
```



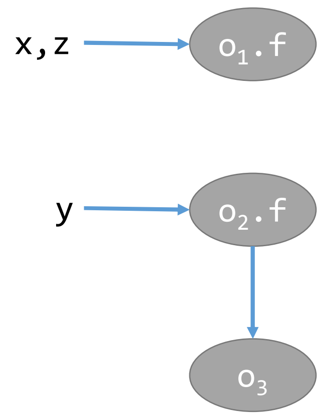
13

- X points to A
- Y points to A
- Z = x so z points to A
- If we were constructing a call graph, this is OK
    - It doesn't matter that there's only one A in the program
- But if we want to know which objects point to B, this is imprecise
    - x, y, z may all refer to some object A that refers to B

Example: One Abstract Object Per Creation Site

```
class A {
    public B f;
    public void foo() {}
}
class B {}

A x = new A(); // o1
A y = new A(); // o2
A z = x;       // o1
y.f = new B()  // o3
```



- X points to object Ax
- Y points to object Ay
- Assign z = x so z points to Ax
- Update y’s field to point to B
    - So Ay points to B, but not Ax

Field Sensitivity

An analysis is *field-sensitive* if its fields are distinctly represented in the solution. Otherwise, an analysis is *field insensitive*.

- Not distinguishing fields may decrease precision and *increase cost*
- But the interaction with other dimensions of precision is not clear
  - More evaluation is needed

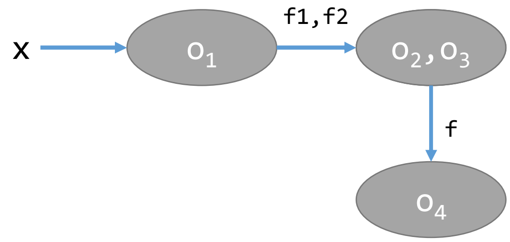
15

- An analysis is field-sensitive if it represents fields in an object distinctly. Otherwise it is field-insensitive.
- A study (co-authored by Dr. Ryder) found that a field-insensitive analysis decreases precision (expected) but also increases cost
  - So using field-sensitive analyses seems to be better
- But the interaction with other dimensions of precision is not clear
  - More evaluation is needed

Example: Field Insensitive Analysis

```
class A {
    public B f1;
    public C f2;
}
class B {}
class C { public D f; }
class D {}

A x = new A();     // o1
x.f1 = new B();    // o2
x.f2 = new C();    // o3
x.f2.f = new D();  // o4
```



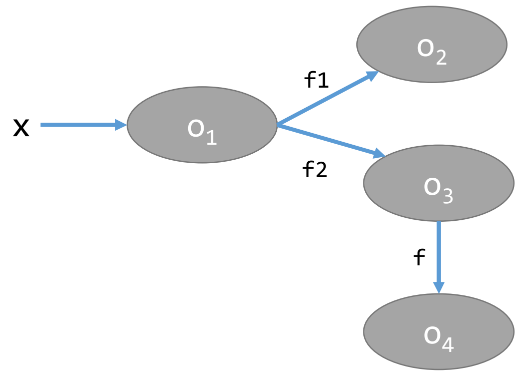
16

- X points to A
- Field x.f1 points to a new object B
- Field x.f2 points to a new object C
- But since we are field insensitive, all we know is that A points to B and C
- Now x.f1.f points to new object D
    - i.e. object B should point to object D
    - But we're field insensitive, so all we know is that B and C may point to D

Example: Field Sensitive Analysis

```
class A {
    public B f1;
    public C f2;
}
class B {}
class C { public D f; }
class D {}

A x = new A();    // o1
x.f1 = new B();   // o2
x.f2 = new C();   // o3
x.f2.f = new D(); // o4
```



17

- X points to A
- Field x.f1 points to a new object B
- Field x.f2 points to a new object C
- We’re field sensitive, so A points to two separate objects B and C
- Now x.f1.f points to new object D
    - i.e. object B should point to object D

Reference Representation

- Generally, each reference has a unique representative
- Alternative approaches:
  - One abstract reference per type
  - One abstract reference per method
- Fewer references are less precise, but the analysis is more efficient

- Examples: CTA, FTA/MTA, XTA

18

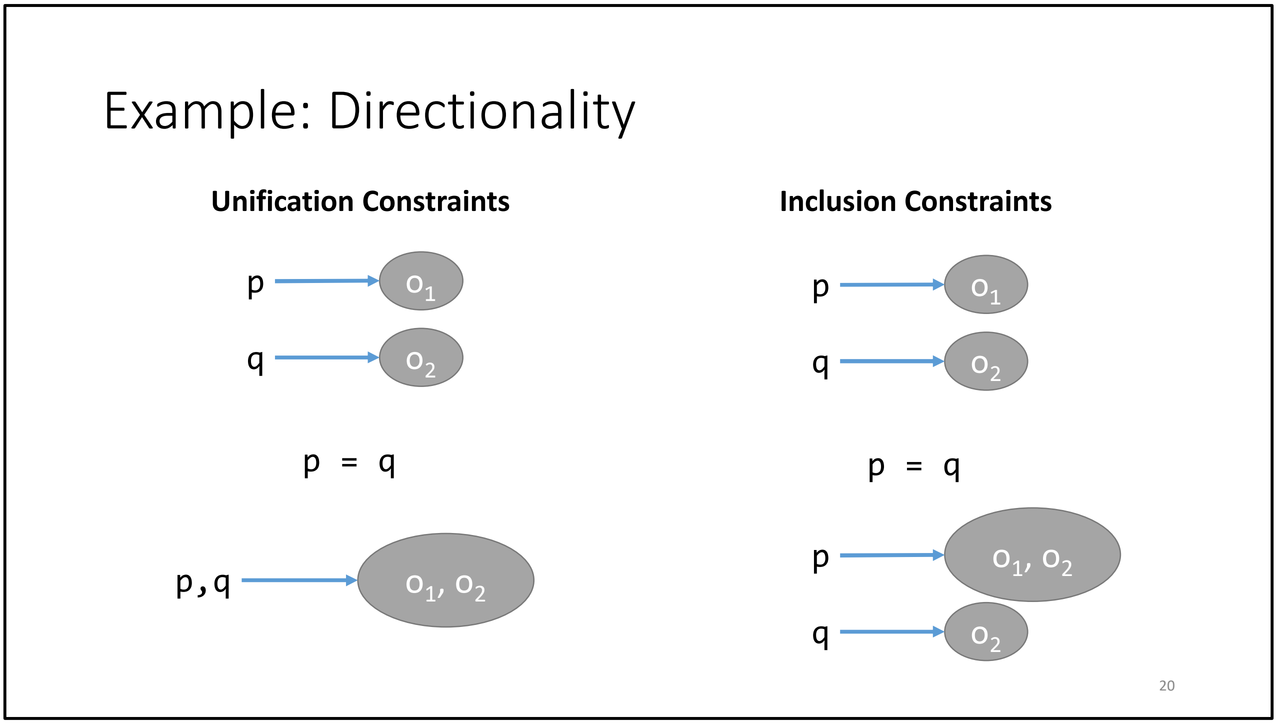
- In general, each reference (i.e. variable or field) has a unique representative
- Other approaches are less precise, but improve performance
  - One abstract reference per type
  - One abstract reference per method
- For examples, see the call graph algorithms from the first class (Tip and Palsberg, OOPSLA '00)
  - Note that the other algorithms with fewer references were often too imprecise

Directionality

- How does the analysis interpret assignments (p = q)?
- Symmetric (unification constraints)
  - p and q have the same information after the assignment
  - E.g. Steensgaard's pointer analysis (worst case almost linear time)
- Directional (inclusion constraints)
  - Information flows from q to p
  - E.g. Andersen's pointer analysis (worst case cubic time)
- Inclusion more precise than unification, but slightly more cost

19

- The final dimension is directionality
  - Concerns pointer assignment statements
- The assignment can be *symmetric* or *directional*
- Symmetric (expressed as unification constraints)
  - Pointers p and q have the same information after the assignment
  - Similar to Steensgaard's analysis
- Directional (expressed as inclusion constraints)
  - Information flows from q to p
  - Similar to Andersen's pointer analysis
- Inclusion constraints are generally more precise than unification
  - But worst case cubic time vs worst case almost linear time
- But in practice, inclusion constraints are practical and worth the extra cost



- Consider an analysis that has partially run
  - P points to o1, q points to o2
- Analysis sees the assignment p – q
- Unification constraint will take the union of the two points-to sets
  - So p and q may both point to o1 and o2
- But the inclusion constraint is different
  - Information from q flows to p, but q remains constant
  - q’s points-to set is still o2
  - But now p may point to o1 or o2
  - p’s set “includes” q’s set

Open Issues

- Reflection
- Native methods
- Exceptions
- Dynamic class loading
- Incomplete programs
- Benchmarks

21

- There are still open issues that analyses must address
- We’ve already seen some of these come up before
  - Reflection: create objects, call methods, access fields at runtime without knowing types at compile time
  - Native methods: Java calls C code, analyzer needs to account for whatever C could do
  - Exceptions: Affects the control flow of a program
  - Dynamic class loading: Basically eval is evil
  - Incomplete programs: If you don’t have access to the library to analyze
  - Benchmarks: Need good benchmarks for evaluating different analyses, to validate them and better understand trade-offs
- Note that Averroes is one approach for handling incomplete programs

Conclusions

- Different dimensions affect the precision and cost of an analysis
  - Challenge: picking the right analysis for a specific application, and making the appropriate precision/cost trade-off
- Observations:
  - OO programs usually have many small methods, and method calls are primary control flow structure
  - Context sensitivity probably more useful than flow sensitivity
  - Inclusion constraints more precise than unification, and still practical
- No single analysis works for all applications

22

- We've seen how different dimensions can affect the precision and cost of an analysis
- The challenge is to pick the right analysis and trade-offs for a specific application
  - What is the required precision? How much performance cost can you handle?
- There are some general observations
  - OO code seems to have many small methods
  - Control flow is basically done by methods calling each other
  - So context sensitivity seems more useful than flow sensitivity
  - Inclusion constraints are more precise than unification constraints, and worth the extra cost
- The important point is that no single analysis works for all applications

Discussion

- Some of the approaches seem to be "obviously" better than others. Are there cases where this might not be the case?
- Not all of the dimensions are binary. Could we use some hybrid approach?
  - E.g. part of an analysis is flow sensitive, the rest is flow insensitive
- What are other dimensions of precision in a reference analysis?

23

- This is a paper from 13 years ago, so programs and hardware have changed
  - Will some approaches which were too expensive in 2003 be feasible in 2016?
  - Paper didn't seem to find flow sensitivity to be very practical, but how about now?
- Some of the dimensions presented are pretty binary, but others aren't
  - Does it make sense to take some hybrid approaches?
  - E.g. part of an analysis is flow sensitive, another part is flow insensitive
  - What would this look like?
- Are there any other dimensions of precision we could consider?