Reference Analyses

- Review of type-based reference analyses for call graph construction
    - See CS515 posted lecture notes
    - Related points-to analyses for C pointers
- Capturing flow in reference analysis
    - Variable Type Analysis for Java
    - Field-sensitive Anderson points-to analysis for Java

Reference Analysis

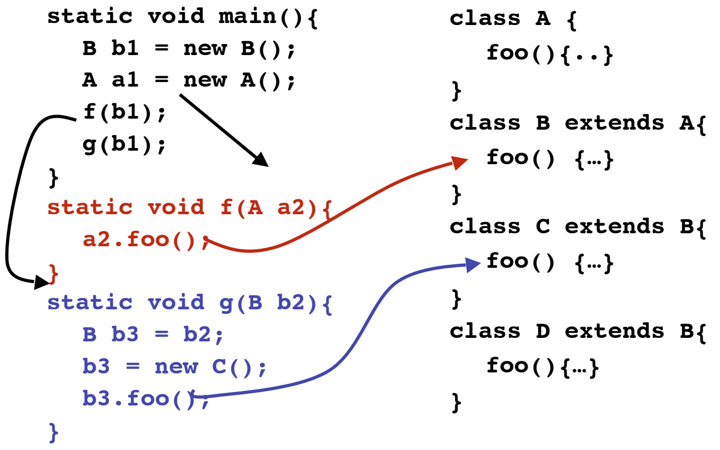
- OOPLs need type information about objects to which reference variables can point to resolve dynamic dispatch
- Often data accesses are indirect to object fields through a reference, so that the set of objects that might be accessed depends on which object that reference can refer at execution time
- Need to pose this as a compile-time program analysis with representations for reference variables/fields, objects and classes.

Reference Analysis

- Many reference analyses developed over past 10+ years address problem using different algorithm and program representation choices that affect precision and cost
  - Class analyses use an abstract object (with or without fields) to represent all objects of a class
  - Points-to analyses use object instantiations, grouped by some mechanism (e.g., object creation sites)

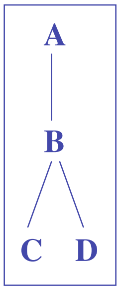
Example

cf Frank Tip, 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(){…}
}
```



Class Hierarchy Analysis

- **First method for reference analysis was CHA by Craig Chamber’s group (UWashington)**
  - **Idea: look at class hierarchy to determine what classes of object can be pointed to by a reference declared to be of class A,**
    - **in Java this is the subtree in the inheritance hierarchy rooted at A,** ***cone (A)***

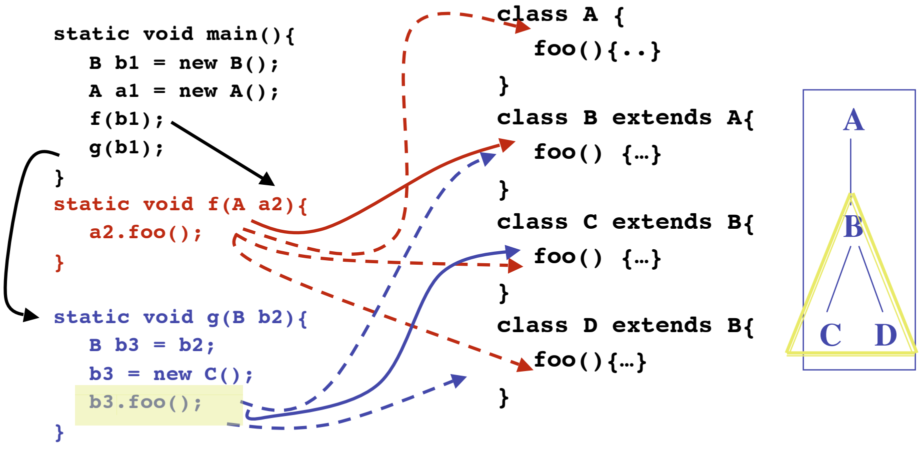
    and find out what methods may be called at a virtual call site
  - **Makes assumption that whole program is available**
  - **Ignores flow of control**
  - **Uses 1 abstract object per class and 1 abstract reference per class**

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

---

cf Frank Tip, OOPSLA’00

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();
}
```

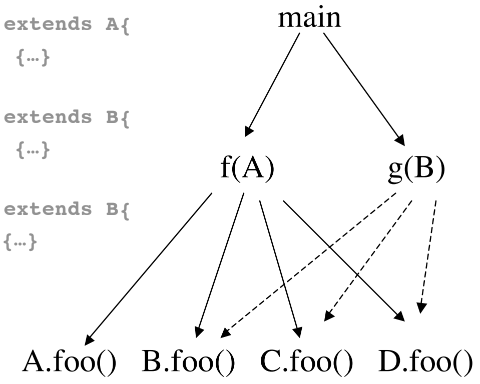
```
class A {
   foo(){..}
}
class B extends A{
   foo() {…}
}
class C extends B{
   foo() {…}
}
class D extends B{
   foo(){…}
}
```

Cone(Declared_type(receiver))

CHA Example - Call Graph

```
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(){…}
}
```

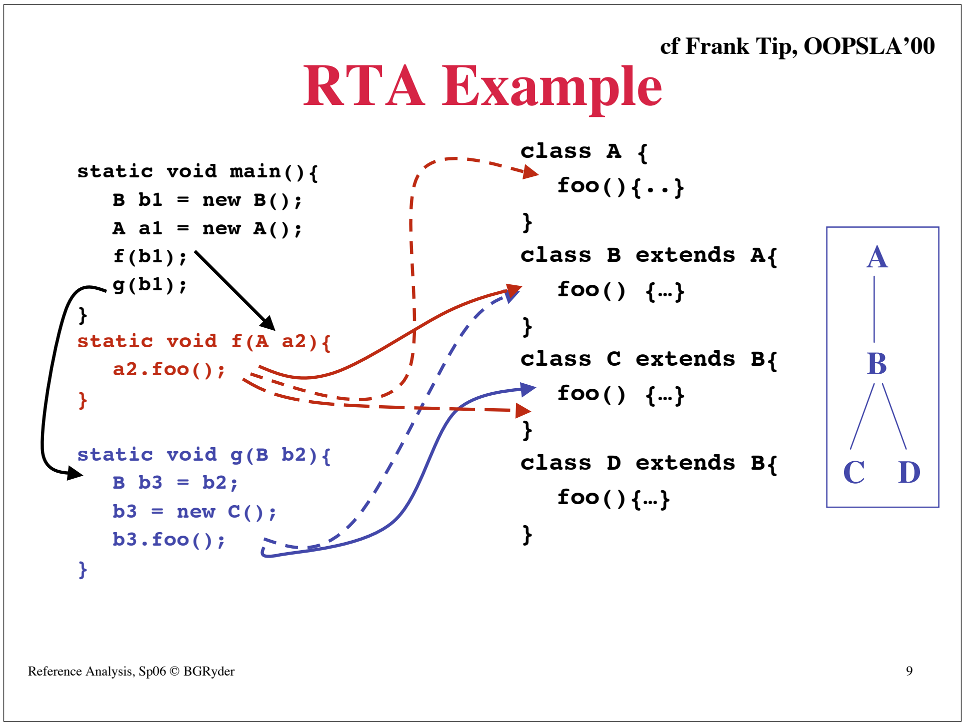


Call Graph

Rapid Type Analysis

- **Improves on CHA**
- **Constructs call graph on-the-fly, interleaved with the analysis**
- **Only expands calls if has seen an instantiated object of an appropriate type**
  - **Ignores classes which have not been instantiated as possible receiver types**
- **Makes assumption that whole program is available**
- **Uses 1 abstract object per class and 1 abstract reference per class**

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



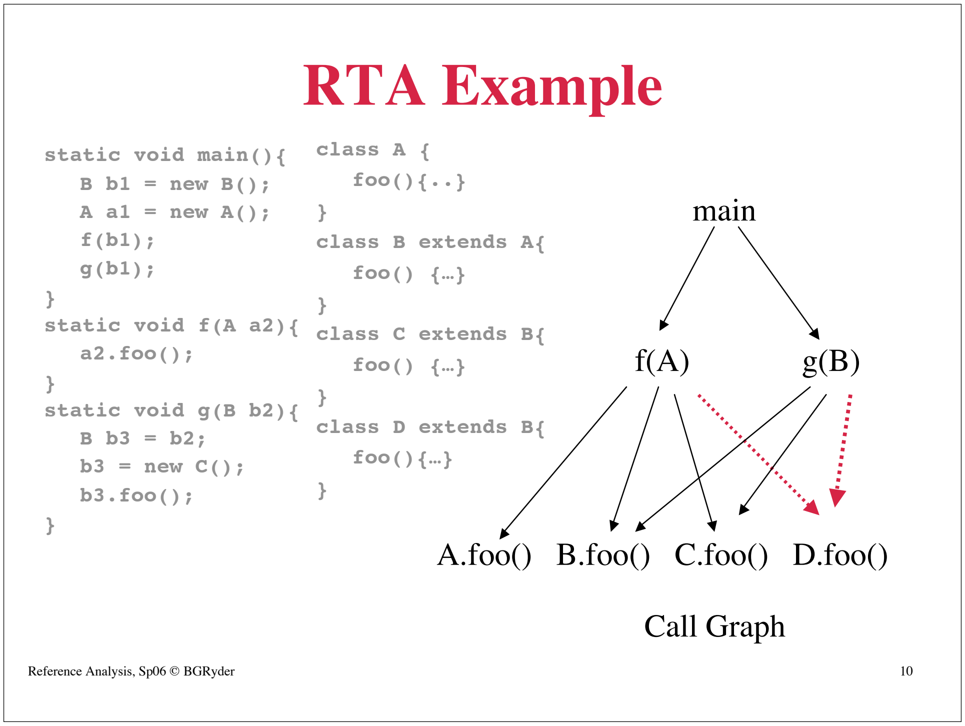
cf Frank Tip, OOPSLA'00

RTA 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(){…}
}
```



RTA 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(){…}
}
```

Call Graph

Reference Analysis

- The analysis can incorporate information about flow of control in the program or ignore it
  - *Flow sensitivity* (accounts for statement order)
  - *Context sensitivity* (separates calling contexts)
- Program representation used for analysis can incorporate reachability of methods as part of the analysis or assume all methods are reachable
- Techniques can be differentiated by their solution formulation (that is, kinds of relations) and *directionality* used
  - e.g., for assignments

p = q, interpreted as
Pts-to(q) ⊆ Pts-to(p) <Andersen> vs. Pts-to(q) = Pts-to(p) <Steensgaard>

Type-based vs Flow-based

- Uses only class hierarchy
- Same points-to set for every reference of a type
- Always 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-sensitive or flow-insensitive
- Can be context-sensitive or context-insensitive
- May be expensive
- Related to points-to approaches for C
- Okay for side-effect and dependence calculations

Sensitivity

- ***Flow sensitivity***
    - **If the problem requires that you consider the sequential order of statements in the program, then problem is FS (e.g., reaching defs)**
    - **If the order of processing statements can be arbitrary, then the problem is FI (e.g., may be referenced)**
- ***Context sensitivity***
    - **If the problem requires that you analyze each method independently for each of its instantiations, then the problem is context-sensitive (e.g., a add method for a Collection)**
    - **Otherwise, if you summarize the effects of executing a method over all its instantiations, then the problem is context-insensitive**

Points-to Analyses for C

- **Popular flow- and context-insensitive formulations of points-to analysis**
    - **Scalable**
    - **Good enough for ensuring safety of some optimizations**
    - **Good for program understanding applications**
    - **Not great for applications needing precise def-use information (e.g., program slicing, testing)**
    - **General approach is *unification* or *inclusion* constraints**
    - **Newer versions kept track of individual *struct* fields as pointer targets**
- **Extended to points-to analyses for OOPL reference variables**

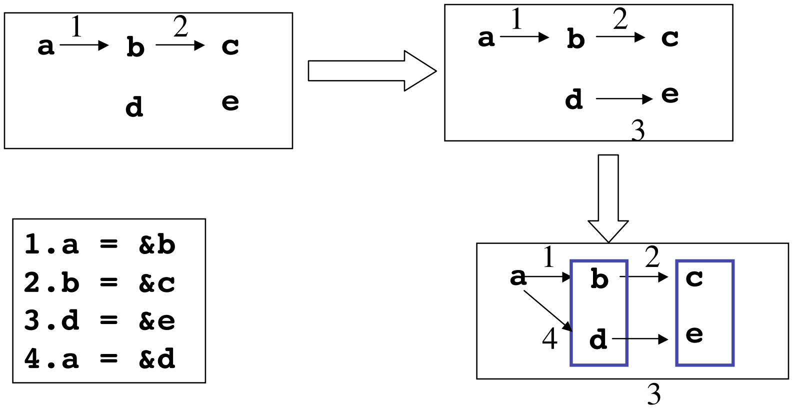
Points-to Analyses for C

- Steensgaard's algorithm (POPL'96)
  - Uses unification constraints so that for pointer assignments, p = q, algorithm makes Pts-to(p)=Pts-to(q)
    - This union operation is done recursively for multiple-level pointers
  - Reduces the size of the points-to graph (in terms of both nodes and edges)
    - *Almost linear* solution time in terms of program size, O(n)
    - Uses fast union-find algorithm
    - Imprecision stems from merging points-to sets
  - One points-to set per pointer variable over entire program

---

cf M Shapiro and S. Horwitz, "Fast and Accurate Flow-insensitive Points-to Analysis" POPL'97

Steensgaard - Example



```
1.a = &b
2.b = &c
3.d = &e
4.a = &d
```

Points-to sets found a ⟶ b,d ⟶ c,e

Steensgaard Solution Procedure - At a glance

- Find all pointer assignments in program
- Form set of points-to graph nodes from pointer variables/fields and variables (in the heap or whose address has been taken)
  - Examine each statement, in arbitrary order, and construct points-to edges
    - Merge nodes (and edges) where indicated by unification constraints
- After linear pass over these assignments, points-to graph is complete

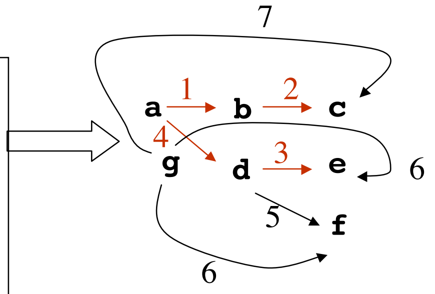
Points-to Analysis for C

- Andersen's analysis
  - Uses inclusion constraints so that for pointer assignments, p = q, algorithm makes

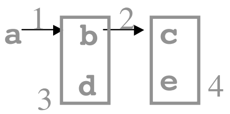
    $\text{Pts-to}(q) \subseteq \text{Pts-to}(p)$
  - Points-to graph is larger than Steensgaard's and more precise
  - Worst case cubic complexity in program size, $O(n^3)$, to construct the points-to graph
  - One points-to set per pointer variable over entire program

Andersen - Example

```
1.a = &b
2.b = &c
3.d = &e
4.a = &d
5.d = &f
6.g = d
7.g = *a
```



Steensgaard solution



Field-sensitive Points-to Analysis

- **Andersen-style points-to analysis**
  - **On-the-fly call graph construction**
  - **Flow- and context-insensitive**
  - **Keep track of reference fields of abstract objects**
  - **Creation site abstraction of objects**
  - **Calculate points-to sets of each reference variable and field**
- **Algorithms:**
    - **Rountev, Milanova, Ryder, OOPSLA’01, (annotated inclusion constraints)**
    - **Lhotak, Hendren, CC’03 (Soot’s Spark package)**
    - **Sundaresan et. Al, OOPSLA’00 (VTA - an approximation)**

Andersen's Solution Procedure - At a glance

- Find all pointer assignments in program
- Form set of points-to graph nodes from pointer variables/fields and variables on the heap or whose address is taken
  - Examine each statement, in arbitrary order, and construct points-to edges
    - Need to create more edges when see p = q assignments so that all outgoing points-to edges from q are copied to be outgoing from p (i.e. processing inclusion constraints)
    - If new outgoing edges are added to q during the algorithm, they must be also copied to p

---

cf. V.Sundaresan, et. al, "Practical Virtual Method Call Resolution for Java", OOPSLA'00

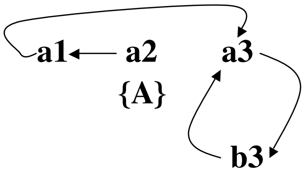
VTA - Variable Type Analysis

- Analysis included in the McGill SOOT system
- How works? follows type propagation from a new site through plausible chains of assignments to reference variables
- Builds a type propagation graph, using a CHA call graph
  - Nodes are reference vars/fields/parameters
  - Edges represent reference to reference assignments
  - Initializes types for some reference nodes with type from an associated object creation site
  - Propagates types along directed edges
- Effectively, this is a points-to analyses using inclusion relations and abstract objects with fields, that traces flow through reference assignments

VTA

- Goal: to analyze program in only 1 iteration over the graph
- Uses a separate representative program-wide for each named reference
- Propagates one abstract object per class to represent all created objects of that class
- Is a flow-insensitive, context-insensitive analysis

```
//B extends A
A a1, a2, a3; B b3;
a2 = new A();
a1 = a2;
a3 = a1;
a3 = b3;
b3 = (B)a3;
```

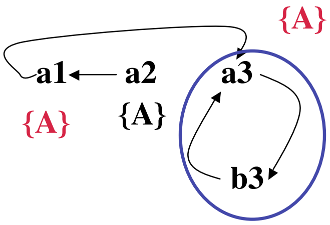


Initial graph and types

cf. Sundaresan et. al

VTA Example

```
//B extends A
A a1, a2, a3; B b3;
a2 = new A();
a1 = a2;
a3 = a1;
a3 = b3;
b3 = (B)a3;
```



scc collapsed

Empirical results report removal of 10-65% methods, 17-65% edges of CHA call graph; helps especially in library code; Considerably better than RTA over CHA in resolving calls (over entire application)

Q: For what client analyses is VTA appropriate?

Field-sensitive Points-to Analysis (FieldSens)

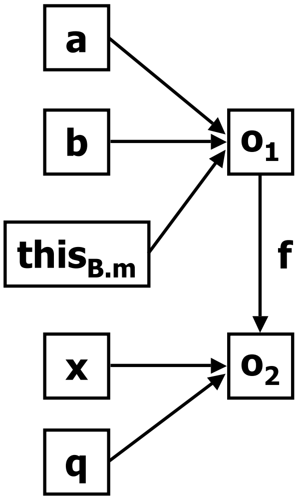
- Flow-insensitive, context-insensitive extension of Andersen’s analysis for C
  - Have to handle dynamic dispatch, fields, and libraries
- Can use a precomputed callgraph or can compute an on-the-fly callgraph from the points-to relations being calculated
- Distinguishes object fields
- Originally formulated as a constraint solution problem -- admits a dataflow formulation too

Rountev, A. Milnova, B. Ryder, “Points-to Analysis for Java Using Annotated Constraints”, OOPSLA’00;
Lahotak and Hendren, “Scaling Java Points-to Analysis using SPARK”, CC’03

Points-to Analysis in Action

```
class A { void m(X p) {..} }
class B extends A {
    X f;
    void m(X q) { this.f=q; }
}
B b = new B();
X x = new X();
A a = b;
a.m(x);
```

A.m() not analyzed because it’s unreachable.



Rules of Algorithm

- 4 types of reference assignment statements
  - Each has points-to effects
    - Allocation: p=new X()
      - Adds o_X to Pts(p)
    - Copy: p = q
      - $Pts(p) \supseteq Pts(q)$ (i.e., If $o \in Pts(q)$, then $o \in Pts(p)$)
    - Field store: p.f = q
      - If $o \in Pts(p)$ and $r \in Pts(q)$, then $r \in Pts(o.f)$
    - Field load: p = q.g
      - If $o \in Pts(q)$ and $oo \in Pts(o.g)$ then $oo \in Pts(p)$
- Algorithm described as construction of a points-to graph
  - Nodes: reference variables and fields
  - Edges: ref ---> object or field labelled: obj--->obj

FieldSens Algorithm

Initially

- Make a list of all reference assignments excluding allocations
- Process allocations first and create the corresponding points-to relations
- Create a worklist intialized with all the reference variables and fields
  - As the algorithm proceeds, any reference variable (or object field) whose points-to set has changed will be put on the worklist

FieldSens Algorithm, cont

Repeat until the worklist is empty

- Remove a variable from the worklist and iterate through the statements (involving this variable)
  - Calculate points-to relations implied by the assignment statement; if this changes the points-to set of a variable or object field, add it to the worklist (Note: if p points to o and o.f's points-to set is changed, <p.f= >, then we record o.f as a variable whose points-to set has changed.)

Differences with Algm for C

- Fields cannot be ignored
  - Field-sensitive versus field-based
- No explicit address operator & in Java
- Reachability of code
  - On-the-fly call graph construction versus use of a static approx call graph
  - Possibly multiple entries to call graph (e.g., static initializers, thread start methods, finalizers, etc)
- Need to know entire set of classes that will ever be loaded during execution
  - Because of reflection
  - Large Java libraries vs. smaller C libraries
- Can use strong typing to filter out spurious points-to relations

Field-sensitive Points-to Analysis w Inclusion Constraints

- **Based on Andersen's points-to analysis**
- **Define and solve a system of annotated set-inclusion constraints - different from dataflow formuation**
  - **Handles virtual calls by simulation of run-time method lookup**
  - **Models the fields of objects**
  - **Extended BANE (UC Berkeley) constraint solver**
- **Analyzes only possibly executed code**
  - **Ignores unreachable code from libraries**

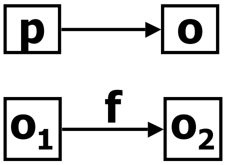
Rountev, A. Milnova, B. Ryder, "Points-to Analysis for Java Using Annotated Constraints" OOPSLA'00

Annotated Constraints

- **Form:** $L \subseteq_a R$
  - L **and** R **denote set expressions**
  - **Annotation** a**: additional information (e.g., object fields)**
- **Kinds of set expressions** L **and** R
  - **Set variables: represent points-to sets**
  - ***ref* terms: represent objects**
  - **Other kinds of expressions**

Set variables and *ref* terms

- Set variables represent points-to sets
  - For each reference variable p: V_P
  - For each object o: V_o
- Object o is denoted by term *ref*(o,V_o)

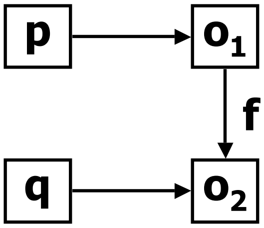


➡ $ref(o,V_o) \subseteq V_P$

➡ $ref(o_2,V_{o_2}) \subseteq_f V_{o_1}$

Example: Accessing Fields

p = new A();	$ref(o_1,V_{o_1}) \subseteq V_p$
q = new B();	$ref(o_2,V_{o_2}) \subseteq V_q$
p.f = q;	$V_p \subseteq proj(ref,W)$ $V_q \subseteq_f W$



Think of W as representing V_{o1} and V_{o3} when p points to o1, o3

Constraint generation

Example: Solving Constraints

Constraint resolution

$ref(o_1,V_{o_1}) \subseteq V_p$

$ref(o_2,V_{o_2}) \subseteq V_q$

$V_p \subseteq proj(ref,W)$

$V_q \subseteq_f W$

$W \subseteq V_{o_1}$

$V_q \subseteq_f V_{o_1}$

$ref(o_2,V_{o_2}) \subseteq_f V_{o_1}$

$o_1.f$ points to o_2

Example: Virtual Calls

Constraint generation from actual call:

p.m(x); ⇨ $V_P \subseteq_{\boxed{m}} \boxed{lam(V_x)}$

Constraint resolution:

receiver object **o** ⇨ $ref(\boxed{o},V_O) \subseteq V_P$

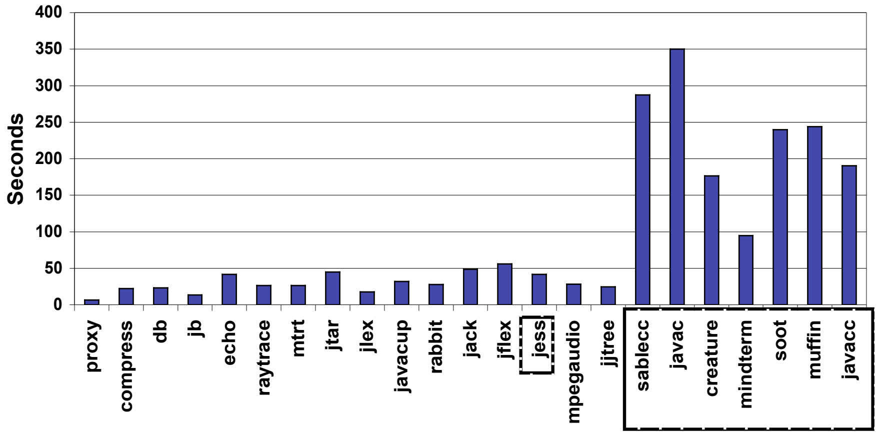
Actual method called, **A.m(z)** ⇨ $V_x \subseteq V_z$

$ref(o,V_O) \subseteq V_{this(A.m)}$

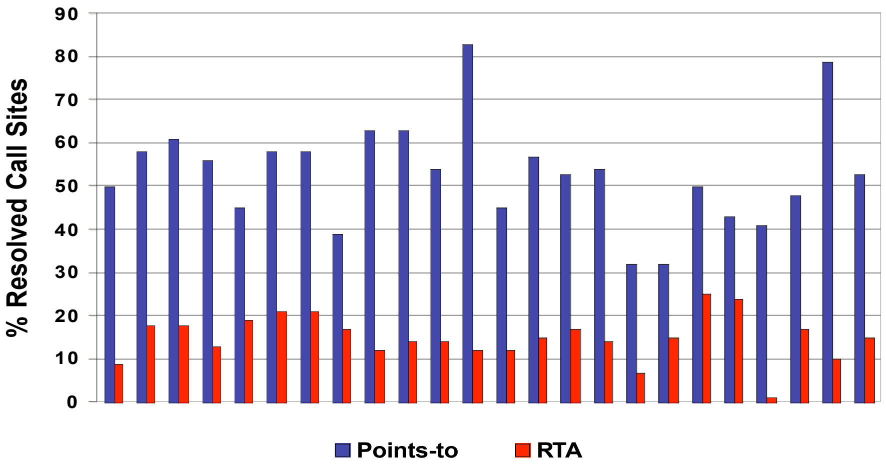
Experiments

- 23 Java programs: 14 – 677 user classes
  - Added the necessary library classes (JDK 1.1)
  - Machine: 360 MHz, 512Mb SUN Ultra-60
- Cost measured in time and memory
- Precision (wrt usage in client analyses and transformations)
  - Object read-write information
  - Call graph construction
  - Synchronization removal and stack allocation

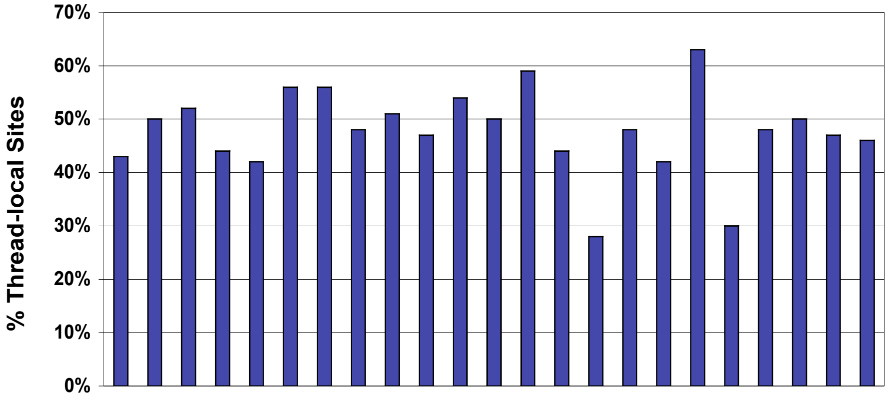
Analysis Time



Resolution of Virtual Call Sites



Thread-local new sites



Points-to Analysis in *Spark*

Lhotak and Hendren, "Scaling Java Points-to Analysis Using Spark", CC'03

- **Framework for points-to analyses in Soot**
  - **Intermediate representation uses def-use chains to achieve SSA-like precision**
  - **Allows selection of static versus on-the-fly cal lgraph construction**
  - **Allows selection of field-sensitive versus field-based analysis**
  - **Uses declared types of references (and fields) to filter points-to propagation**
  - **Collapses cycles of references in points-to assignment graph (as they all have the same points-to set); uses union-find algorithm**
  - **Compared various set implementations for efficiency**
  - **Uses native code simulation framework**

Spark Experiments

- **Best algorithms found**
  - **On-the-fly types, on-the-fly call graph, field-sensitive**
    - **For clients needing the best precision**
  - **On-the-fly types, CHA call graph, field-sensitive**
    - **Faster than #1, but more edges in callgraph**
  - **On-the-fly types, CHA call graph, field-based**
    - **Fastest analysis, but least precise**

Lhotak & Hendren
CC'03

Spark Findings

Deref sites for 0 means p.f seen in CHA call graph could not be reached

Call site of 0 means unreachable; call site of 1 means direct call

Table II. Analysis precision.

		Dereference Sites (% of total)							Call Sites (% of total)			
		0	1	2	3-10	11-100	101-1000	1001+	0	1	2	3+
compress	nt-otf-fs	35.2	23.4	6.3	14.1	5.9	0.1	14.9	53.8	42.6	1.6	1.9
	at-otf-fs	35.3	32.7	8.0	17.4	4.3	2.2	0.0	53.8	42.6	1.6	1.9
	ot-otf-fs	36.9	32.1	7.8	17.0	4.3	1.8	0.0	54.6	42.3	1.3	1.8
	ot-cha-fs	20.5	39.6	10.1	21.8	6.0	2.1	0.0	40.8	51.7	2.6	4.9
	ot-otf-fb	26.3	38.1	9.4	19.2	5.1	1.9	0.0	48.0	47.4	2.0	2.6
	ot-cha-fb	16.0	41.6	10.9	22.9	6.4	2.2	0.0	37.5	54.3	2.9	5.2
javac	nt-otf-fs	31.4	22.2	6.0	12.9	5.8	6.4	15.2	50.1	45.3	1.9	2.7
	at-otf-fs	31.6	33.9	8.7	17.7	5.7	2.4	0.0	50.1	45.3	1.9	2.7
	ot-otf-fs	33.0	33.3	8.6	17.3	5.7	2.0	0.0	50.8	45.2	1.5	2.5
	ot-cha-fs	18.4	40.0	10.5	21.5	7.2	2.3	0.0	38.0	53.9	2.6	5.5
	ot-otf-fb	23.6	38.6	10.0	19.2	6.5	2.1	0.0	44.6	49.9	2.1	3.3
	ot-cha-fb	14.5	41.7	11.3	22.5	7.6	2.4	0.0	34.9	56.3	3.0	5.8
sablecc	nt-otf-fs	31.6	24.2	5.9	12.7	9.5	0.2	15.8	49.9	45.8	2.1	2.2
	at-otf-fs	31.7	37.9	7.4	16.2	4.9	2.0	0.0	49.9	45.8	2.1	2.2
	ot-otf-fs	33.1	37.4	7.3	15.7	4.9	1.6	0.0	50.8	45.5	1.6	2.0
	ot-cha-fs	18.4	44.1	9.2	20.1	6.4	1.9	0.0	37.9	54.2	2.9	5.0
	ot-otf-fb	23.6	42.6	8.7	17.7	5.7	1.7	0.0	44.7	50.3	2.2	2.8
	ot-cha-fb	14.4	45.8	10.0	21.0	6.8	1.9	0.0	34.9	56.6	3.3	5.2
jedit	nt-otf-fs	25.6	29.6	6.6	12.7	3.8	1.5	20.2	43.8	52.0	1.9	2.2
	at-otf-fs	25.7	42.4	9.0	16.3	4.7	2.0	0.0	43.8	52.0	1.9	2.2
	ot-otf-fs	27.1	42.0	8.9	15.9	4.3	1.9	0.0	44.6	51.9	1.4	2.1
	ot-cha-fs	14.5	47.9	10.7	19.4	5.5	2.1	0.0	33.2	59.3	2.3	5.1
	ot-otf-fb	18.9	46.7	10.0	17.6	4.8	2.0	0.0	38.6	56.7	1.9	2.8
	ot-cha-fb	12.1	49.0	11.0	20.1	5.7	2.1	0.0	30.7	61.5	2.5	5.3

Lhotak & Hendren
CC'03

Spark Best Algorithms

Table V. Overall Results (time in seconds, space in MB, precision in precent).

Benchmark	methods (CHA)	stmts (CHA)	types	ot-otf-fs			ot-cha-fs			ot-cha-fb		
				time	space	prec.	time	space	prec.	time	space	prec.
compress	15183	278902	2770	52	106	69.1	13	127	60.1	10	90	57.6
db	15185	278954	2763	52	107	68.9	14	128	59.9	11	90	57.4
jack	15441	288142	2816	54	112	68.7	14	132	60.1	11	94	57.6
javac (1.1.8)	4602	86454	874	8	27	63.6	3	24	57.4	1	16	55.1
javac	16307	301801	2940	89	131	66.3	18	148	58.4	11	104	56.2
jess	15794	288831	2917	57	115	68.1	15	136	59.2	10	97	56.8
mpegaudio	15385	283482	2782	56	112	68.6	16	134	59.7	11	93	57.4
raytrace	15312	281587	2789	53	107	68.5	13	129	59.6	11	91	57.1
sablecc	16977	300504	3070	95	136	70.5	18	158	62.5	14	112	60.3
soot	17498	310935	3435	88	143	68.3	19	162	60.4	18	116	58.4
jedit	19621	367317	3395	100	218	69.1	38	244	62.3	21	143	61.1

#Jimple stmts

Precision means #deref sites with Pts-to set of size 0 or 1