OOPLs - call graph construction

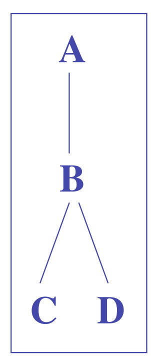
- **Compile-time analysis of reference variables and fields**
  - **Determines to which objects (or types of objects) a reference variable may refer during execution**
  - **Primarily hierarchy-based methods**
    - **Class hierarchy analysis (CHA)**
    - **Rapid type analysis (RTA)**
  - **Incorporating flow of control**
    - **Tip-Palsberg class analyses (XFA)**

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(){…}
}
```



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., creation sites)**
- **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)**

Reference Analysis

- **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) vs. Pts-to(q) = Pts-to(p)

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 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**

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

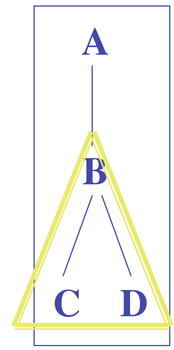
CHA Example

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(){…}
}
```

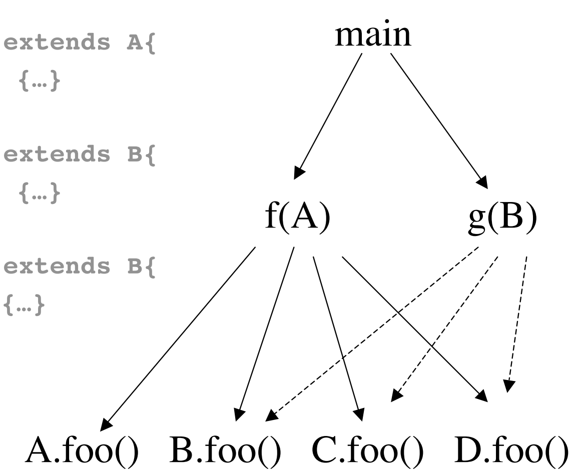


Cone(Declared_type(receiver))

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(){…}
}
```



Call Graph

More on CHA

- Type of receiver needn't be uniquely resolvable to devirtualize a call
  - Need *applies-to* set for each method (the set of classes for which this method is the target when the runtime type of the receiver is one of those classes)
    - At a call site, take set of possible classes for receiver and intersect that with each possible method's applies-to set.
    - If only one method's set has a non-empty intersection, then invoke that method directly
    - Otherwise, need to use dynamic dispatch at runtime
  - Also can use runtime checks of actual receiver type (through reflection) to cascade through a small number of choices for direct calls, given predictions due to static or dynamic analysis

Rapid Type Analysis

- Improves CHA
- Constructs call graph on-the-fly, interleaved with the analysis
- Only expands calls if has seen an instantiated object of 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

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

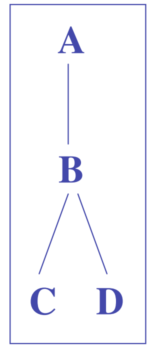
RTA Example

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();
}
```

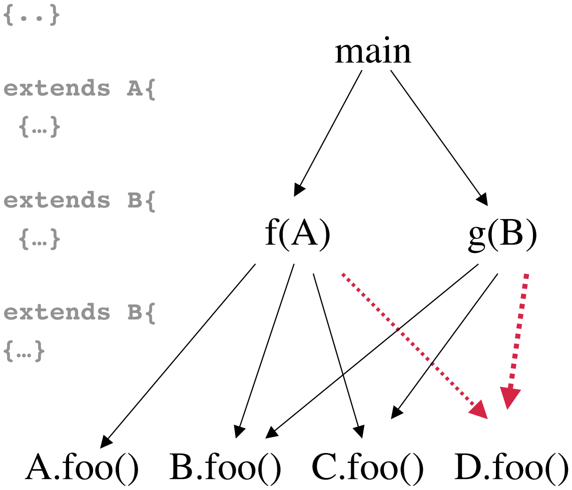
```
class A {
   foo(){..}
}
class B extends A{
   foo() {…}
}
class C extends B{
   foo() {…}
}
class D extends B{
   foo(){…}
}
```



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

Comparisons

Bacon-Sweeney, OOPSLA'96

```
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( );
}
```

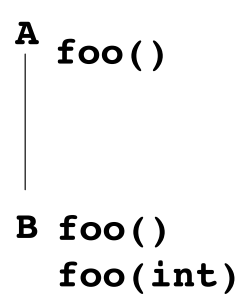
CHA resolves result2 call uniquely to B.foo() because B has no subclasses, however it cannot do the same for the result3 call.
RTA resolves the result3 call uniquely because only B has been instantiated.

Type Safety Limitations

Bacon-Sweeney, OOPSLA'96

- **CHA and RTA both assume type safety of the code they examine**

```
//#1
void* x = (void*) new B
B* q = (B*) x;//a safe downcast
int case1 = q->foo()
//#2
void* x = (void*) new A
B* q = (B*) x;//an unsafe downcast
int case2 = q->foo()//probably no error
//#3
void* x = (void*) new A
B* q = (B*) x;//an unsafe downcast
int case3 = q->foo(666)//runtime error
```



These analyses can't distinguish these 3 cases!

Experimental Comparison

Bacon and Sweeney, OOPSLA'96

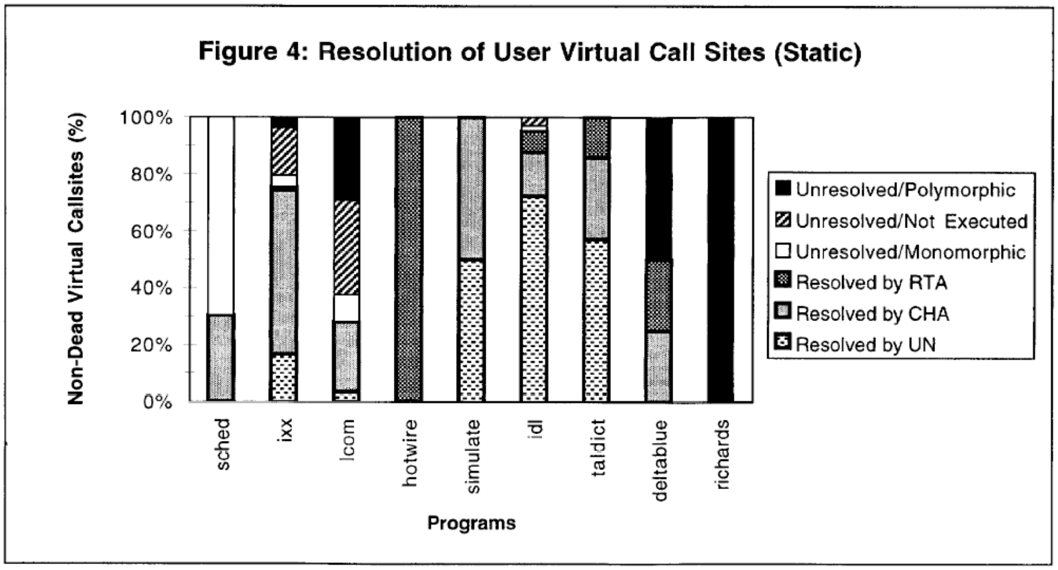
Benchmark	Lines	Description
sched	5,712	RS/6000 Instruction Timing Simulator
ixx	11,157	IDL specification to C++ stub-code translator
lcom	17,278	Compiler for the "L" hardware description language
hotwire	5,335	Scriptable graphical presentation builder
simulate	6,672	Simula-like simulation class library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

Table 1: Benchmark Programs. Size is given in non-blank lines of code

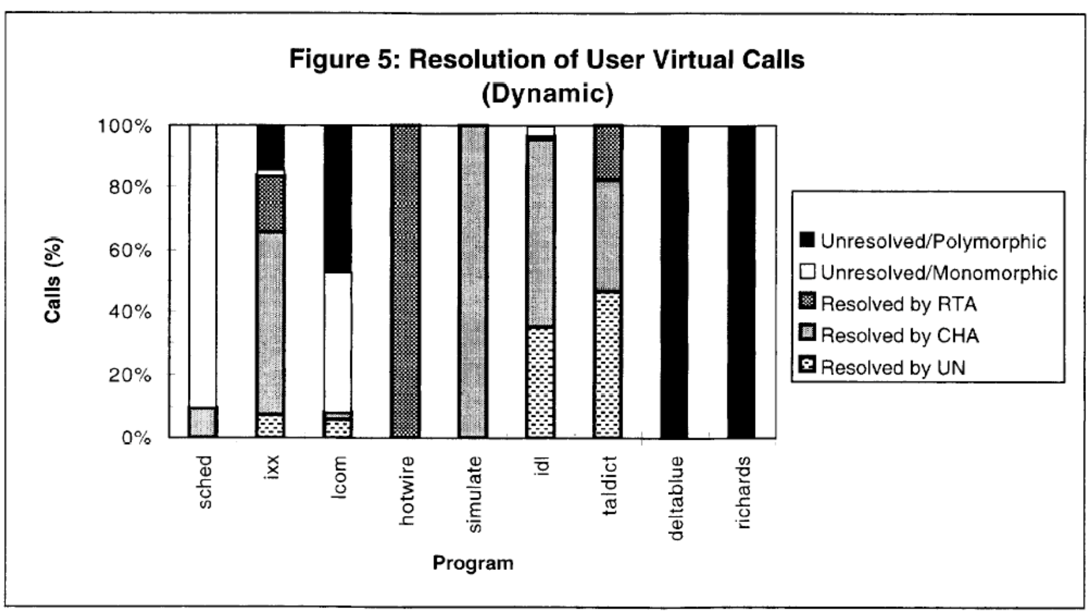
Data Characteristics

- **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**
  - **Importance of benchmarks**

Static Resolution



Dynamic Resolution



Findings

- 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 cfg's of methods and identifying call sites
- Claim that RTA is good enough for call graph construction so that more precise analyses are not necessary for this task

Dimensions of Analysis

- How to achieve more precision in analysis for slightly increased cost?
  - Incorporate flow in and out of methods
  - Refine abstract object representing a class to include its fields
  - Incorporate locality of reference usage in program into analysis rather than 1 'references' solution over the entire program
  - Always use reachability criteria in constructing call graph

Tip and Palsberg Analyses

- **Tip and Palsberg, OOPSLA'00, explored several algorithms that incorporated flow, which are more precise than RTA**
  - **Track classes propagated into and out of method calls through parameter passing**
  - **Objects have one representative object per class, with or without distinct fields**
  - **Reference expressions are grouped by class or by method**

Tip and Palsberg, "Scalable Propagation-based Call Graph Construction Algorithms", OOPSLA'00

XTA Analysis

Tip and Palsberg, "Scalable Propagation-based Call Graph Construction Algorithms", OOPSLA'00

- **Start at main() method.**
- **Do a reachability propagation of classes through an on-the-fly constructed call graph**
  - **At any point in the algorithm there is a set of reachable methods R, starting from main()**
- **Associate a set of classes that reach method M, S_M (this is having all references of a class with one abstract representative per method, not one representative for the entire program)**
- **Uses abstract objects with fields to represent all instances of a class**

XTA Analysis

- **Q: How to expand virtual e.m() in reachable method M ?**
  - **Expand virtual call only by appropriate $C \in S_M$ where $C \in$ cone(declaredType(e)) to call M'**
    - **Make M' reachable**
    - **Add cone(paramType(M')) $\cap S_M$ to $S_{M'}$ (adds possible actual param types for M' from M, to set of classes that reach M')**
    - **Add cone(returnType(M')) $\cap S_{M'}$ to S_M**
    - **Add C to $S_{M'}$**
  - **For each object created in M (new A()), if M is reachable, then $A \in S_M$**
  - **For each field read =*.f in M, if M is reachable, then $S_f \subseteq S_M$**
  - **For each field write *.f = in M, if M is reachable, then cone(declaredType(f)) $\cap S_M \subseteq S_f$**

Example of XTA

cf Frank Tip, OOPSLA'00

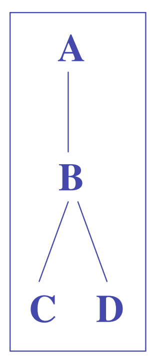
{A,B}

```
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();
}
```

{A,B}

{B,C}

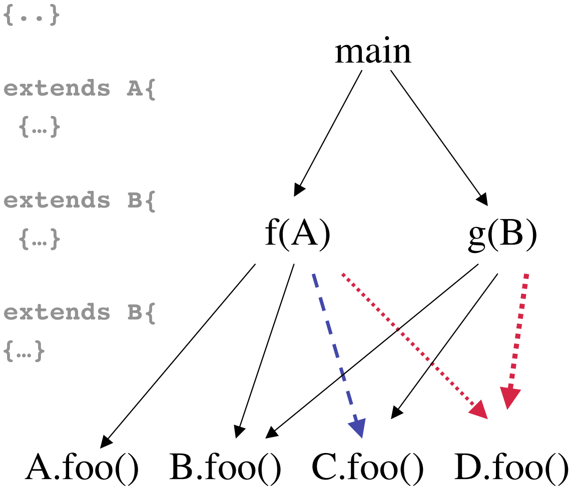
```
class A {
   foo(){..}
}
class B extends A{
   foo() {…}
}
class C extends B{
   foo() {…}
}
class D extends B{
   foo(){…}
}
```



XTA 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(){…}
}
```

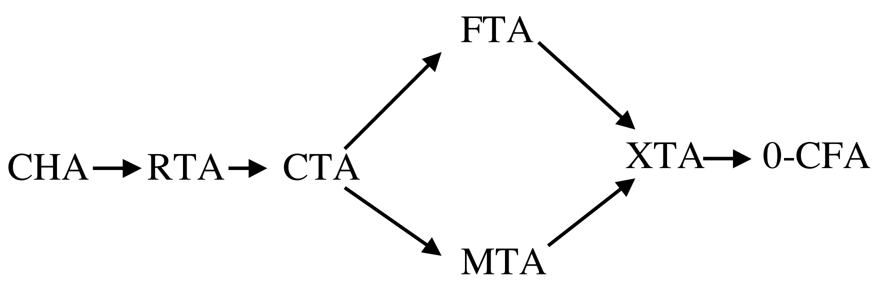


Call Graph

Variants of XTA

- **CTA - uses one abstract object per class, without fields; keeps one program-wide representative for each type of reference**
- **MTA - uses one abstract object per class with fields distinguished but keeps one program-wide representative for each type of reference**
- **FTA - uses one abstract object per class without fields; has one representative per method for each type of reference**

Analysis Precision



arrows show increasing cost and precision

Details

- **Algorithm is iterative and must go until hit a fixed point.**
- **Conditions are expressed as constraints which must be true for the solution**
  - **Additions to reference sets trigger more propagation of new information through the cfg's and calls**
- **Impressive results**

Java Program Dataset

benchmark	# classes	# methods	#fields (reference-typed)	# virtual call sites
Hanoi	44	379	232 (107)	285
Ice Browser	76	761	500 (253)	922
mBird	2,050	17,946	6739 (4284)	3,269
Cindy	468	4,449	3075 (1677)	5,085
CindyApplet	468	4,449	3075 (1677)	2,502
eSuite Sheet	588	5,590	4305 (1412)	4,459
eSuite Chart	733	8,302	5448 (2141)	8,074
javaFig 1.43	161	2,108	1526 (971)	3,482
BLOAT	282	2,677	1255 (541)	6,623
JAX 6.3	309	2,754	1252 (579)	3,836
javac	210	1,512	1107 (406)	3,621
Res. System	2,332	21,495	12487 (6334)	23,640

Findings

- **Paper compares all 4 methods with RTA with regard to call graph construction**
- **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**

Findings, cont.

- The number of edges in the call graph is significantly reduced by XTA over RTA (.3%-29% fewer, 7% on average)
- 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 that become monomorphic

Findings

benchmark	RTA			XTA		
	unreached	mono	poly	unreached	mono	poly
Hanoi	34.0%	61.6%	4.4%	34.0%	62.7%	3.3%
Ice Browser	4.0%	91.4%	4.7%	4.0%	91.6%	4.5%
mBird	14.2%	73.4%	12.3%	17.4%	70.9%	11.7%
Cindy	49.3%	45.0%	5.7%	49.4%	45.5%	5.0%
CindyApplet	72.0%	24.6%	3.4%	72.3%	24.5%	3.2%
eSuite Sheet	28.1%	68.4%	3.5%	28.2%	69.1%	2.8%
eSuite Chart	13.3%	76.6%	10.1%	15.7%	76.0%	8.3%
javaFig 1.43	9.1%	87.1%	3.9%	9.7%	87.2%	3.1%
BLOAT	6.6%	82.4%	11.1%	7.0%	82.2%	10.8%
JAX 6.3	18.7%	75.9%	5.4%	18.9%	76.8%	4.3%
javac	3.0%	77.6%	19.4%	3.0%	77.7%	19.3%
Res. System	18.1%	72.0%	9.9%	18.2%	74.0%	7.9%
AVERAGE			7.8%			7.0%

Conclusions

- Using distinct reference representatives per method adds precision
- Using distinct fields per abstract object does not seem to add much precision
  - Note: other authors disagree with this finding
  - Possibilities include
    - no-fields,
    - fields of an abstract object per class,
    - fields of a representative of a group of object creation sites.