Lecture 4 – Class Hierarchy Analysis

- A type-based reference analysis used for inexpensive call graph construction
- Requires whole program, that is all class definitions with all of their defined methods
    - Is useful even if the call can not be resolved to a direct call – can use a "type-case" expression to resolve at runtime
    - Requires static types & inheritance structure
- Ecoop 1995 paper concerned mainly with efficiency of run-time resolution of virtual calls

J. Dean, G. Grove, C. Chambers, "Optimization of Object-oriented Programs Using Static Class Hierarchy Analysis" ECOOP 1995.

Class Hierarchy Analysis

- 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)*

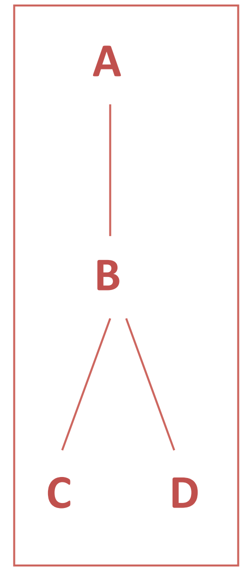
  and find out what methods may be called at a virtual call site
  - Makes assumption that entire inheritance hierarchy is available
    - Depending on its shape, might transform a virtual call into a direct call because there is only 1 choice of (matching) function
  - Ignores flow of control in program
  - Just using declared type information
  - Might be able to resolve call

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(){…}
}
```



Run-time call graph

Using CHA

- Use declared type of receiver, consult hierarchy for possible concrete receiver types
- For each concrete type of the receiver, find the local (or inherited) matching function
  - If there is only one function, for all the possible concrete types, then resolve the virtual call to a direct call at compile time
  - If there are only a few possible concrete types with different functions, then write a type-based case statement, querying the type of the concrete receiver, and executing the corresponding function
  - Otherwise, resolve at runtime.

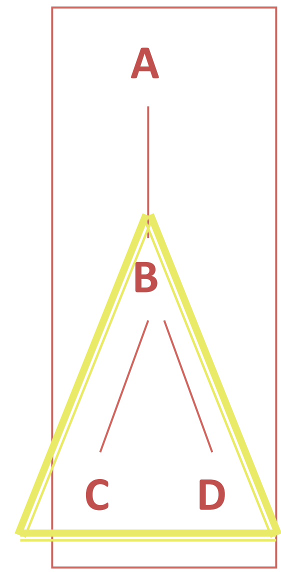
cf Frank Tip, OOPSLA' 00

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(){…}
}
```

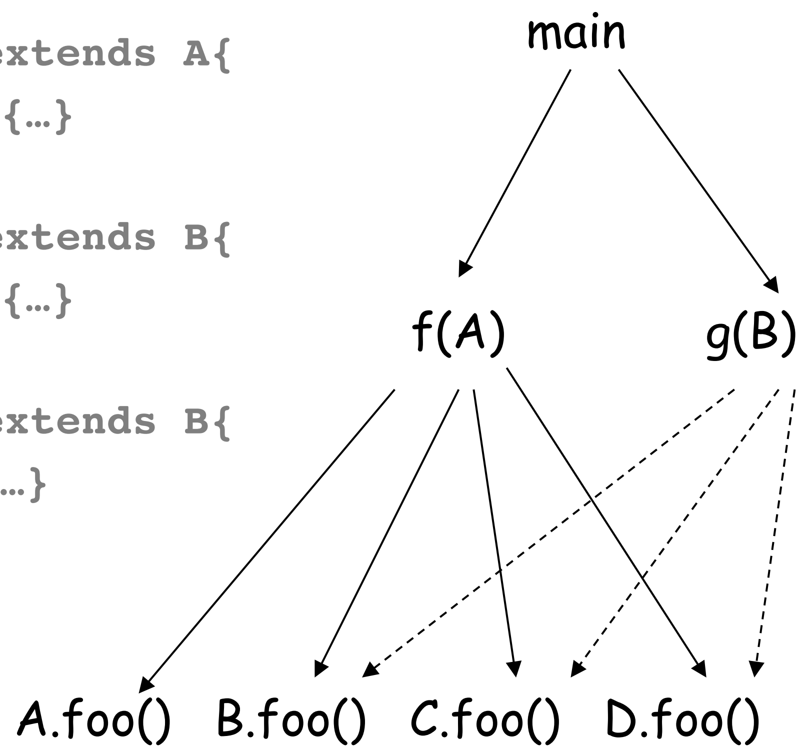


Cone(Declared_type(receiver))

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(){…}
}
```

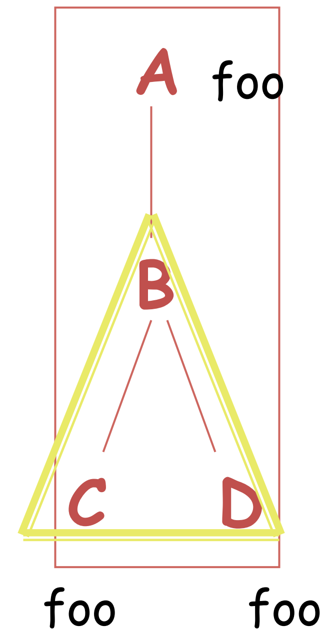


Call Graph

Example of a type-case translation

```
A a = new A()
…
s = a.foo();
```

```
if (a.class() == C){
   s = a.C::foo();}
else if (a.class() == D){
   s = a.D::foo();}
else s = A::foo();
```



What happens if at runtime the concrete object referred to by a is NOT of type A,B,C, or D?

Empirical Results

- Use of CHA (rather than intraprocedural analyses) to resolve virtual calls in Cecil resulted in
  - Visible speed improvements
  - Saved between 12-21% of space
- Code specialization improved performance better than CHA, but also increased code size
- Profile-guided prediction did best alone of all the techniques, but with CHA added, gained at least 45% in performance