EXERCISE 19

SUMMARY FUNCTION REVIEW

Write your name and answer the following on a piece of paper

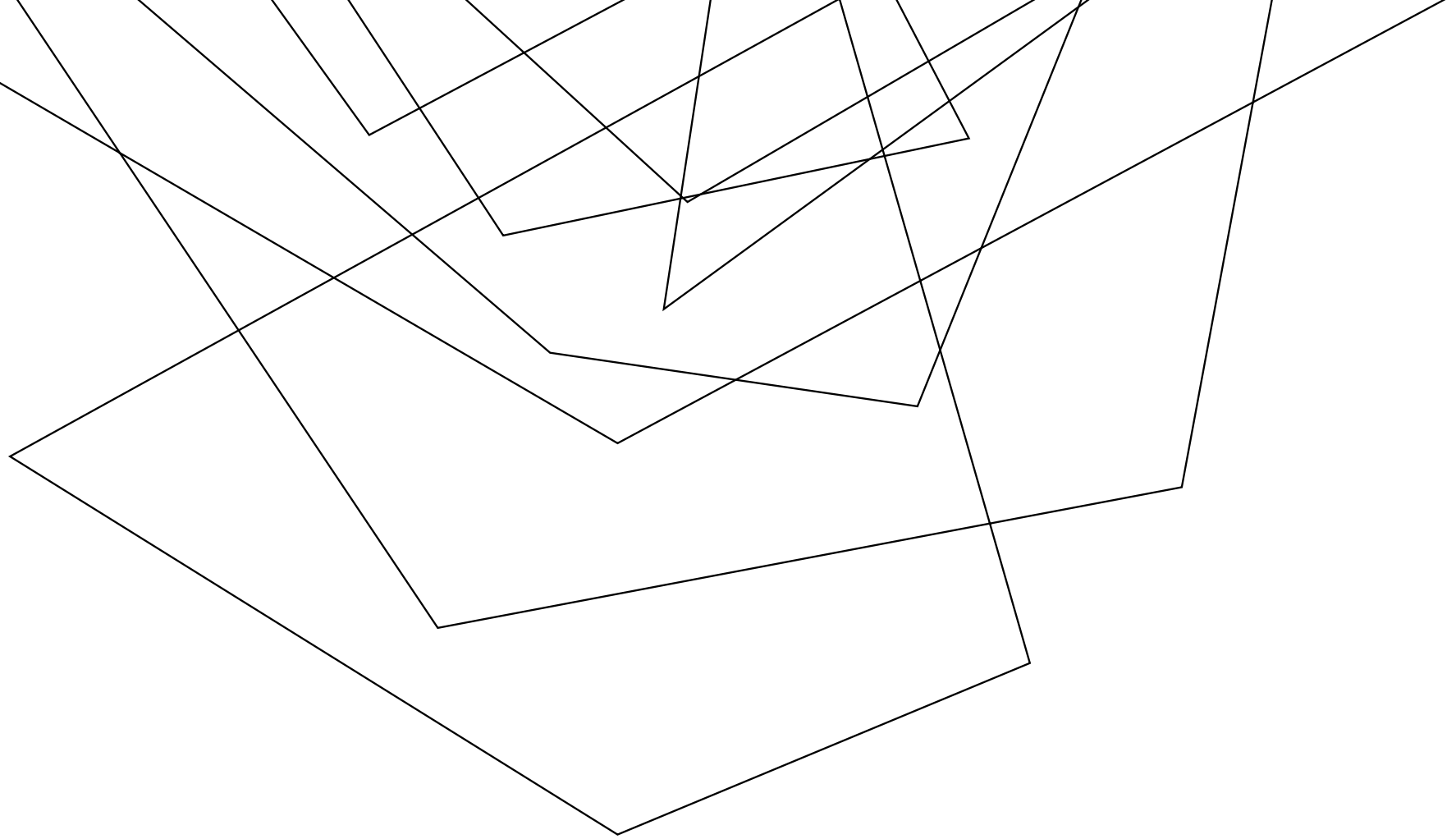
Compute GMOD / GREF for the below program

```
 1: int A,B,C,D;
 2: void baz(){
 3:   D = B;
 4:   A = C;
 5:   foo();
 6: }
 7: void bar(){
 8:   B = 2;
 9:   baz();
10: }
11: void foo(){
12:    A = 1;
13:    bar();
14: }
15: int main(){
16:    foo();
17:    foo();
18: }
```

EXERCISE 19 SOLUTION

SUMMARY FUNCTION REVIEW

ADMINISTRIVIA AND ANNOUNCEMENTS



CALL TARGETS

EECS 677: Software Security Evaluation

Drew Davidson

LAST TIME: SUMMARY FUNCTIONS

REVIEW: LAST LECTURE

CAN WE OVER-APPROXIMATE A FUNCTION'S EFFECT WITHOUT BUILDING THE SUPERGRAPH?

Motivation

- Allows "quick", modular analysis

Manifestations

- GMOD/GREF (what variables are touched?)
- Abstract transfer functions

foo(+) → T

foo(4);

```
foo ( int a; ) {
    return a + 1;
}
```

ABSTRACT TRANSFER FUNCTIONS

REVIEW: LAST LECTURE

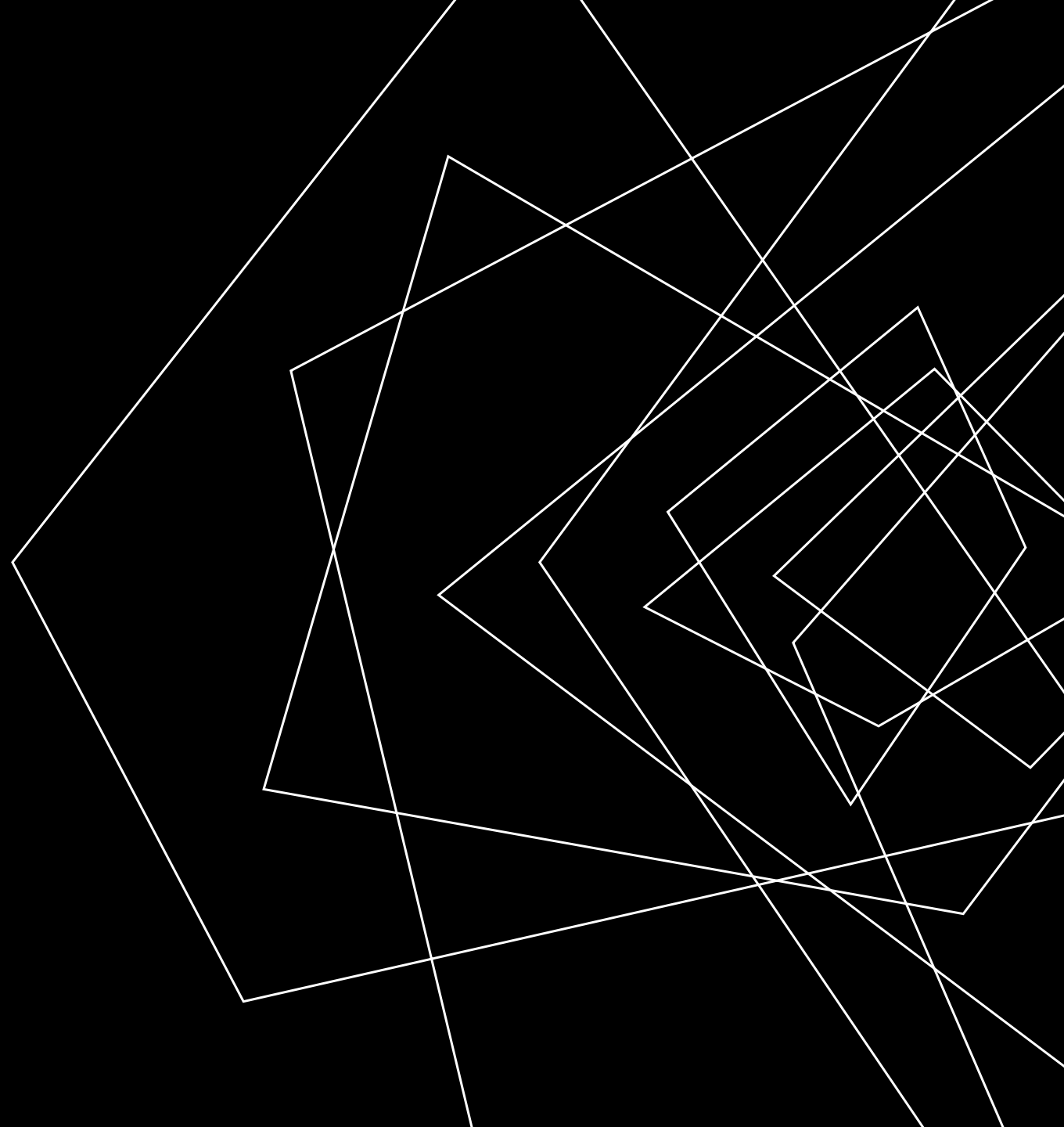
Can we over-Approximate a function's effect without building the supergraph?

OVERVIEW

HOW DO WE FIGURE OUT CONTROL TRANSFER TARGETS IN THE FIRST PLACE?

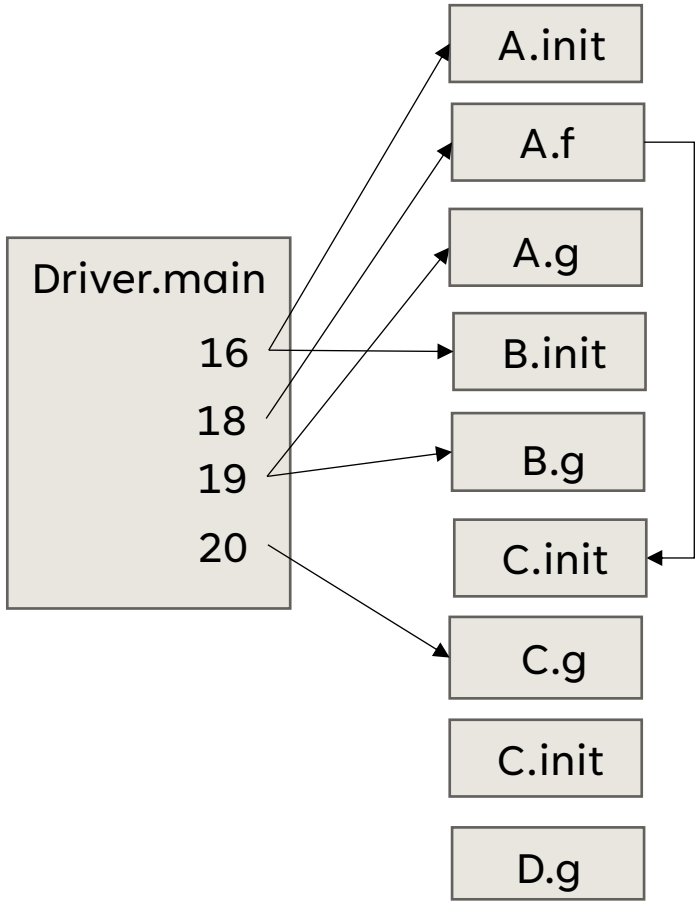
LECTURE OUTLINE

- Dynamic Dispatch
- Algorithms



DYNAMIC DISPATCH

A HISTORY OF COMPUTING




```
 1: class A{
 2:     public A f(){ return new C(); }
 3:     public String g(){ return "A"; }
 4: };
 5: class B : public A{
 6:     public String g(){ return "B"; }
 7: };
 8: class C : public A{
 9:     public String g(){ return "C"; }
10: };
11: class D : public A{
12:     public String g(){ return "D"; }
13: };
14: class Driver {
15:   public void main(String[] args){
16:     A[] aArr = {new A(), new B()};
17:     for (A a : aArr){
18:       A res = a.f();
19:       print(a.g());
20:       print(res.g());
21:     }
22:   }
23: };
```

DYNAMIC DISPATCH: GETS COMPLICATED!

A HISTORY OF COMPUTING

DYNAMIC DISPATCH: GETS COMPLICATED!

A HISTORY OF COMPUTING

Direct Calls

Not so bad

Indirect Calls

Quite a bit harder: multiple targets possible!

DYNAMIC DISPATCH: GETS COMPLICATED!

A HISTORY OF COMPUTING

Direct Calls

Not so bad

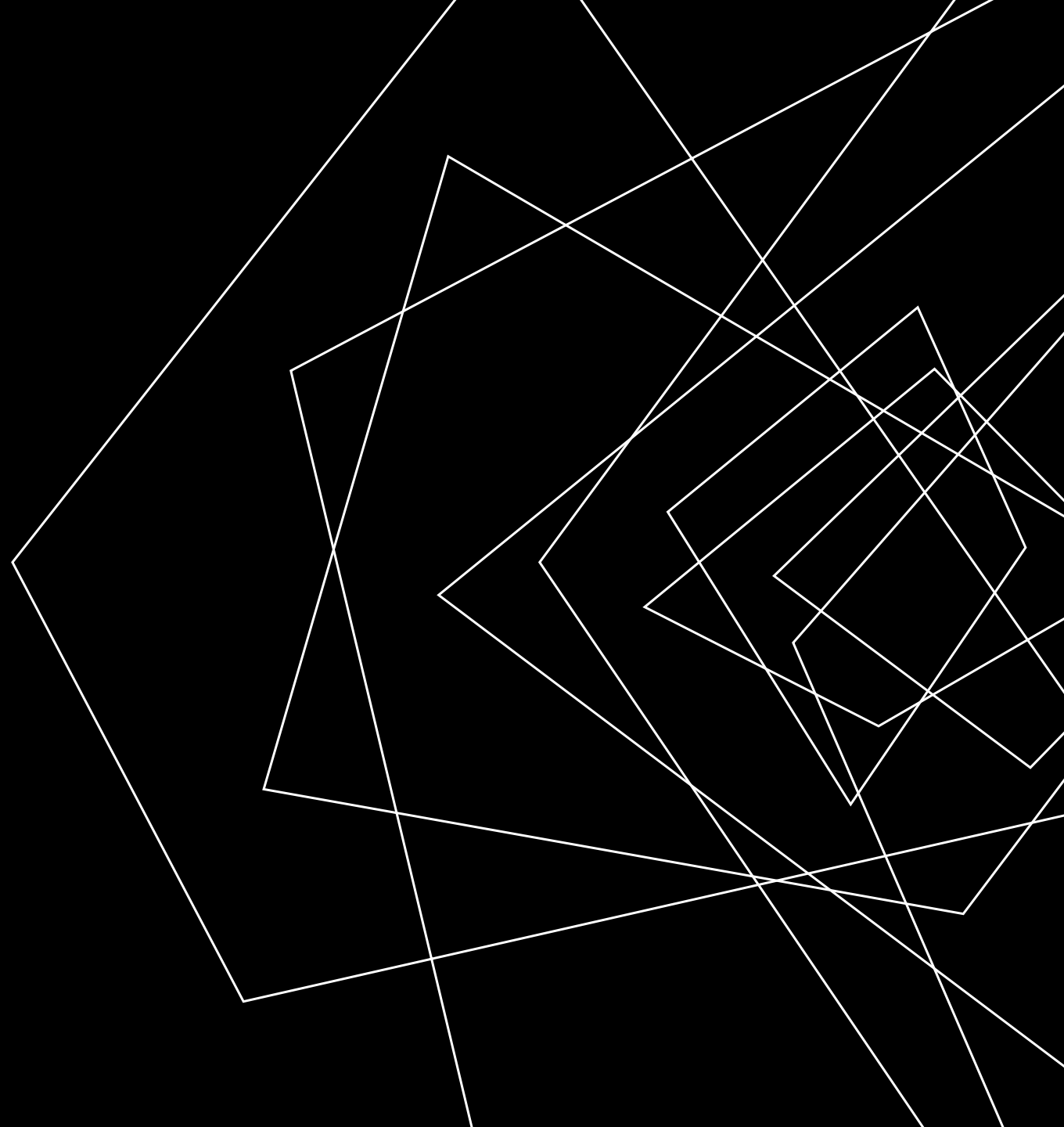
Indirect Calls

Quite a bit harder: multiple targets possible!

```
 1: class A{
 2:     public A f(){ return new C(); }
 3:     public String g(){ return "A"; }
 4: };
 5: class B : public A{
 6:     public String g(){ return "B"; }
 7: };
 8: class C : public A{
 9:     public String g(){ return "C"; }
10: };
11: class D : public A{
12:     public String g(){ return "D"; }
13: };
14: class Driver {
15:   public void main(String[] args){
16:     A[] aArr = {new A(), new B()};
17:     for (A a : aArr){
18:       A res = a.f();
19:       print(a.g());
20:       print(res.g());
21:     }
22:   }
23: };
```

LECTURE OUTLINE

- Dynamic Dispatch
- Algorithms

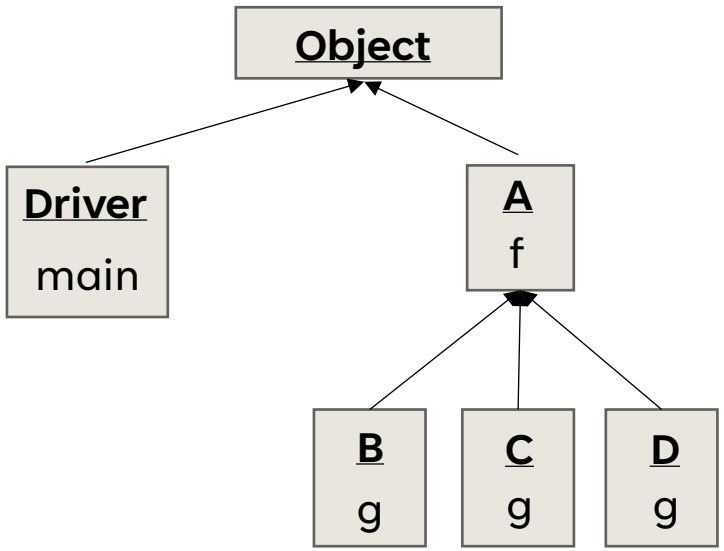


CLASS HIERARCHY ANALYSIS (CHA)

A HISTORY OF COMPUTING

CONSIDER THE SAFE OVER-APPROXIMATION

Treat call as declared type, or any subtype




```
 1: class A{
 2:     public A f(){ return new C(); }
 3:     public String g(){ return "A"; }
 4: };
 5: class B : public A{
 6:     public String g(){ return "B"; }
 7: };
 8: class C : public A{
 9:     public String g(){ return "C"; }
10: };
11: class D : public A{
12:     public String g(){ return "D"; }
13: };
14: class Driver {
15:   public void main(String[] args){
16:     A[] aArr = {new A(), new B()};
17:     for (A a : aArr){
18:       A res = a.f();
19:       print(a.g());
20:       print(res.g());
21:     }
22:   }
23: };
```

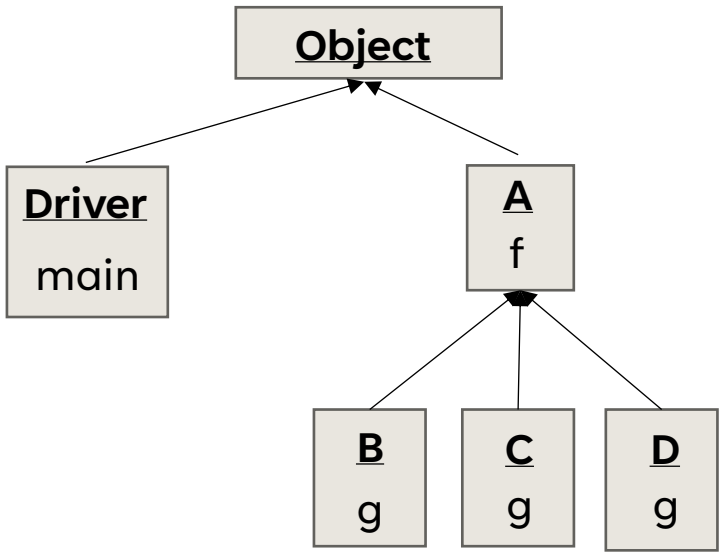
RAPID TYPE ANALYSIS (RTA)

A HISTORY OF COMPUTING

REFINEMENT OVER CHA

Consider only initialized classes

Consider only reachable code




```
 1: class A{
 2:     public A f(){ return new C(); }
 3:     public String g(){ return "A"; }
 4: };
 5: class B : public A{
 6:     public String g(){ return "B"; }
 7: };
 8: class C : public A{
 9:     public String g(){ return "C"; }
10: };
11: class D : public A{
12:     public String g(){ return "D"; }
13: };
14: class Driver {
15:   public void main(String[] args){
16:     A[] aArr = {new A(), new B()};
17:     for (A a : aArr){
18:       A res = a.f();
19:       print(a.g());
20:       print(res.g());
21:     }
22:   }
23: };
```

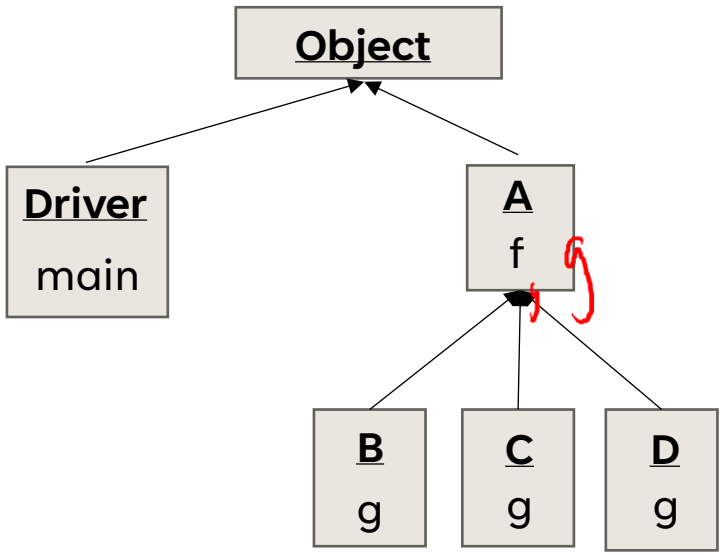
CHA

A HISTORY OF COMPUTING

REFINEMENT OVER CHA

Consider only initialized classes

Consider only reachable code




```
1: class A{
2:     public A f(){ return new C(); }
3:     public String g(){ return "A"; }
4: };
5: class B : public A{
6:     public String g(){ return "B"; }
7: };
8: class C : public A{
9:     public String g(){ return "C"; }
10: };
11: class D : public A{
12:     public String g(){ return "D"; }
13: };
14: class Driver {
15:   public void main(String[] args){
16:     A[] aArr = {new A(), new B()};
17:     for (A a : aArr){
18:       A res = a.f();
19:       print(a.g());
20:       print(res.g());
21:     }
22:   }
23: };
```

RAPID TYPE ANALYSIS (RTA)

A HISTORY OF COMPUTING

RTA = call graph of functions (initially edgeless)
CHA = call graph via class hierarchy analysis
W = worklist
W.push(main)
while not W.empty:
  M = pop W
  T = allocated types in M
  T = T U allocated types in RTA callers of M
  foreach callsite(C) in M
    if C is statically-dispatched:
      add edge C to C's static target
    else:
      M' = methods called from M in CHA
      M' = M' intersect functions declared in T or T-supertypes
      add edge from M to each M'
      W.pushAll(M')

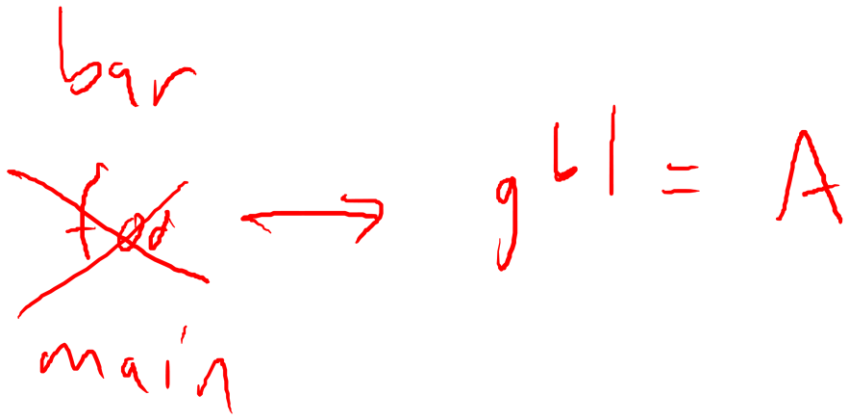
RAPID TYPE ANALYSIS (RTA)

A HISTORY OF COMPUTING

AN UNSOUND ANALYSIS!

```
 1: public static Object gbl;
 2:
 3: public static void main(String[] args){
 4:   foo();
 5:   bar();
 6: }
 7:
 8: public static void foo(){
 9:   Object o = new A();
10:   gbl = o;
11: }
12:
13: public static void bar(){
14:   gbl.toString();
15: }
```

RTA will not include an edge from bar to toString because neither bar or its callers (main) allocated any instance that toString could be called on



RAPID TYPE ANALYSIS (RTA)

A HISTORY OF COMPUTING

AN UNSOUND ANALYSIS!

```
 1: public static Object gbl;
 2:
 3: public static void main(String[] args){
 4:   Object o = foo();
 5:   bar(o);
 6: }
 7:
 8: public static Object foo(){
 9:   return new A();
10:   gbl = o;
11: }
12:
13: public static void bar(Object o){
14:   o.toString();
15: }
```

Call edge to A's toString missing!

Neither bar or its callers (main) allocated a type of A

BEYOND RTA
A HISTORY OF COMPUTING

Assumptions to strengthen analysis

Type safety?

Might not be a safe assumption

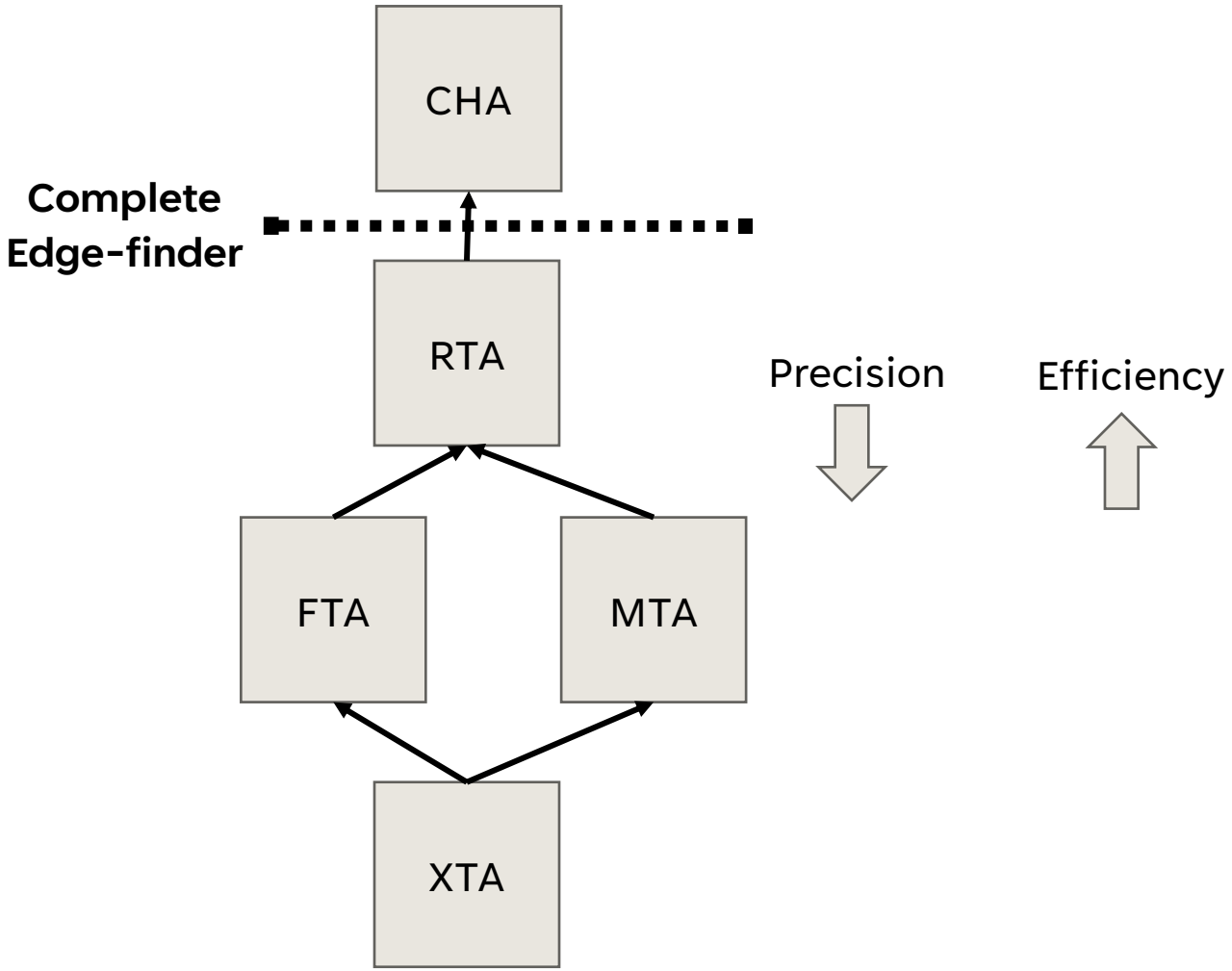
FTA adds the constraint that any method that reads from a field can inherit the field-compatible allocated types of any method that could write to that field.

MTA adds the constraint that types allocated in a method and then passed to a method through a parameter should be compatible with the called method's parameter types. MTA also adds the constraint that the return type of each called method be added to the set of allocated types.

XTA: add both the constraints of MTA an FTA

COMPARING CALL GRAPH ANALYSES

A HISTORY OF COMPUTING



WRAP-UP