

EXERCISE

SUMMARY FUNCTION REVIEW

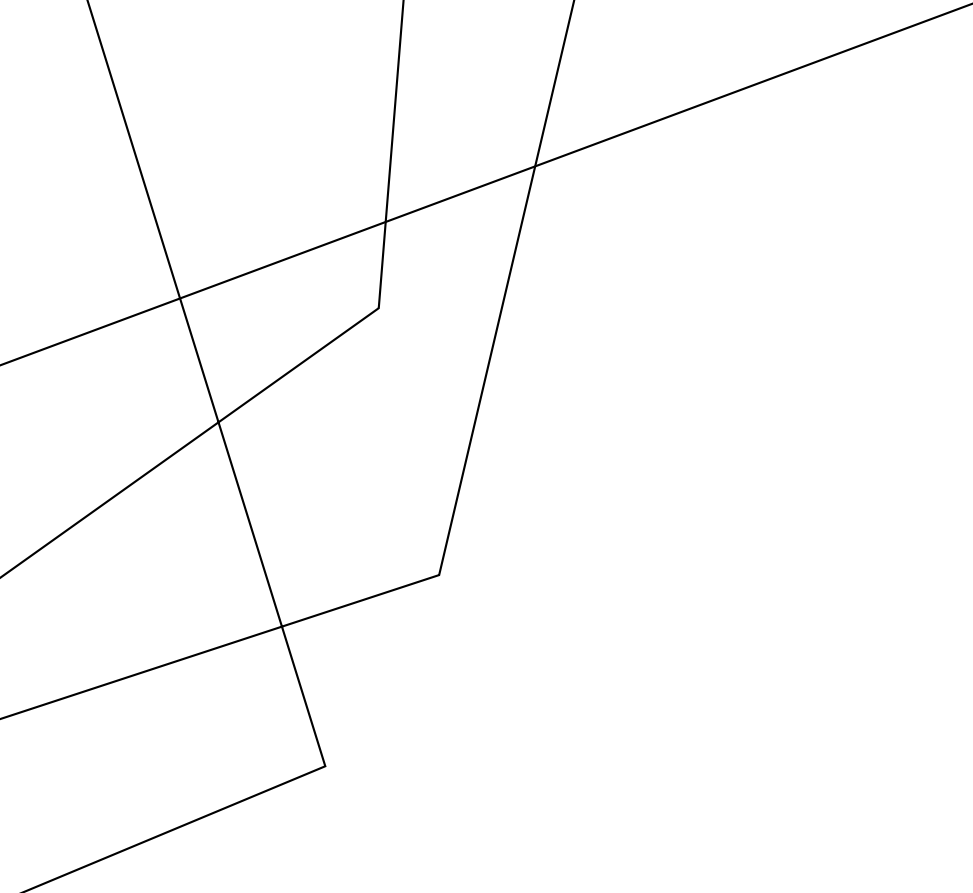
Write your name and answer the following on a piece of paper

Compute GMOD / GREF for the below program

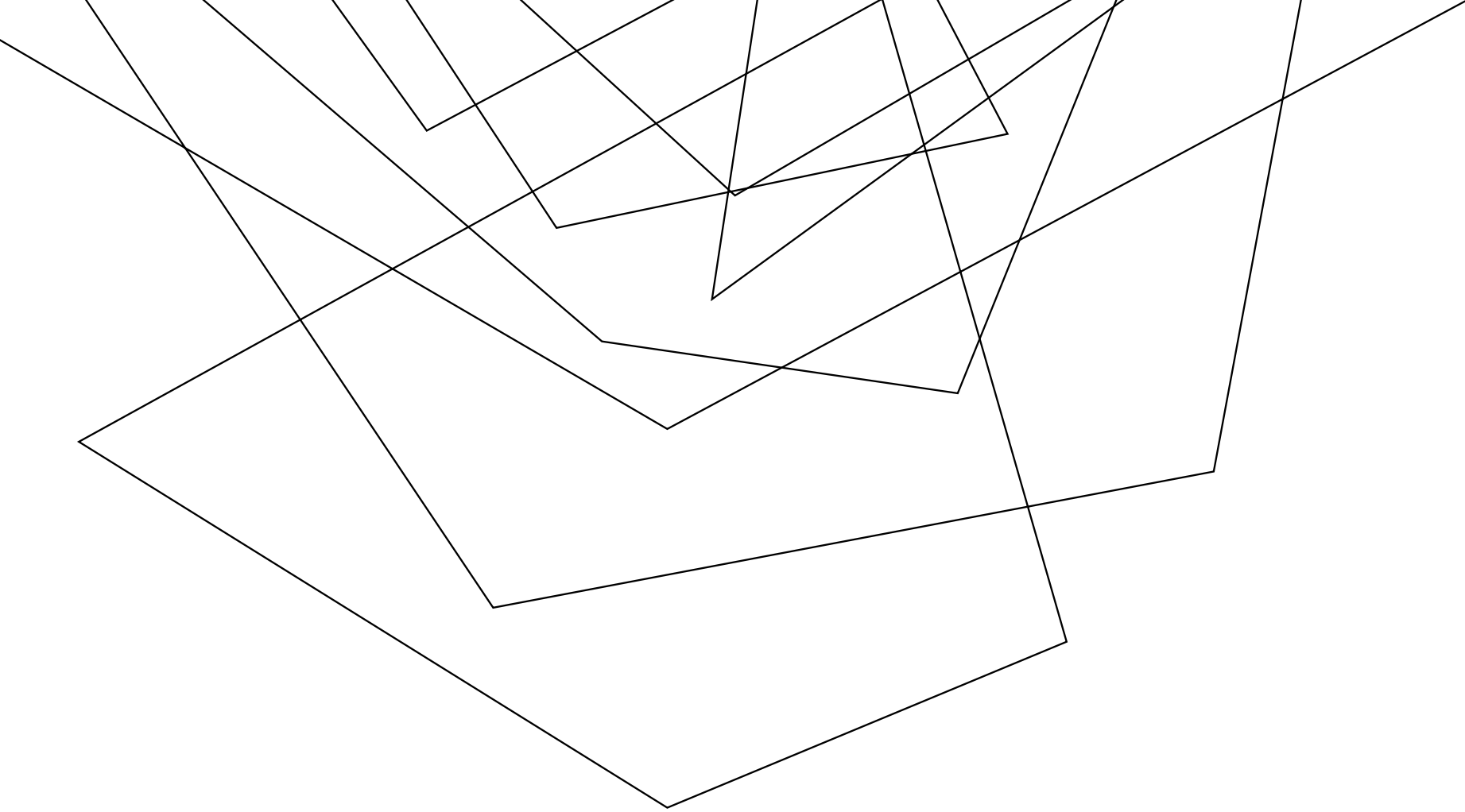
```
1: int A,B,C,D;
2: void baz() {
3:     D = B;
4:     A = C;
5:     foo();
6: }
7: void bar() {
8:     B = 2;
9:     baz();
10: }
11: void foo() {
12:     A = 1;
13:     bar();
14: }
15: int main() {
16:     foo();
17:     foo();
18: }
```

EXERCISE SOLUTION

SUMMARY FUNCTION REVIEW

Abstract geometric lines in the top left corner, consisting of several thin black lines forming a series of overlapping, irregular shapes that resemble a stylized architectural drawing or a map projection.

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

```
int k(int e, int &b) {
```

a = 2;

b = 3;

}

```
int m() {
```

k(c, d);

→ c has the original value
d has the value 3

LAST TIME: SUMMARY FUNCTIONS

REVIEW: LAST LECTURE

Operate on the “complex” callgraph (i.e. sub-nodes for call sites)

GREF Computation

$$\text{GREF}_{\text{fn}}(F) = \text{IREF}_{\text{fn}}(F)$$

$$\text{GREF}_{\text{call}}(C) = \{\}$$

worklist = { all calls and fns }

while(worklist not empty):

 e = pop worklist

 if element e is a call c in function s to function t:

$\text{GREF}_{\text{call}}(c) = \text{GREF}_{\text{fn}}(t)$ with formals mapped to actuals

 if $\text{GREF}_{\text{call}}(c)$ updated, add s to the worklist

 else if element e is a function f:

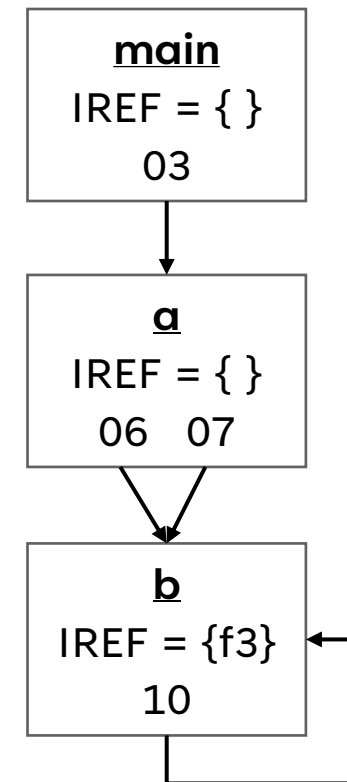
$\text{GREF}_{\text{fn}}(f) = \bigcup \text{GREF}_{\text{call}}(c)$ for all call sites c in f $\cup$ $\text{IREF}(f)$

 if $\text{GREF}_{\text{fn}}(f)$ updated, add all call sites to f to worklist

```

01: int g1, g2, p3;
02: void main() {
03:   a(v1);
04: }
05: void a( int f1){
06:   b(v2, v3);
07:   b(v4, v5);
08: }
09: void b( int f2, int f3){
10:   print(f3);
11:   b(g1, g2);
12: }

```



LAST TIME: SUMMARY FUNCTIONS

REVIEW: LAST LECTURE

GREF Computation

$GREF_{fn}(F) = IREF_{fn}(F)$

$GREF_{call}(C) = \{\}$

worklist = { all calls and fns }

while(worklist not empty):

 e = pop worklist

 if element e is a call c in function s to function t:

$GREF_{call}(c) = GREF_{fn}(t)$ with formals mapped to actuals

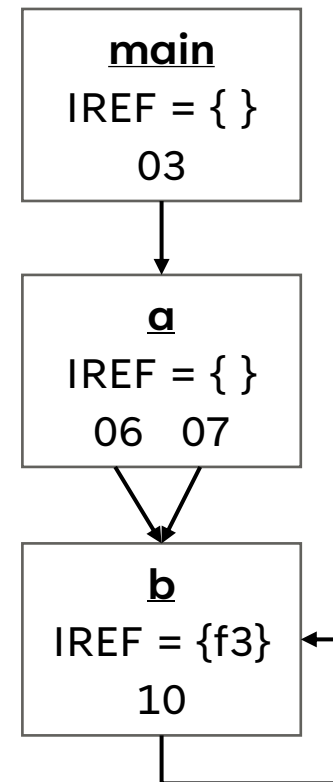
 if $GREF_{call}(c)$ updated, add s to the worklist

 else if element e is a function f:

$GREF_{fn}(f) = \bigcup GREF_{call}(c)$ for all call sites c in f $\cup IREF(f)$

 if $GREF_{fn}(f)$ updated, add all call sites to f to worklist

```
01: int g1, g2, p3;
02: void main() {
03:   a(v1);
04: }
05: void a( int f1){
06:   b(v2, v3);
07:   b(v4, v5);
08: }
09: void b( int f2, int f3){
10:   print(f3);
11:   b(g1, g2);
12: }
```



Worklist: { **b** 11 main a 03 06 07 }

	main	a	b	03	06	07	11
GREF	{ }	{ }	{f3}	{ }	{ }	{ }	{ }

LAST TIME: SUMMARY FUNCTIONS

REVIEW: LAST LECTURE

GREF Computation

$GREF_{fn}(F) = IREF_{fn}(F)$

$GREF_{call}(C) = \{\}$

worklist = { all calls and fns }

while(worklist not empty):

 e = pop worklist

 if element e is a call c in function s to function t:

$GREF_{call}(c) = GREF_{fn}(t)$ with formals mapped to actuals

 if $GREF_{call}(c)$ updated, add s to the worklist

 else if element e is a function f:

$GREF_{fn}(f) = \bigcup GREF_{call}(c)$ for all call sites c in f $\cup IREF(f)$

 if $GREF_{fn}(f)$ updated, add all call sites to f to worklist

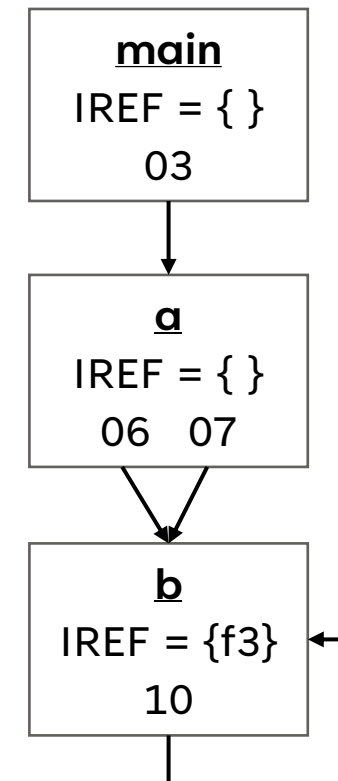
Worklist: { 11 main a 03 06 07 }

	main	a	b	03	06	07	11
GREF	{ }	{ }	{f3}	{ }	{ }	{ }	{ }

```

01: int g1,g2, p3;
02: void main() {
03:   a(v1);
04: }
05: void a( int f1){
06:   b(v2, v3);
07:   b(v4, v5);
08: }
09: void b( int f2, int f3){
10:   print(f3);
11:   b(g1, g2);
12: }

```



Process b

$$\begin{aligned}
 GREF_{fn}(b) &= \bigcup GREF_{call}(11) \cup IREF(b) \\
 &= \{\} \cup \{f3\} \\
 &= \{f3\} \quad (\text{no update, don't change worklist})
 \end{aligned}$$

LAST TIME: SUMMARY FUNCTIONS

REVIEW: LAST LECTURE

GREF Computation

$GREF_{fn}(F) = IREF_{fn}(F)$

$GREF_{call}(C) = \{ \}$

worklist = { all calls and fns }

while(worklist not empty):

 e = pop worklist

 if element e is a call c in function s to function t:

$GREF_{call}(c) = GREF_{fn}(t)$ with formals mapped to actuals

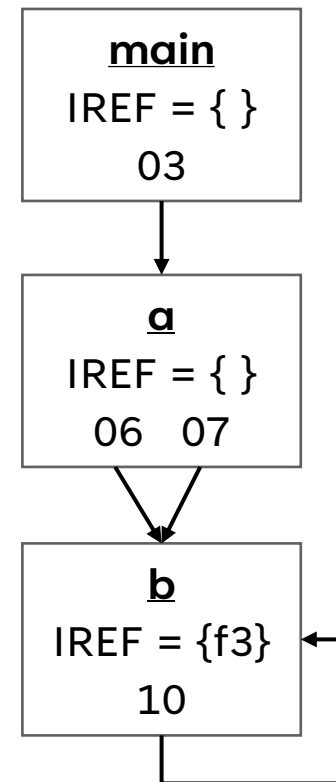
 if $GREF_{call}(c)$ updated, add s to the worklist

 else if element e is a function f:

$GREF_{fn}(f) = \bigcup GREF_{call}(c)$ for all call sites c in f $\cup IREF(f)$

 if $GREF_{fn}(f)$ updated, add all call sites to f to worklist

```
01: int g1, g2, p3;
02: void main() {
03:   a(v1);
04: }
05: void a( int f1){
06:   b(v2, v3);
07:   b(v4, v5);
08: }
09: void b( int f2, int f3){
10:   print(f3);
11:   b(g1, g2);
12: }
```



Worklist: { **11** main a 03 06 07 }

	main	a	b	03	06	07	11
GREF	{ }	{ }	{f3}	{ }	{ }	{ }	{ }

LAST TIME: SUMMARY FUNCTIONS

REVIEW: LAST LECTURE

GREF Computation

$$\text{GREF}_{\text{fn}}(F) = \text{IREF}_{\text{fn}}(F)$$

$$\text{GREF}_{\text{call}}(C) = \{ \}$$

worklist = { all calls and fns }

while(worklist not empty):

 e = pop worklist

 if element e is a call c in function s to function t:

$\text{GREF}_{\text{call}}(c) = \text{GREF}_{\text{fn}}(t)$ with formals mapped to actuals

 if $\text{GREF}_{\text{call}}(c)$ updated, add s to the worklist

 else if element e is a function f:

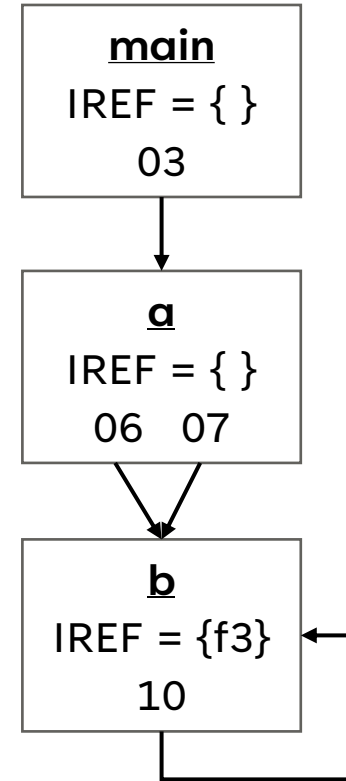
$\text{GREF}_{\text{fn}}(f) = \bigcup \text{GREF}_{\text{call}}(c)$ for all call sites c in f $\cup \text{IREF}(f)$

 if $\text{GREF}_{\text{fn}}(f)$ updated, add all call sites to f to worklist

```

01: int g1, g2, p3;
02: void main() {
03:   a(v1);
04: }
05: void a( int f1){
06:   b(v2, v3);
07:   b(v4, v5);
08: }
09: void b( int f2, int f3){
10:   print(f3);
11:   b(g1, g2);
12: }

```



Worklist: { main a 03 06 07 **b** }

	main	a	b	03	06	07	11
IREF	{ }	{ }	{f3}	{ }	{ }	{ }	~~{ }~~
							{ g2 }

Process 11

$\text{GREF}_{\text{call}}(11) = \text{GREF}_{\text{fn}}(b)$ with g1/f2 and g2/f3

$$= \{f3\} \text{ g2}$$

$$= \{g2\}$$

(update, add b add worklist)

LAST TIME: SUMMARY FUNCTIONS

REVIEW: LAST LECTURE

GREF Computation

$GREF_{fn}(F) = IREF_{fn}(F)$

$GREF_{call}(C) = \{ \}$

worklist = { all calls and fns }

while(worklist not empty):

 e = pop worklist

 if element e is a call c in function s to function t:

$GREF_{call}(c) = GREF_{fn}(t)$ with formals mapped to actuals

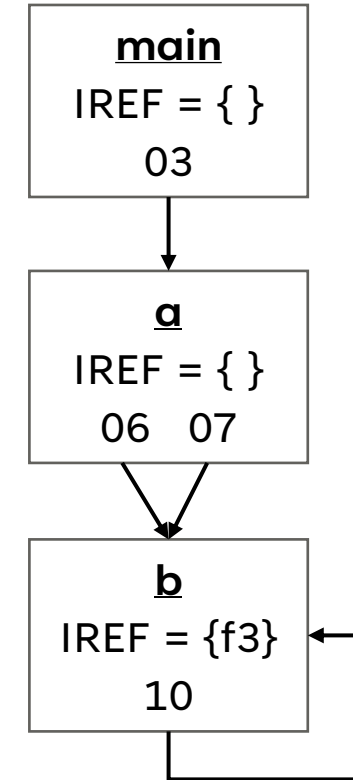
 if $GREF_{call}(c)$ updated, add s to the worklist

 else if element e is a function f:

$GREF_{fn}(f) = \bigcup GREF_{call}(c)$ for all call sites c in f $\cup IREF(f)$

 if $GREF_{fn}(f)$ updated, add all call sites to f to worklist

```
01: int g1,g2, p3;
02: void main() {
03:   a(v1);
04: }
05: void a( int f1){
06:   b(v2, v3);
07:   b(v4, v5);
08: }
09: void b( int f2, int f3){
10:   print(f3);
11:   b(g1, g2);
12: }
```



Worklist: { main a 03 06 07 b }

	main	a	b	03	06	07	11
GREF	{ }	{ }	{f3}	{ }	{ }	{ }	{ g2 }

LAST TIME: SUMMARY FUNCTIONS

REVIEW: LAST LECTURE

GREF Computation

$GREF_{fn}(F) = IREF_{fn}(F)$

$GREF_{call}(C) = \{\}$

worklist = { all calls and fns }

while(worklist not empty):

 e = pop worklist

 if element e is a call c in function s to function t:

$GREF_{call}(c) = GREF_{fn}(t)$ with formals mapped to actuals

 if $GREF_{call}(c)$ updated, add s to the worklist

 else if element e is a function f:

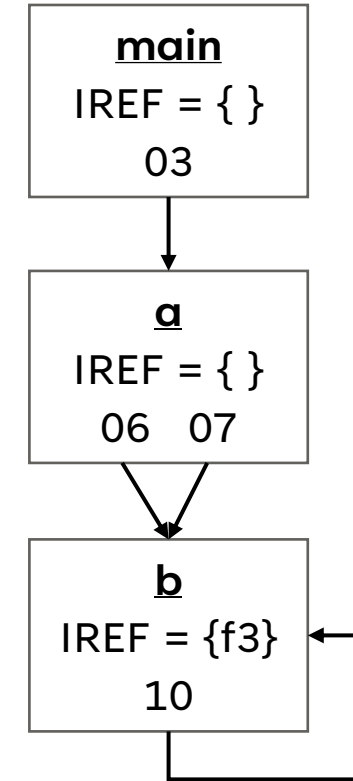
$GREF_{fn}(f) = \bigcup GREF_{call}(c)$ for all call sites c in f $\cup IREF(f)$

 if $GREF_{fn}(f)$ updated, add all call sites to f to worklist

```

01: int g1, g2, p3;
02: void main() {
03:   a(v1);
04: }
05: void a( int f1){
06:   b(v2, v3);
07:   b(v4, v5);
08: }
09: void b( int f2, int f3){
10:   print(f3);
11:   b(g1, g2);
12: }

```



Worklist: { a 03 06 07 b }

	main	a	b	03	06	07	11
--	------	---	---	----	----	----	----

GREF	{ }	{ }	{f3}	{ }	{ }	{ }	{ g2 }
------	-----	-----	------	-----	-----	-----	--------

Process main

$$GREF_{fn}(main) = \bigcup GREF_{call}(03) \cup IREF(main)$$

$$= \{\} \cup \{\}$$

(no update)

LAST TIME: SUMMARY FUNCTIONS

REVIEW: LAST LECTURE

GREF Computation

$GREF_{fn}(F) = IREF_{fn}(F)$

$GREF_{call}(C) = \{ \}$

worklist = { all calls and fns }

while(worklist not empty):

 e = pop worklist

 if element e is a call c in function s to function t:

$GREF_{call}(c) = GREF_{fn}(t)$ with formals mapped to actuals

 if $GREF_{call}(c)$ updated, add s to the worklist

 else if element e is a function f:

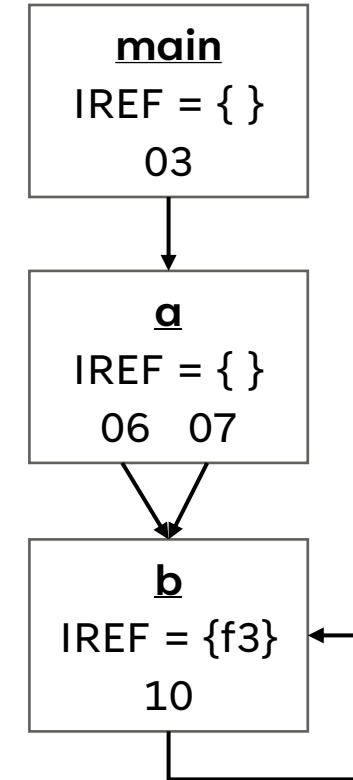
$GREF_{fn}(f) = \bigcup GREF_{call}(c)$ for all call sites c in f $\cup IREF(f)$

 if $GREF_{fn}(f)$ updated, add all call sites to f to worklist

```

01: int g1,g2, p3;
02: void main() {
03:   a(v1);
04: }
05: void a( int f1){
06:   b(v2, v3);
07:   b(v4, v5);
08: }
09: void b( int f2, int f3){
10:   print(f3);
11:   b(g1, g2);
12: }

```



Worklist: { **a** 03 06 07 b }

	main	a	b	03	06	07	11
GREF	{ }	{ }	{f3}	{ }	{ }	{ }	{ g2 }

LAST TIME: SUMMARY FUNCTIONS

REVIEW: LAST LECTURE

GREF Computation

$GREF_{fn}(F) = IREF_{fn}(F)$

$GREF_{call}(C) = \{\}$

worklist = { all calls and fns }

while(worklist not empty):

 e = pop worklist

 if element e is a call c in function s to function t:

$GREF_{call}(c) = GREF_{fn}(t)$ with formals mapped to actuals

 if $GREF_{call}(c)$ updated, add s to the worklist

 else if element e is a function f:

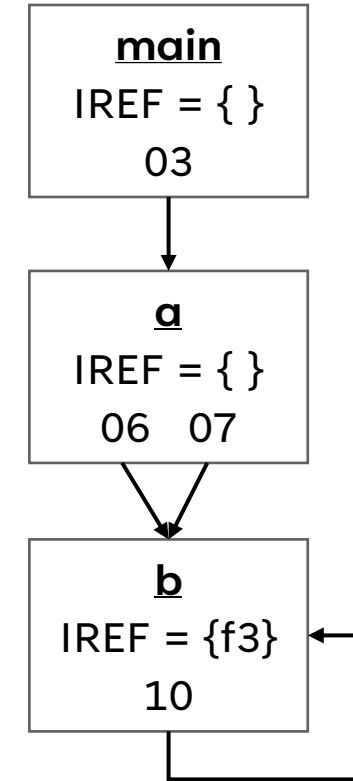
$GREF_{fn}(f) = \bigcup GREF_{call}(c)$ for all call sites c in f $\cup IREF(f)$

 if $GREF_{fn}(f)$ updated, add all call sites to f to worklist

```

01: int g1, g2, p3;
02: void main() {
03:   a(v1);
04: }
05: void a( int f1){
06:   b(v2, v3);
07:   b(v4, v5);
08: }
09: void b( int f2, int f3){
10:   print(f3);
11:   b(g1, g2);
12: }

```



Worklist: { 03 06 07 b }

	main	a	b	03	06	07	11
GREF	{ }	{ }	{f3}	{ }	{ }	{ }	{ g2 }

Process a

$$GREF_{fn}(a) = GREF_{call}(06) \cup GREF_{call}(07) \cup IREF(a)$$

$$= \{ \} \cup \{ \} \cup \{ \}$$

No update

LAST TIME: SUMMARY FUNCTIONS

REVIEW: LAST LECTURE

GREF Computation

$GREF_{fn}(F) = IREF_{fn}(F)$

$GREF_{call}(C) = \{ \}$

worklist = { all calls and fns }

while(worklist not empty):

 e = pop worklist

 if element e is a call c in function s to function t:

$GREF_{call}(c) = GREF_{fn}(t)$ with formals mapped to actuals

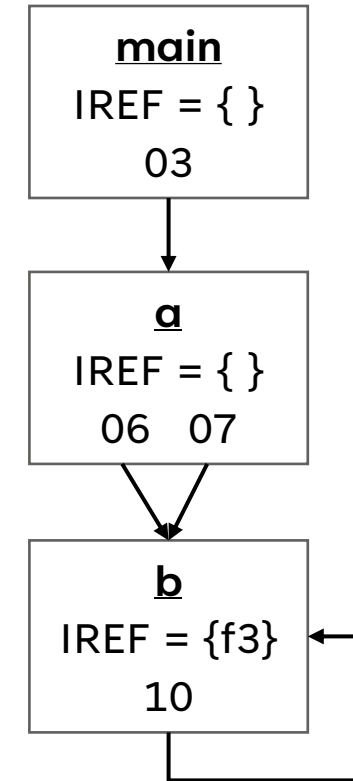
 if $GREF_{call}(c)$ updated, add s to the worklist

 else if element e is a function f:

$GREF_{fn}(f) = \bigcup GREF_{call}(c)$ for all call sites c in f $\cup IREF(f)$

 if $GREF_{fn}(f)$ updated, add all call sites to f to worklist

```
01: int g1,g2, p3;
02: void main() {
03:   a(v1);
04: }
05: void a( int f1){
06:   b(v2, v3);
07:   b(v4, v5);
08: }
09: void b( int f2, int f3){
10:   print(f3);
11:   b(g1, g2);
12: }
```



Worklist: { 03 06 07 b }

	main	a	b	03	06	07	11
GREF	{ }	{ }	{f3}	{ }	{ }	{ }	{ g2 }

LAST TIME: SUMMARY FUNCTIONS

REVIEW: LAST LECTURE

GREF Computation

$GREF_{fn}(F) = IREF_{fn}(F)$

$GREF_{call}(C) = \{ \}$

worklist = { all calls and fns }

while(worklist not empty):

 e = pop worklist

 if element e is a call c in function s to function t:

$GREF_{call}(c) = GREF_{fn}(t)$ with formals mapped to actuals

 if $GREF_{call}(c)$ updated, add s to the worklist

 else if element e is a function f:

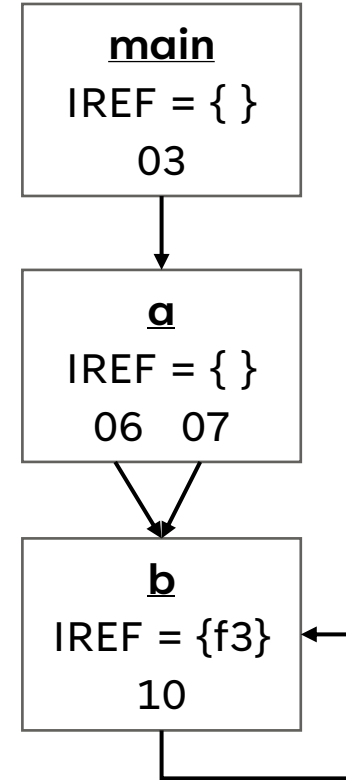
$GREF_{fn}(f) = \bigcup GREF_{call}(c)$ for all call sites c in f $\cup IREF(f)$

 if $GREF_{fn}(f)$ updated, add all call sites to f to worklist

```

01: int g1, g2, p3;
02: void main() {
03:   a(v1);
04: }
05: void a( int f1){
06:   b(v2, v3);
07:   b(v4, v5);
08: }
09: void b( int f2, int f3){
10:   print(f3);
11:   b(g1, g2);
12: }

```



Process 03

$GREF_{call}(03) = GREF_{fn}(a)$ with v1/f1
= { }

(no update)

Worklist: {				06 07 b}			
	main	a	b	03	06	07	11
GREF	{ }	{ }	{f3}	{ }	{ }	{ }	{ g2 }

LAST TIME: SUMMARY FUNCTIONS

REVIEW: LAST LECTURE

GREF Computation

$GREF_{fn}(F) = IREF_{fn}(F)$

$GREF_{call}(C) = \{ \}$

worklist = { all calls and fns }

while(worklist not empty):

 e = pop worklist

 if element e is a call c in function s to function t:

$GREF_{call}(c) = GREF_{fn}(t)$ with formals mapped to actuals

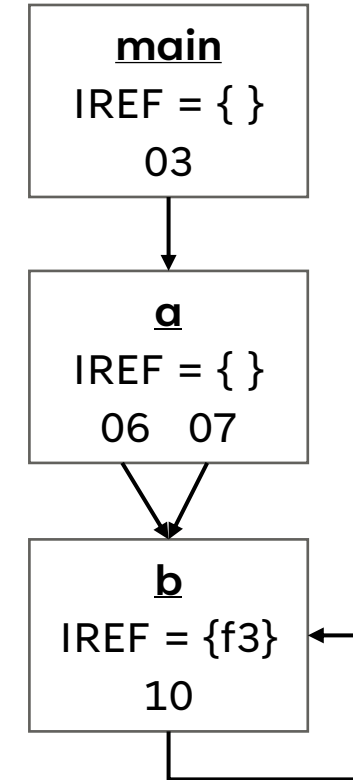
 if $GREF_{call}(c)$ updated, add s to the worklist

 else if element e is a function f:

$GREF_{fn}(f) = \bigcup GREF_{call}(c)$ for all call sites c in f $\cup IREF(f)$

 if $GREF_{fn}(f)$ updated, add all call sites to f to worklist

```
01: int g1, g2, p3;
02: void main() {
03:   a(v1);
04: }
05: void a( int f1){
06:   b(v2, v3);
07:   b(v4, v5);
08: }
09: void b( int f2, int f3){
10:   print(f3);
11:   b(g1, g2);
12: }
```



Worklist: { 06 07 b }

	main	a	b	03	06	07	11
GREF	{ }	{ }	{f3}	{ }	{ }	{ }	{ g2 }

LAST TIME: SUMMARY FUNCTIONS

REVIEW: LAST LECTURE

GREF Computation

$GREF_{fn}(F) = IREF_{fn}(F)$

$GREF_{call}(C) = \{ \}$

worklist = { all calls and fns }

while(worklist not empty):

 e = pop worklist

 if element e is a call c in function s to function t:

$GREF_{call}(c) = GREF_{fn}(t)$ with formals mapped to actuals

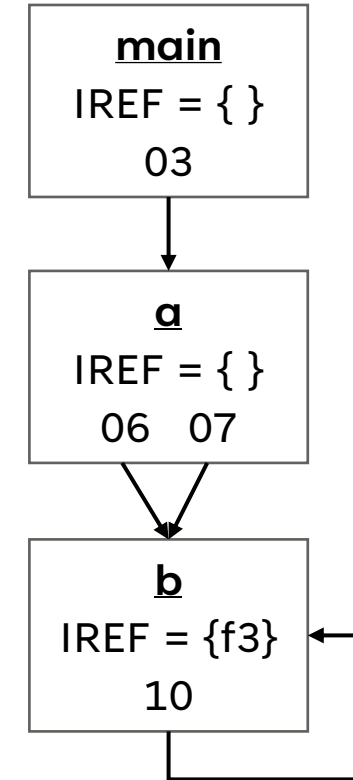
 if $GREF_{call}(c)$ updated, add s to the worklist

 else if element e is a function f:

$GREF_{fn}(f) = \bigcup GREF_{call}(c)$ for all call sites c in f $\cup IREF(f)$

 if $GREF_{fn}(f)$ updated, add all call sites to f to worklist

```
01: int g1, g2, p3;
02: void main() {
03:   a(v1);
04: }
05: void a( int f1){
06:   b(v2, v3);
07:   b(v4, v5);
08: }
09: void b( int f2, int f3){
10:   print(f3);
11:   b(g1, g2);
12: }
```



Worklist: { 07 b **a** }

	main	a	b	03	06	07	11
GREF	{ }	{ }	{f3}	{ }	~~{ }~~ {v3}	{ }	{ g2 }

Process 06

$GREF_{call}(06) = GREF_{fn}(b)$ with v2/f2 and v3/f3

= { ~~f3~~ } **v3**

= {v3}

(update, add a to worklist)

LAST TIME: SUMMARY FUNCTIONS

REVIEW: LAST LECTURE

GREF Computation

$GREF_{fn}(F) = IREF_{fn}(F)$

$GREF_{call}(C) = \{ \}$

worklist = { all calls and fns }

while(worklist not empty):

 e = pop worklist

 if element e is a call c in function s to function t:

$GREF_{call}(c) = GREF_{fn}(t)$ with formals mapped to actuals

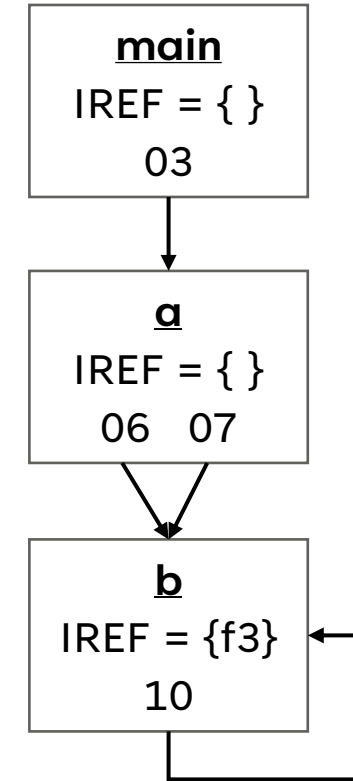
 if $GREF_{call}(c)$ updated, add s to the worklist

 else if element e is a function f:

$GREF_{fn}(f) = \bigcup GREF_{call}(c)$ for all call sites c in f $\cup IREF(f)$

 if $GREF_{fn}(f)$ updated, add all call sites to f to worklist

```
01: int g1,g2, p3;
02: void main() {
03:   a(v1);
04: }
05: void a( int f1){
06:   b(v2, v3);
07:   b(v4, v5);
08: }
09: void b( int f2, int f3){
10:   print(f3);
11:   b(g1, g2);
12: }
```



Worklist: { 07 b a }

	main	a	b	03	06	07	11
GREF	{ }	{ }	{f3}	{ }	{v3}	{ }	{ g2 }

LAST TIME: SUMMARY FUNCTIONS

REVIEW: LAST LECTURE

GREF Computation

$$\text{GREF}_{\text{fn}}(F) = \text{IREF}_{\text{fn}}(F)$$

$$\text{GREF}_{\text{call}}(C) = \{ \}$$

worklist = { all calls and fns }

while(worklist not empty):

 e = pop worklist

 if element e is a call c in function s to function t:

$\text{GREF}_{\text{call}}(c) = \text{GREF}_{\text{fn}}(t)$ with formals mapped to actuals

 if $\text{GREF}_{\text{call}}(c)$ updated, add s to the worklist

 else if element e is a function f:

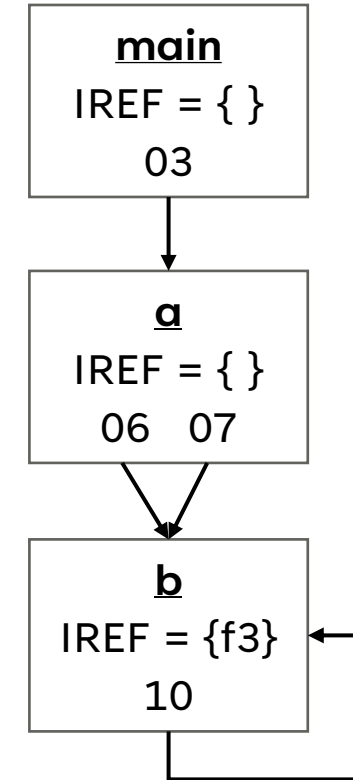
$\text{GREF}_{\text{fn}}(f) = \bigcup \text{GREF}_{\text{call}}(c)$ for all call sites c in f $\cup \text{IREF}(f)$

 if $\text{GREF}_{\text{fn}}(f)$ updated, add all call sites to f to worklist

```

01: int g1, g2, p3;
02: void main() {
03:   a(v1);
04: }
05: void a( int f1){
06:   b(v2, v3);
07:   b(v4, v5);
08: }
09: void b( int f2, int f3){
10:   print(f3);
11:   b(g1, g2);
12: }

```



Worklist: {		b	a	}				
	main	a	b	03	06	07	11	
GREF	{ }	{ }	{f3}	{ }	{v3}	~~{ }~~ {v5}	{ g2 }	

Process 07

$$\begin{aligned}
 \text{GREF}_{\text{call}}(07) &= \text{GREF}_{\text{fn}}(b) \text{ with } v4/f2 \text{ and } v5/f3 \\
 &= \{ \cancel{f3} \} v5 \\
 &= \{ v5 \}
 \end{aligned}$$

(update, add a to worklist, already there)

LAST TIME: SUMMARY FUNCTIONS

REVIEW: LAST LECTURE

GREF Computation

$GREF_{fn}(F) = IREF_{fn}(F)$

$GREF_{call}(C) = \{ \}$

worklist = { all calls and fns }

while(worklist not empty):

 e = pop worklist

 if element e is a call c in function s to function t:

$GREF_{call}(c) = GREF_{fn}(t)$ with formals mapped to actuals

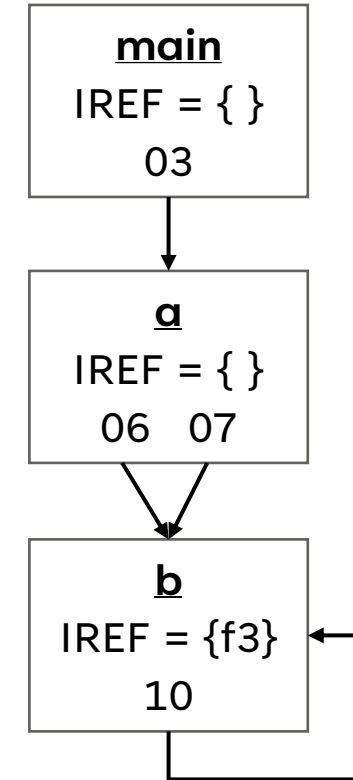
 if $GREF_{call}(c)$ updated, add s to the worklist

 else if element e is a function f:

$GREF_{fn}(f) = \bigcup GREF_{call}(c)$ for all call sites c in f $\cup IREF(f)$

 if $GREF_{fn}(f)$ updated, add all call sites to f to worklist

```
01: int g1,g2, p3;
02: void main() {
03:   a(v1);
04: }
05: void a( int f1){
06:   b(v2, v3);
07:   b(v4, v5);
08: }
09: void b( int f2, int f3){
10:   print(f3);
11:   b(g1, g2);
12: }
```



Worklist: { **b** a }

	main	a	b	03	06	07	11
GREF	{ }	{ }	{f3}	{ }	{v3}	{v5}	{ g2 }

LAST TIME: SUMMARY FUNCTIONS

REVIEW: LAST LECTURE

GREF Computation

$$\text{GREF}_{\text{fn}}(F) = \text{IREF}_{\text{fn}}(F)$$

$$\text{GREF}_{\text{call}}(C) = \{ \}$$

worklist = { all calls and fns }

while(worklist not empty):

 e = pop worklist

 if element e is a call c in function s to function t:

$\text{GREF}_{\text{call}}(c) = \text{GREF}_{\text{fn}}(t)$ with formals mapped to actuals

 if $\text{GREF}_{\text{call}}(c)$ updated, add s to the worklist

 else if element e is a function f:

$\text{GREF}_{\text{fn}}(f) = \bigcup \text{GREF}_{\text{call}}(c)$ for all call sites c in f $\cup \text{IREF}(f)$

 if $\text{GREF}_{\text{fn}}(f)$ updated, add all call sites to f to worklist

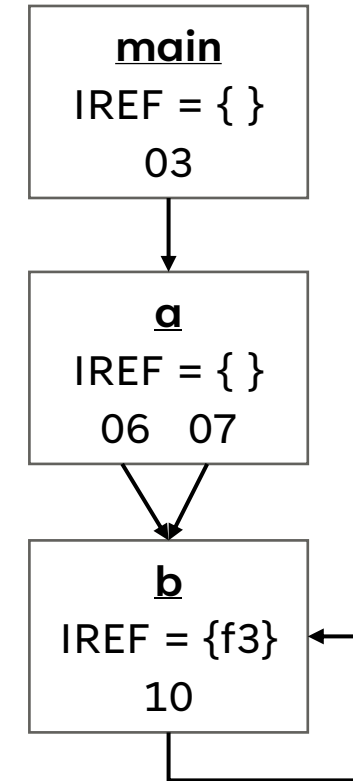
Worklist: { a 11 06 07 }

	main	a	b	03	06	07	11
GREF	{ }	{ }	~~{f3}~~	{ }	{v3}	{v5}	{g2}
			{f3,g2}				

```

01: int g1,g2, p3;
02: void main() {
03:   a(v1);
04: }
05: void a( int f1){
06:   b(v2, v3);
07:   b(v4, v5);
08: }
09: void b( int f2, int f3){
10:   print(f3);
11:   b(g1, g2);
12: }

```



Process b

$$\begin{aligned}
 \text{GREF}_{\text{fn}}(b) &= \text{GREF}_{\text{call}}(11) \cup \text{IREF}(b) \\
 &= \{g2\} \cup \{f3\} \\
 &= \{f3, g2\}
 \end{aligned}$$

Updated add 11, 06, 07

LAST TIME: SUMMARY FUNCTIONS

REVIEW: LAST LECTURE

GREF Computation

$GREF_{fn}(F) = IREF_{fn}(F)$

$GREF_{call}(C) = \{ \}$

worklist = { all calls and fns }

while(worklist not empty):

 e = pop worklist

 if element e is a call c in function s to function t:

$GREF_{call}(c) = GREF_{fn}(t)$ with formals mapped to actuals

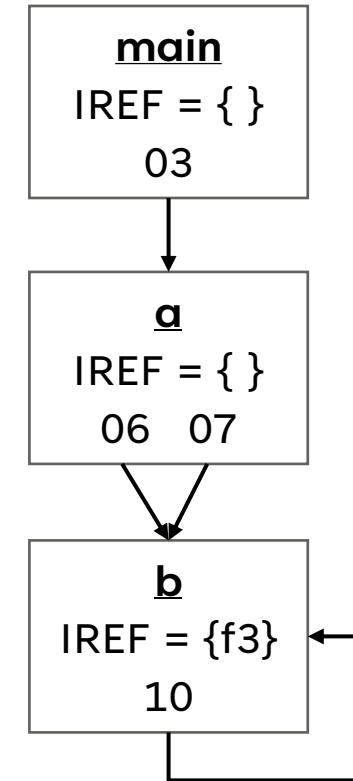
 if $GREF_{call}(c)$ updated, add s to the worklist

 else if element e is a function f:

$GREF_{fn}(f) = \bigcup GREF_{call}(c)$ for all call sites c in f $\cup IREF(f)$

 if $GREF_{fn}(f)$ updated, add all call sites to f to worklist

```
01: int g1,g2, p3;
02: void main() {
03:   a(v1);
04: }
05: void a( int f1){
06:   b(v2, v3);
07:   b(v4, v5);
08: }
09: void b( int f2, int f3){
10:   print(f3);
11:   b(g1, g2);
12: }
```



Worklist: { a 11 06 07 }

	main	a	b	03	06	07	11
GREF	{ }	{ }	{f3,g2}	{ }	{v3}	{v5}	{ g2 }

LAST TIME: SUMMARY FUNCTIONS

REVIEW: LAST LECTURE

GREF Computation

$$\text{GREF}_{\text{fn}}(F) = \text{IREF}_{\text{fn}}(F)$$

$$\text{GREF}_{\text{call}}(C) = \{ \}$$

worklist = { all calls and fns }

while(worklist not empty):

 e = pop worklist

 if element e is a call c in function s to function t:

$\text{GREF}_{\text{call}}(c) = \text{GREF}_{\text{fn}}(t)$ with formals mapped to actuals

 if $\text{GREF}_{\text{call}}(c)$ updated, add s to the worklist

 else if element e is a function f:

$\text{GREF}_{\text{fn}}(f) = \bigcup \text{GREF}_{\text{call}}(c)$ for all call sites c in f $\cup \text{IREF}(f)$

 if $\text{GREF}_{\text{fn}}(f)$ updated, add all call sites to f to worklist

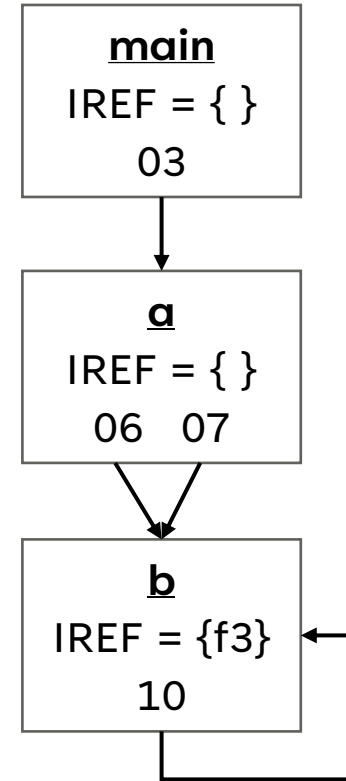
Worklist: { 11 06 07 03 }

 main a b 03 06 07 11

GREF { } ~~{ }~~ {f3,g2} { } {v3} {v5} { g2 }

 {v3,v5}

```
01: int g1,g2, p3;
02: void main() {
03:   a(v1);
04: }
05: void a( int f1){
06:   b(v2, v3);
07:   b(v4, v5);
08: }
09: void b( int f2, int f3){
10:   print(f3);
11:   b(g1, g2);
12: }
```



Process a

$$\begin{aligned} \text{GREF}_{\text{fn}}(a) &= \text{GREF}_{\text{call}}(06) \cup \text{GREF}_{\text{call}}(07) \cup \text{IREF}(a) \\ &= \{v3\} \cup \{v5\} \cup \{ \} \end{aligned}$$

Updated add 03

LAST TIME: SUMMARY FUNCTIONS

REVIEW: LAST LECTURE



LAST TIME: SUMMARY FUNCTIONS

REVIEW: LAST LECTURE

GREF Computation

$GREF_{fn}(F) = IREF_{fn}(F)$

$GREF_{call}(C) = \{ \}$

worklist = { all calls and fns }

while(worklist not empty):

 e = pop worklist

 if element e is a call c in function s to function t:

$GREF_{call}(c) = GREF_{fn}(t)$ with formals mapped to actuals

 if $GREF_{call}(c)$ updated, add s to the worklist

 else if element e is a function f:

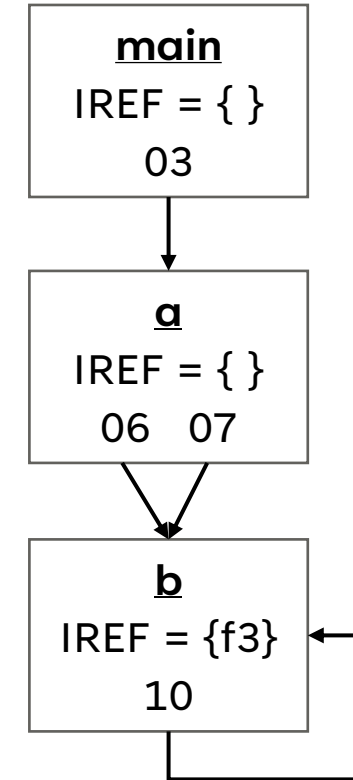
$GREF_{fn}(f) = \bigcup GREF_{call}(c)$ for all call sites c in f $\cup IREF(f)$

 if $GREF_{fn}(f)$ updated, add all call sites to f to worklist

```

01: int g1,g2, p3;
02: void main() {
03:   a(v1);
04: }
05: void a( int f1){
06:   b(v2, v3);
07:   b(v4, v5);
08: }
09: void b( int f2, int f3){
10:   print(f3);
11:   b(g1, g2);
12: }

```



Worklist: { }

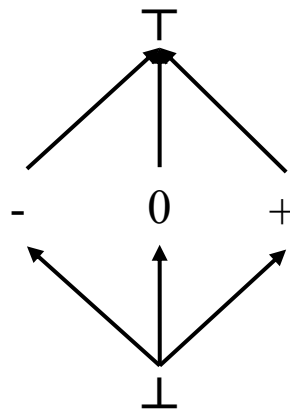
	main	a	b	03	06	07	11
GREF	{g2}	{ g2,v3,v5 }	{f3,g2}	{g2}	{v3,g2}	{v5, g2}	{ g2 }

ABSTRACT TRANSFER FUNCTIONS

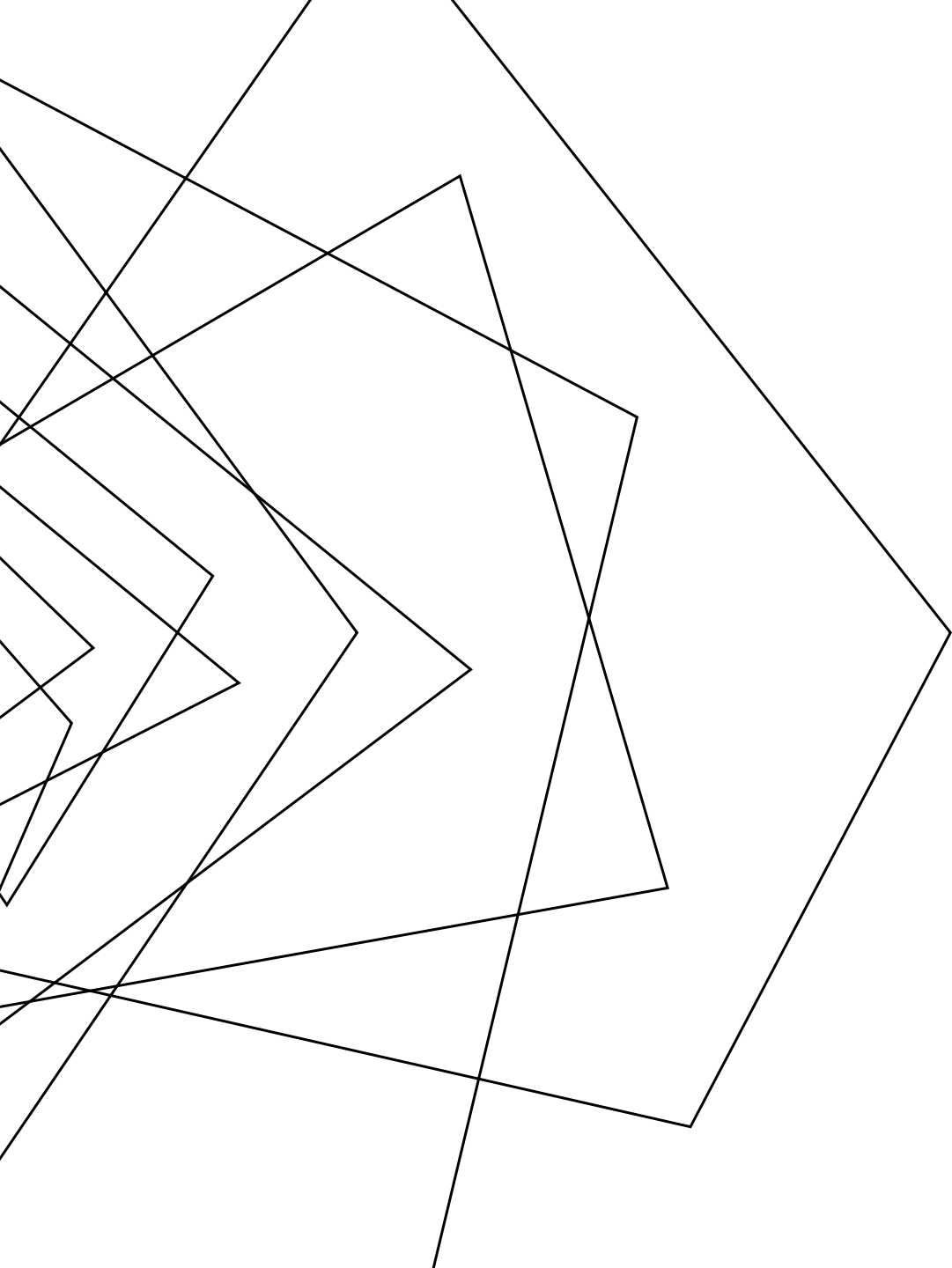
REVIEW: LAST LECTURE

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

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



foo(arg is $\top$) $\rightarrow$ return is $\top$
 foo(arg is +) $\rightarrow$ return is $\top$
 foo(arg is -) $\rightarrow$ return is $\top$
 foo(arg is 0) $\rightarrow$ return is +
 foo(arg is $\perp$) $\rightarrow$ return is $\top$



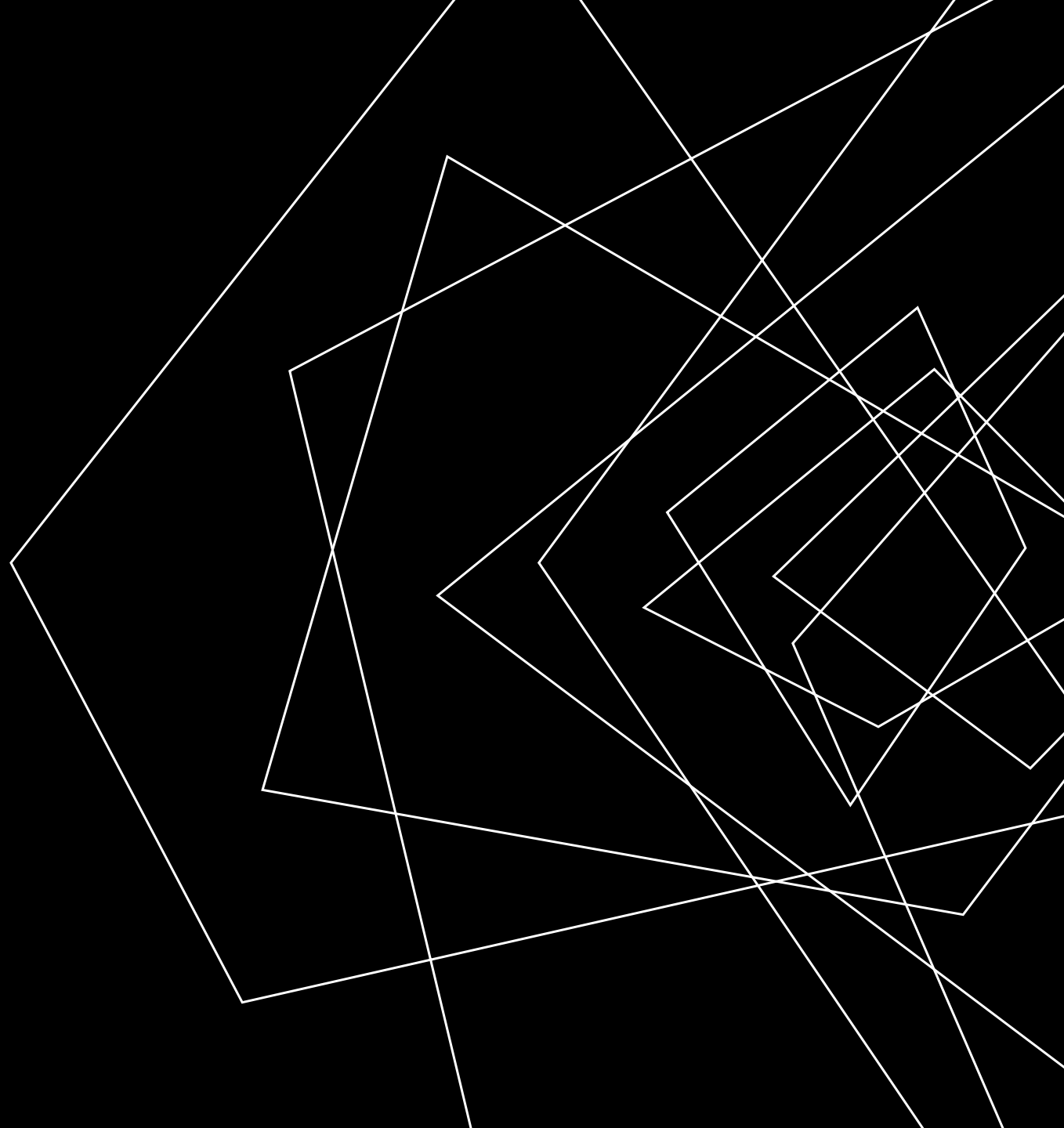
OVERVIEW

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



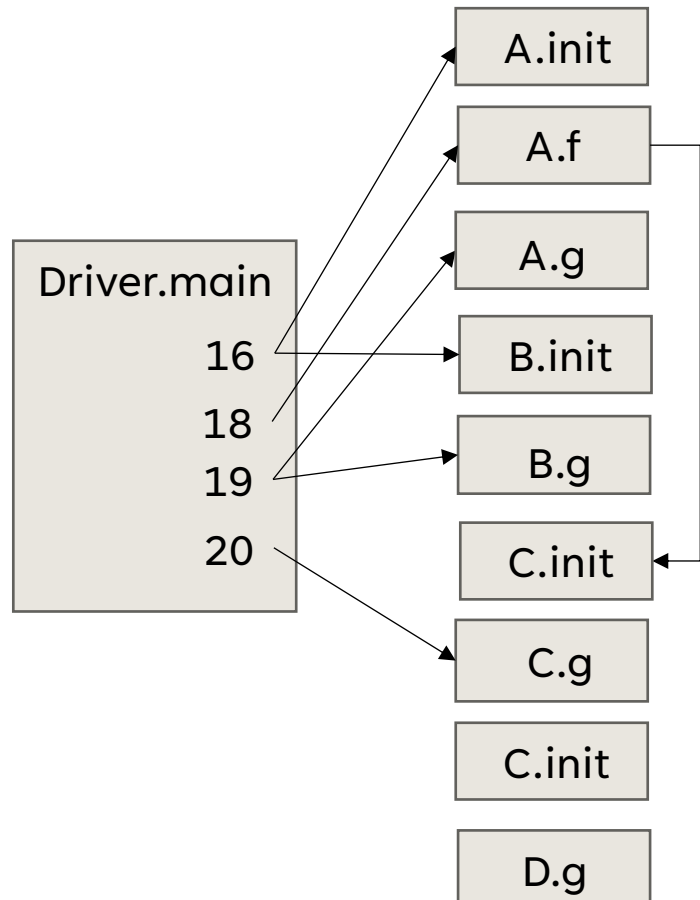
LECTURE OUTLINE

- 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(); ← 2, 2
19:             print(a.g()); ← 3, 6
20:             print(res.g()); ← 1, 9
21:         }
22:     }
23: };
  
```

DYNAMIC DISPATCH: GETS COMPLICATED!

A HISTORY OF COMPUTING



```
A a;  
if (rml){  
    a = new B();  
}  
else {  
    a = new C();  
}  
a.g();
```

DYNAMIC DISPATCH: GETS COMPLICATED!

A HISTORY OF COMPUTING

DIRECT CALLS

Not so bad

INDIRECT CALLS

Quite a bit harder: multiple targets possible!

```
class A {
    virtual void v() { ... }
    void x() { ... }
}
```

```
class B : public A {
    virtual void v() { ... }
    void x() { ... }
}
```



```
A* inst = new B();
inst->v() ← version declared in B
inst->x() ← version declared in A
```

DYNAMIC DISPATCH: GETS COMPLICATED!

A HISTORY OF COMPUTING

DIRECT CALLS

Not so bad

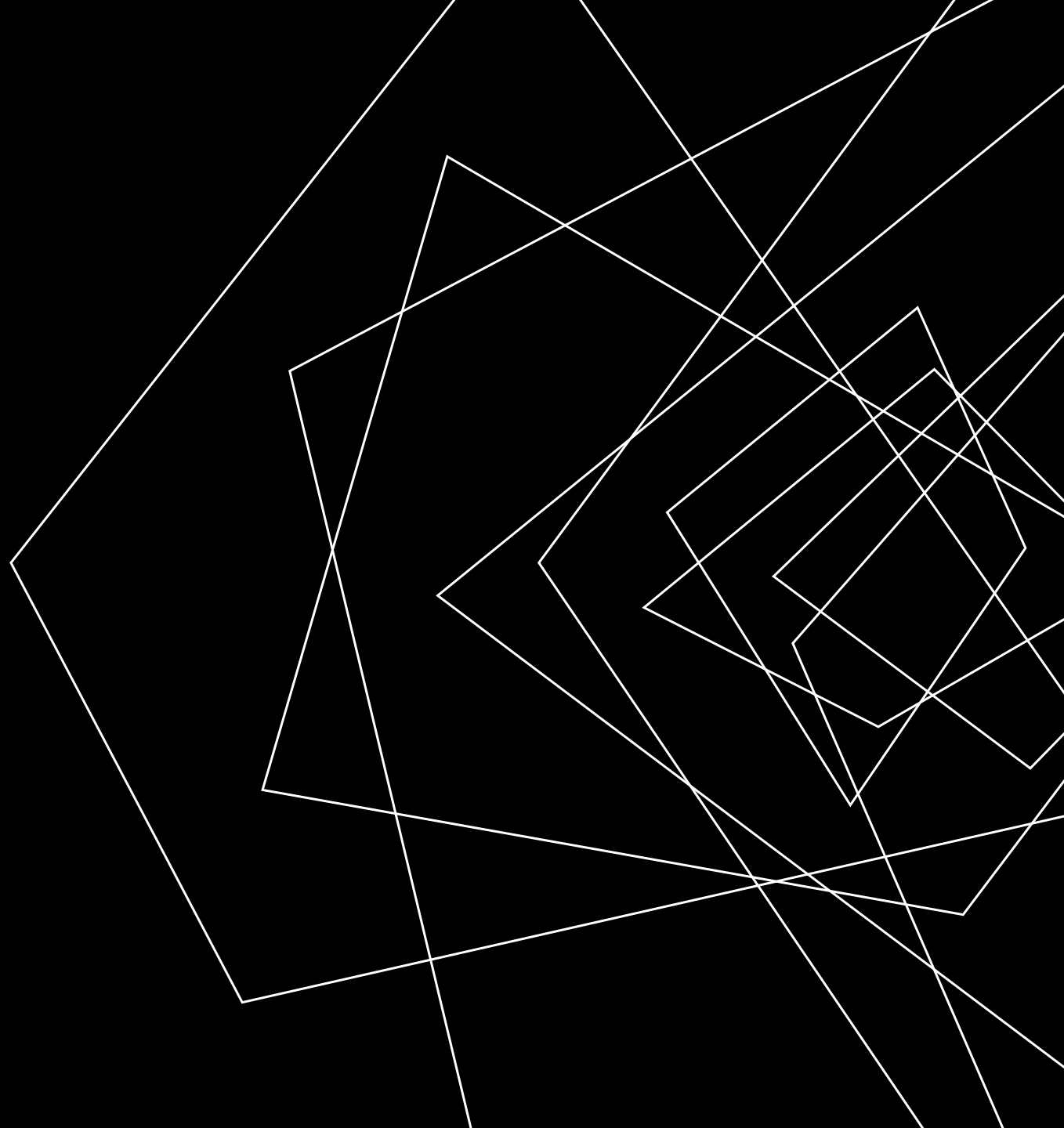
INDIRECT CALLS

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

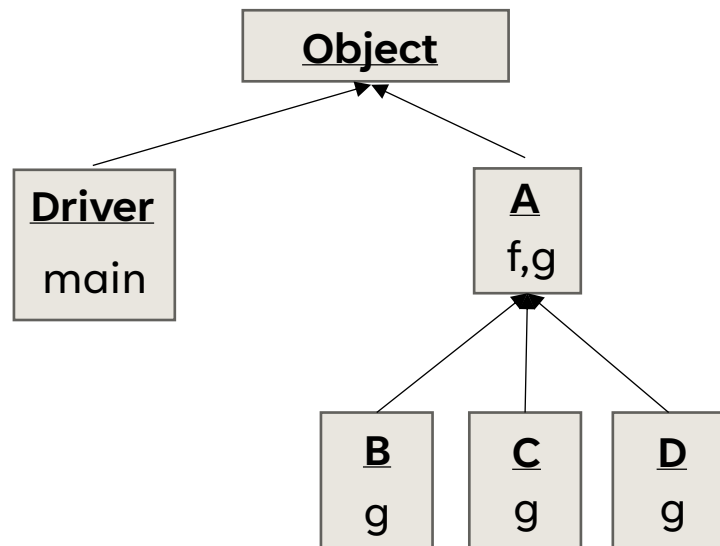


CLASS HIERARCHY ANALYSIS (CHA)

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: };
  
```

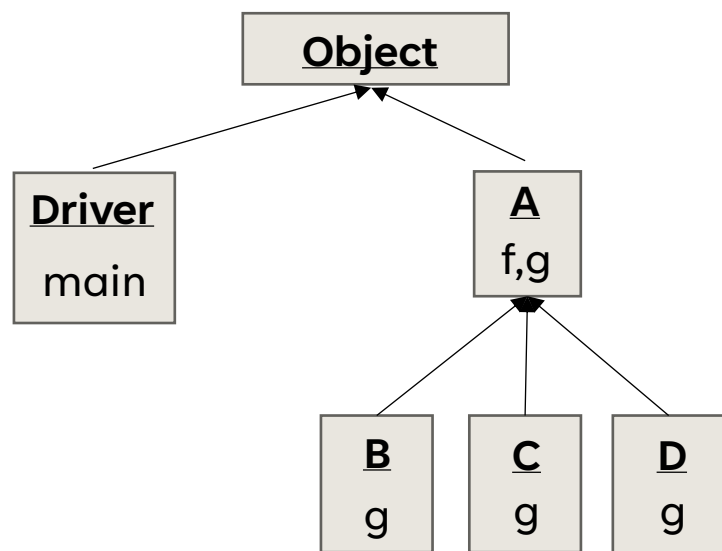
CHA

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: };

```

Orange arrows indicate reachability from the `main` method in the `Driver` class to the `f` and `g` methods of classes `A`, `B`, `C`, and `D`. A pink circle highlights the `new C()` instantiation in line 2 and the `print(res.g())` call in line 20, which are part of the reachable code.

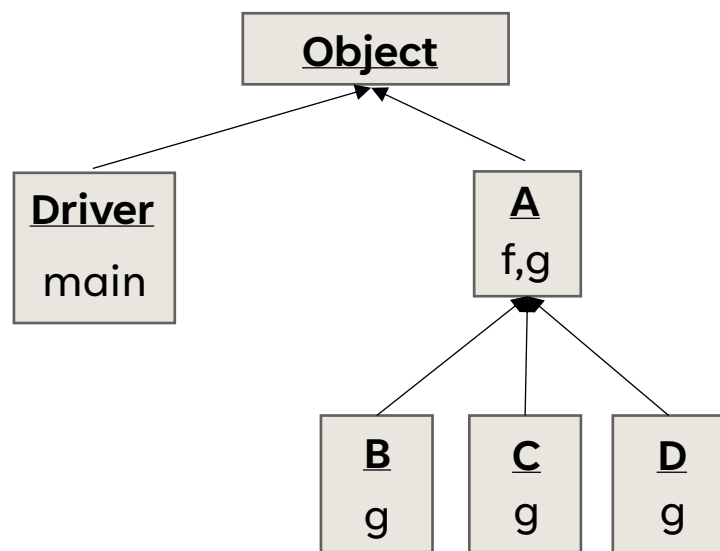
RAPID TYPE ANALYSIS (RTA)

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: };
  
```

The code snippet shows a Java-like class hierarchy. Class **A** has methods **f** (which returns a new **C** object) and **g** (which returns the string "A"). Class **B** inherits from **A** and overrides **g** to return "B". Class **C** inherits from **A** and overrides **g** to return "C". Class **D** inherits from **A** and overrides **g** to return "D". The **Driver** class has a **main** method that creates an array of **A** objects containing **A** and **B**, iterates over them, calls **f** on each, and then prints the result of **g** on both the original object and the result of **f**. Orange arrows highlight the method resolution paths: from **A.f()** to **C.g()**, from **A.g()** to **A.g()**, and from **B.g()** to **B.g()**. A pink circle highlights the **new C()** creation in line 2.

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

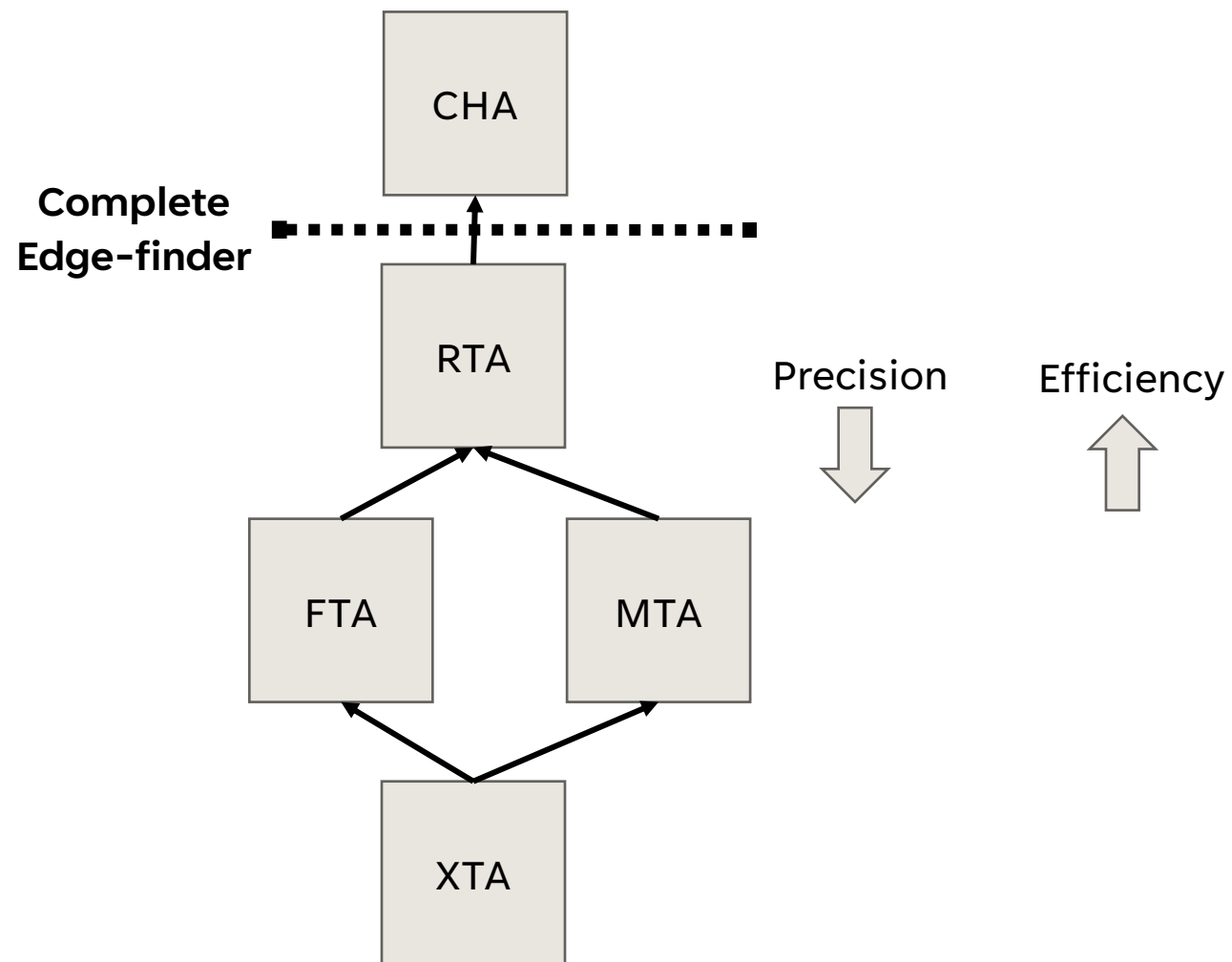
FTA adds the constraint that any method that reads from a field can inherit the field-compatible allocated types of any method that could write to that field.

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 and FTA

COMPARING CALL GRAPH ANALYSES

A HISTORY OF COMPUTING



WRAP-UP

