CS614: Advanced Compilers

Function Inlining and Devirtualization

Manas Thakur
CSE, IIT Bombay

Spring 2025

How do we call and return from functions?

- **In the caller:**
  - Save state of current procedure
    - Program counter (where to resume)
    - Registers (holding current computations)
  - Store arguments in a callee-accessible location
  - Transfer control-flow
- **In the callee:**
  - Collect parameters
  - Declare variables
  - Perform computations (perhaps in temporaries)
- **Return to caller:**
  - Store return value in a caller-accessible location

```
void foo() {
   …
   x = 20;
   r = bar(p, q);
   y = x + r;
   ...
}
int bar(int x, int y) {
   return x + y;
}
```

Function calls are quite expensive.

Every analysis/optimization we have seen till now is *intraprocedural*.

How do we optimize across function calls?

- Control Flow Graphs cannot be used to optimize across function boundaries.
- Without interprocedural constant propagation, the safest "correct" value of `x` and `y` (in `foo`) is bottom!
  - `x` must not be modified in `bar`.
  - `y`'s value needs to be determined by analyzing `bar`.
- Same story for all optimizations across function boundaries.

```
void foo() {
   ...
   x = 20;
   p = 10;
   q = 30;
   r = bar(p, q);
   y = x + r;
   // constant propagation?
}
int bar(int x, int y) {
   return x + y;
}
```

Function calls inhibit optimization.

Compilers try to inline functions!

Function Inlining

- **Idea:**
  - Replace a function call with the body of the callee
- **Benefits:**
  - Eliminate call/return overhead
  - Increase the scope of performing optimizations
    - More constant propagation
    - More partial evaluation
    - More everything else!
  - Hardware:
    - Eliminate two jumps
    - Keep the pipeline filled

How to inline a function?

- Attempt 1: Just copy-paste
  - With assignments from arguments to parameters
  - And of the return value

```
void foo() {
   ...
   x = 20;
   r = bar(p, q);
   y = x + r;
   ...
}
int bar(int x, int y) {
   return x + y;
}
```

```
void foo() {
   ...
   x = 20;
   x = p;
   y = q;
   r = x + y;
   y = x + r;
   ...
}
int bar(int x, int y) {
   return x + y;
}
```

- **Problem:** Final `y` was earlier `20+p+q`; now it is `2p+q`.

How to inline a function?

- Rename variables (uniquely):
  - With assignments from arguments to parameters
  - And of the return value

```
void foo() {
   ...
   x = 20;
   r = bar(p, q);
   y = x + r;
   ...
}
int bar(int x, int y) {
   return x + y;
}
```

```
void foo() {
   ...
   x = 20;
   bar_x = p;
   bar_y = q;
   r = bar_x + bar_y;
   y = x + r;
   ...
}
int bar(int x, int y) {
   return x + y;
}
```

- Final `y` is back to `20+p+q` :-)

Our inlining algorithm

A. Rename variables in the function being inlined

 - Follow use-def chains

B. Add assignments from arguments to parameters

C. Copy-paste code from callee to caller

D. Replace return statements with assignments to collecting variable

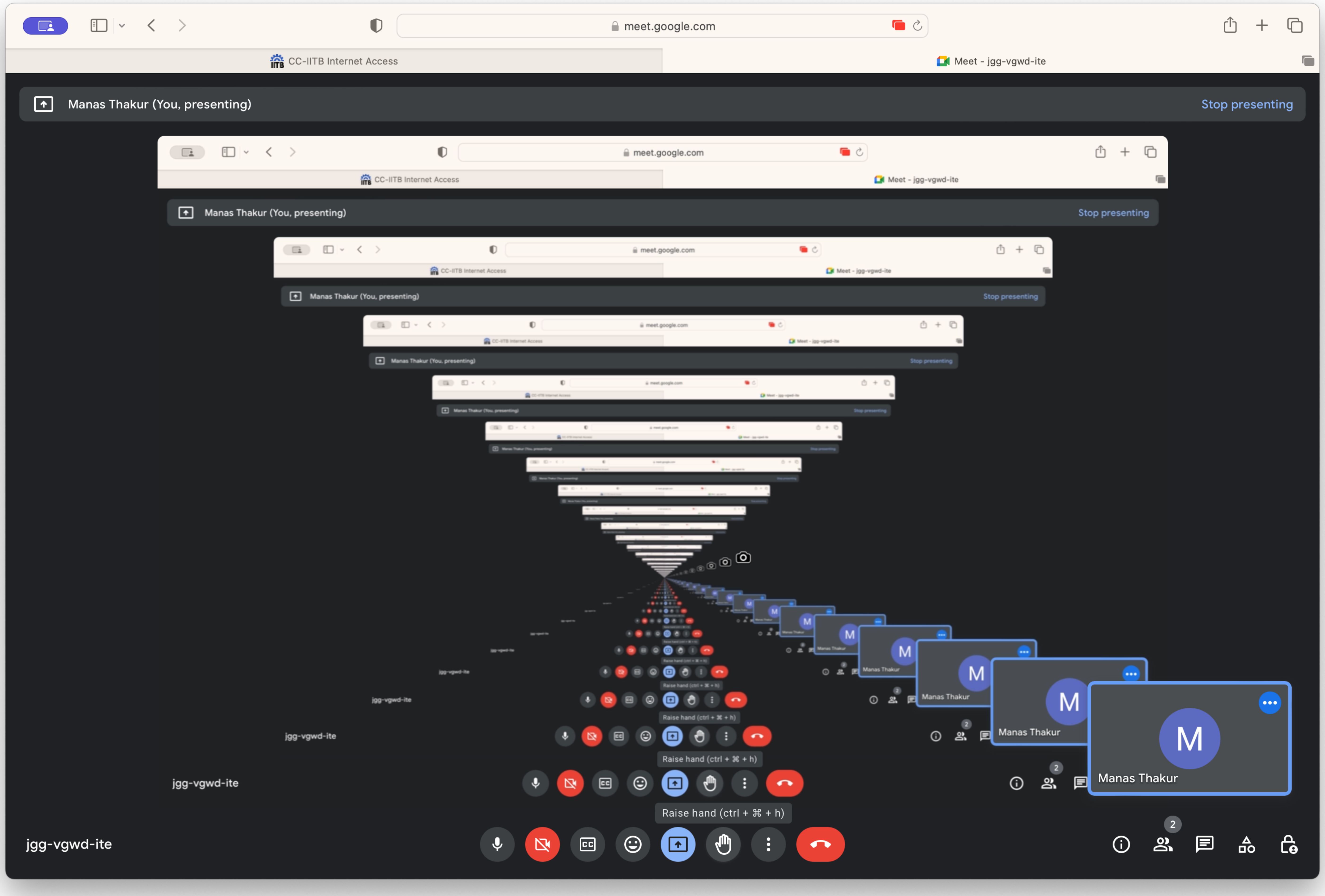
 - What if there are multiple return statements in the callee?

- Later, let the other optimization passes run over the modified CFG of the caller.

Can we always inline a function at its call site?

➤ Not if the calls are recursive

```
int foo(int z) {
   ...
   x = 20;
   r = bar(p, q);
   y = z + r;
   return y;
}
int bar(int x, int y) {
   if (x < y) {
      x += foo(x);
   }
   return x;
}
```



Can we always inline a function at its call site?

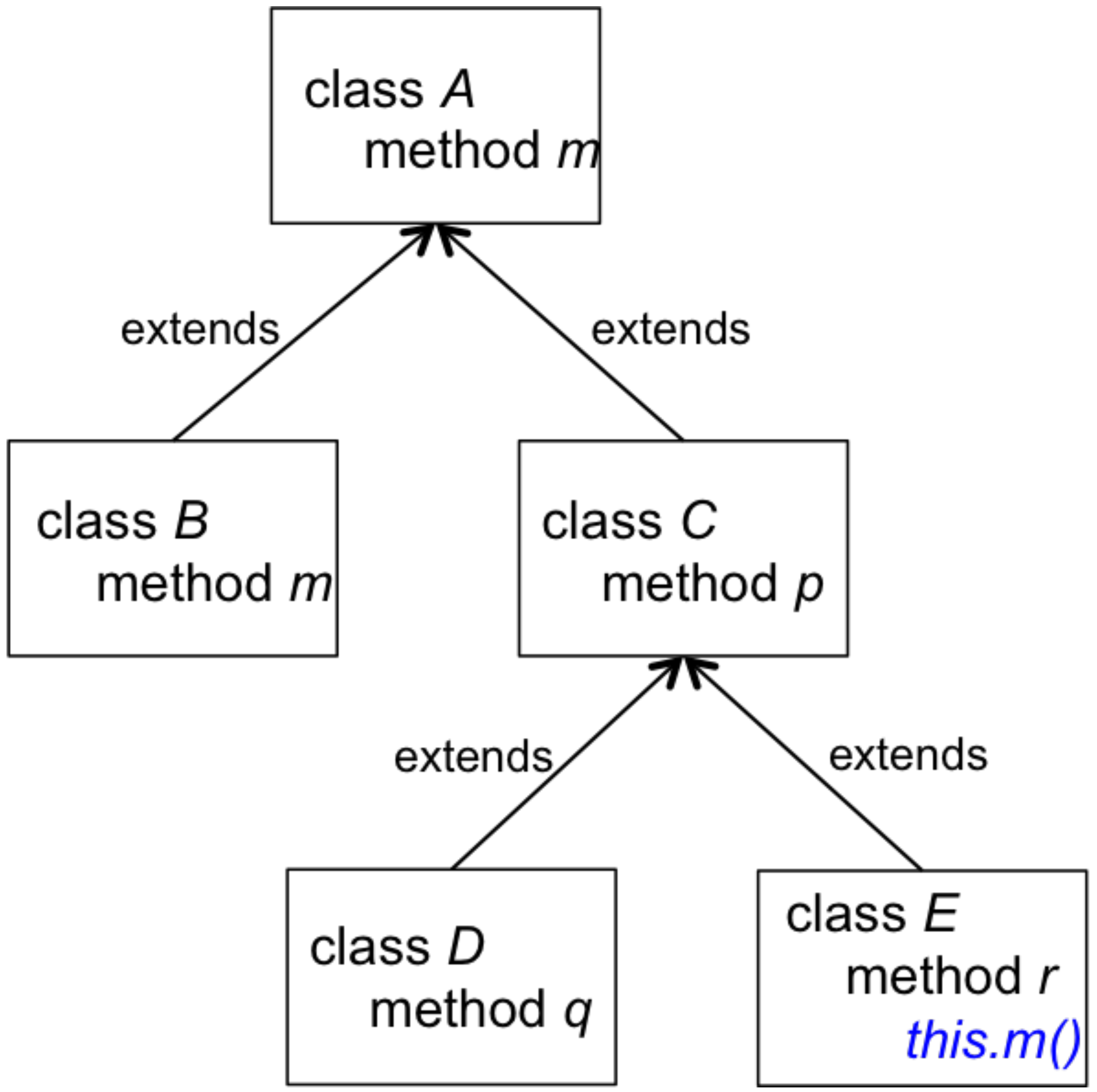
- Not if the call is virtual

```
void foo(T x) {
   U z;
c1:int y = x.bar();
   if (y > 40)
      z = new V();
   else
      z = new W();
c2:z.zap();
}
```

- Which method(s)can be called at `c1` and `c2`?
- If only one can be called (i.e. if we can devirtualize `c1`/`c2`), we can inline.

Devirtualization with Class Hierarchy Analysis

- Look at the class hierarchy of the type of the receiver to determine what methods can be called using the same.
- If a class re-defines a method `foo` defined in its parent class:
  - Only the redefined `foo` can be called using the objects of the extended class.
  - Methods not redefined are still accessible from the parent class.
- **CHA** helps in determining that only one implementation of `m` can be called from the blue call.



Devirtualization with Rapid Type Analysis

- Improves CHA with extra information:
  - Find if a class is ever instantiated in a program
  - If not, then remove it from the sets obtained using CHA
- Catch:
  - Assumption that the whole program is available
  - What about code that cannot be "seen"?
    - e.g., dynamically linked libraries
  - Even more interestingly done in the JVM (later).

```
class A {
   A foo(A x) {...}
}
class B extends A {
   A foo(A x) {...}
}
class D extends A {
   A foo(A x) {...}
}
... new D() ...
...
void bar(A y) {
   y.foo();
   // CHA: {A.foo,B.foo,D.foo}
   // RTA: {D.foo}
}
```

Devirtualization with Control-Flow Analysis

- Tries to find which classes of objects can flow to the receiver reference variables.
- Higher precision of devirtualization than CHA and RTA.

```
class A { A foo(A x) { return x; } }
class B extends A { A foo(A x) { return new D(); } }
class D extends A { A foo(A x) { return new A(); } }
class C extends A { A foo(A x) { return this; } }
```

```
void main() {
    A x = new A();
    while (...)
c1:     x = x.foo(new B());
    A y = new C();
c2: y.foo(x);
}
```

- **CHA/RTA:** Any of the `foo`s can be called at both call sites `c1` and `c2`.
- **Control-flow analysis:**
  - `c1`: {`A.foo`,`B.foo`,`D.foo`}
  - `c2`: {`C.foo`}

Should we always inline an inlinable function at all its call sites?

- Inlining a function increases code size.
  - This may also increase cache misses.
  - A phenomenon very similar to *loop unrolling*.

Instead of inlining, compilers may also replace "monomorphic" indirect calls with direct calls.

- Thus, most compilers are very careful while making inlining decisions:
  - Inline only if the callee is <K bytes.
  - Inline only if the caller does not grow beyond >K bytes.
  - Perform nested inlining only up to a threshold.
- Still, method inlining is among the largest sources of optimization in OO language compilers, and developing heuristics for the same is an interesting research problem.

Improving the Precision of Control-Flow Analysis

- The CFA we saw earlier does not distinguish the calls made to the current method from different call sites.
  - It is *context-insensitive*.
- We can improve precision using a context-sensitive analysis (*k*-CFA),
  - where *k* is the length of the current method's call chain.

```
class A {
  fb() { ... } }
class B extends A {
  fb() { ... } }
```

```
class C {
  void foo() {
    A a1, a2;
    a1 = new A(); //l1
c1: bar(a1);
    a2 = new B(); //l2
c2: bar(a2);
  }
  void bar(A p1) {
    p1.fb();
} }
```

- Context insensitive:
  - `p1 -> {l1, l2}`
- 1-CFA:
  - `At c1: p1 -> {l1}`
  - `At c2: p1 -> {l2}`

k-CFA (Cont.)

```
class C {
  void main() {
    foo();
    ...
    foo();
  }

  void foo() {
    bar();
  }

  void bar() {
    fb();
  }
}
```

- **1-CFA:**
  - 1 context for `fb`
- **2-CFA:**
  - 2 contexts for `fb`
- **3-CFA?**

The Opposite of *Polymorphism*

- **Monomorphic** call:
  - Only one method can be called
  - Target can either be bound statically, and sometimes inlined
- Which calls in Java are always monomorphic?
  - Calls to static methods
  - Calls to final methods
  - Calls to methods of final classes
- We increase the size of the above set using the different analyses seen in the previous slides.
- How could we compare the precision of two CG construction algorithms empirically?
  - Count the identified number of monomorphic call-sites.

What we have seen today

- **CHA:** only look at inheritance relations
- **RTA:** also look at code
- ***0*-CFA:** also analyze code
- ***k*-CFA:** contextualize the analysis

- Some points to note:
  - *k*-CFA is a context-sensitive analysis
  - The length *k* is the length of the call-stack
  - Here the last *k* call-sites form our context

What we haven’t seen today

- There are multiple ways we can define what is a context
  - Called the **context abstraction** of a given context-sensitive analysis
  - Object-sensitive contexts

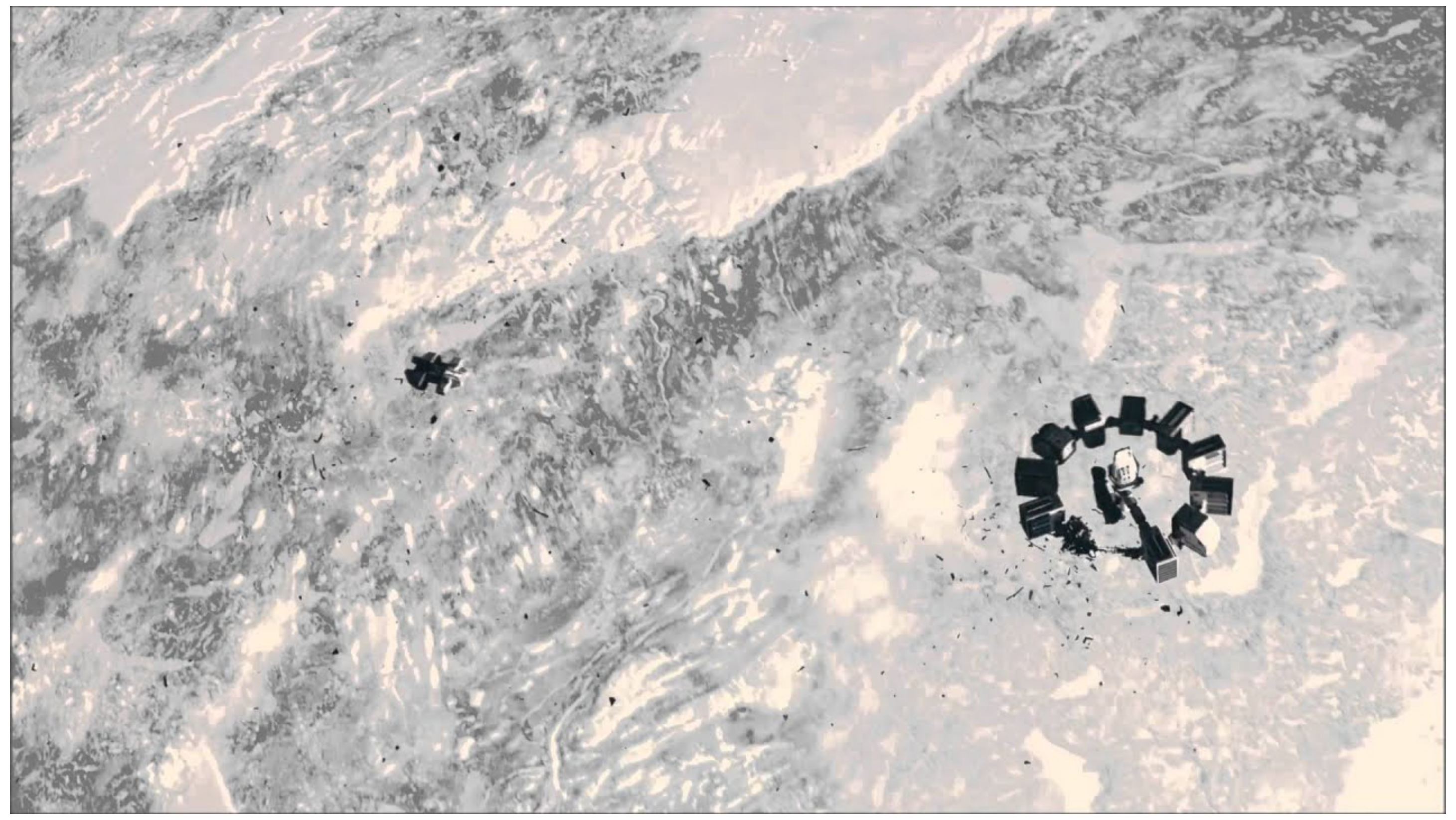
    Ana Milanova, Atanas Rountev, Barbara G. Ryder. TOSEM ‘05.
  - Value contexts

    Uday P. Khedker and Bageshri Karkare. CC’08.
  - Type-sensitive contexts

    Yannnis Smaragdakis, Martin Bravenboer, Ondřej Lhoták. POPL ‘11.
  - LSRV contexts

    Manas Thakur and V. Krishna Nandivada. CC’19.
  - Many more non-context-sensitive algorithms:

    Frank Tip and Jens Palsberg. OOPSLA ’00.



No time for these yet!