CS614: Advanced Compilers

Alias Analysis (Cont.)

Manas Thakur
CSE, IIT Bombay

Spring 2025

Handling method calls

- Points-to relations before the call to `foo`:
  - `a -> {O1}; O1.f -> {O2}; b -> {O3}`
- Can `foo` change `ptsto(a)`?
  - No (call by value).
- Can it change `ptsto(O1.f)`?
  - Yes (the value passed is a reference).
- How about `ptsto(b)`?
  - No.

Note. Sometimes we simply use arrows or `ptsto` instead of `Stack` and `Heap`.

```
a = new A();     //O1
a.f = new B();   //O2
b = new A();     //O3
foo(a,b);
...
static void foo(A p, A q) {
   ...
}
```

This will become crystal clear on the next slide.

- Conservatively, we need to assume everything reachable from O_1 and O_2 may change (that is, may point to all the objects **possible**).

Possibility differs across C/C++ and Java.

Interprocedural points-to updates

- Points-to relations before the call to `foo`:
  - a –> {O_1}; O_1.f –> {O_2}
  - b –> {O_3}; O_3.f –> null
- Points-to relations at the end of `foo`:
  - p –> {O_7}; O_7.f –> {O_8}
  - q –> {O_3}; O_3.f –> {O_9}
- Points-to relations after the call to `foo`:
  - a –> {O_1}; O_1.f –> {O_2}
  - b –> {O_3}; O_3.f –> {O_9}
- **Much** more precise than handling method calls conservatively.

```
a = new A();        //O1
a.f = new B();      //O2
b = new A();        //O3
foo(a,b);
...
static void foo(A p, A q) {
   p = new A();     //O7
   p.f = new B();   //O8
   q.f = new B();   //O9
}
```

Alias information

- Two access paths may alias iff their may-point-to sets intersect.

```
ptsto(a) = {O1}
ptsto(b) = {O3}
ptsto(r) = {O1,O3}
ptsto(s) = {O1,O3}
```

```
ptsto(O1.f) = {O2}
ptsto(O3.f) = {O5,O8}
ptsto(O5.g) = {O12}
ptsto(O8.g) = {O12}
```

```
alias(x,y) == true iff

ptsto(x) ∩ ptsto(y) != Φ
```

```
1.  a = new A();      //O1
2.  a.f = new B();    //O2
3.  b = new A();      //O3
4.  if (*) {
5.     b.f = new B();//O5
6.     r = a;
7.  } else {
8.     b.f = new B();//O8
9.     r = b;
10. }
11. s = r;
12. b.f.g = new A();//O12
```

- alias(a,b)?
- alias(a,r)?
- alias(a.f,b.f)?

The notion of “reachability”

- An object O_x is said to be reachable from a variable v if there is an access path starting at v that may lead to O_x.

```
ptsto(a) = {O1}
ptsto(b) = {O3}
ptsto(r) = {O1,O3}
ptsto(s) = {O1,O3}
```

```
ptsto(O1.f) = {O2}
ptsto(O3.f) = {O5,O8}
ptsto(O5.g) = {O12}
ptsto(O8.g) = {O12}
```

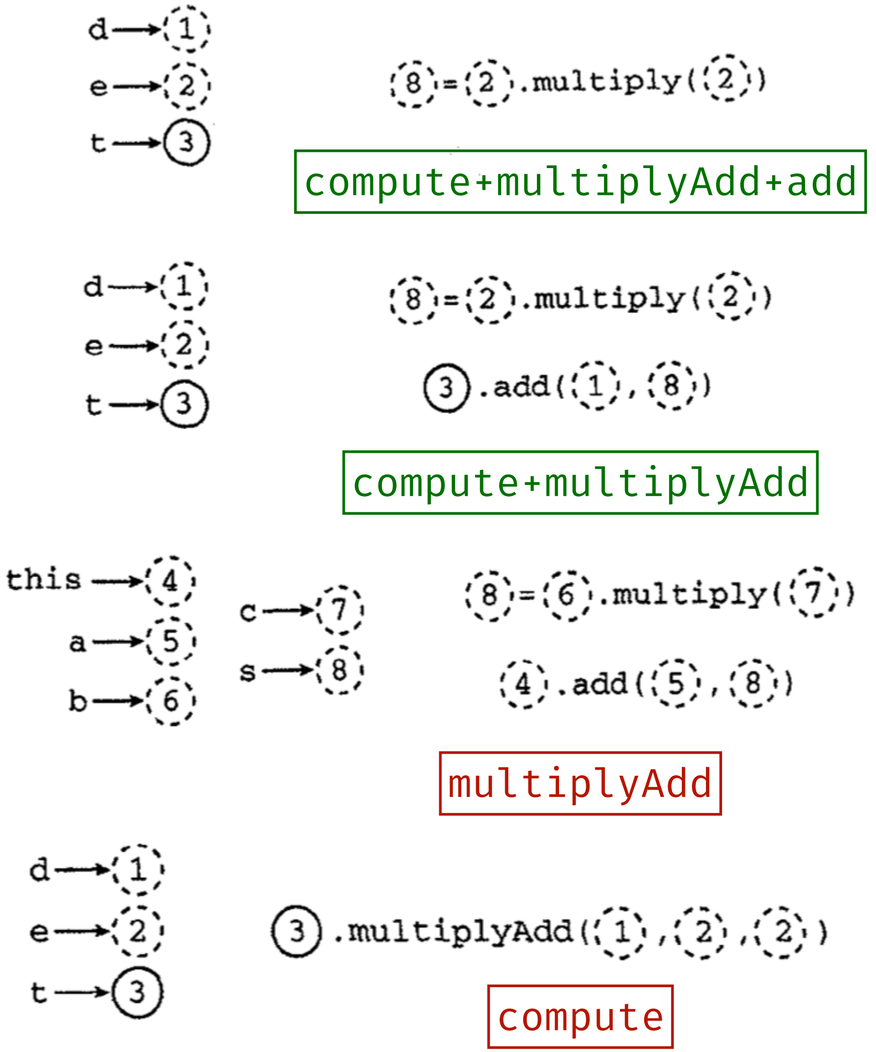
```
1. a = new A();      //O1
2. a.f = new B();    //O2
3. b = new A();      //O3
4. if (*) {
5.     b.f = new B();//O5
6.     r = a;
7. } else {
8.     b.f = new B();//O8
9.     r = b;
10.}
11. s = r;
12. b.f.g = new A();//O12
```

- The objects reachable from a are O_1 and O_2.
- The objects reachable from b are O_3, O_5, O_8, and O_{12}.
- Useful for parallelization (and many other applications):
  - If an object is reachable from a global variable (static field in Java) then it may be accessed by multiple threads.
  - Reachability becomes even clearer if we draw a **Points-To Graph** (on board).

Incremental Analysis

```
class Complex {
    double x, y;
    Complex(double a, double b) {
        x = a; y = b;
    }
    void add(Complex u, Complex v) {
        x = u.x + v.x;
        y = u.y + v.y;
    }
    Complex multiply(Complex m) {
        Complex r = new Complex(x*m.x - y*m.y, x*m.y + y*m.x);
        return r;
    }
    void multiplyAdd(Complex a, Complex b, Complex c) {
        Complex s = b.multiply(c);
        this.add(a, s);
    }
}
```

```
class Client {
    public static void compute(Complex d, Complex e) {
    3:  Complex t = new Complex(0.0, 0.0);
        t.multiplyAdd(d, e, e);
    }
}
```



Discussion on Incremental Compilation

What else can pointer information be used for?

- Escape analysis
- Garbage collection
- Null-check elision
- Loop transformations
- Virtual call resolution
- …

Last topic: How are things that we have learnt different in JIT compilers?

- Almost every compiler optimization depends **heavily** on points-to information.
- Precise pointer analysis does not scale very well over large programs.
- Every PL conference has new papers every year on pointer analysis.

Assignment 4

PolyGone: Eliminating Polymorphism with PTA