CS614: Advanced Compilers

Alias Analysis

Manas Thakur
CSE, IIT Bombay

Spring 2025

Remember this example from parallelization?

- Dependence analysis (while performing loop transformations, instruction scheduling, parallelization, and many others) requires determining whether two variables (e.g. `x` and `y`) could **alias**.
- Two variables (or in general, *access paths*) may alias with each other if they *may point to* the same memory location.

```
foo() {
    x = 10;
}
bar() {
    print *y;
}
```

- **Three key phrases here:**
  - access paths
  - may information
  - points-to relationship

Memory allocation in Java

- The statement `A a = new A();`:
  - allocates an object of class `A` on the heap;
  - calls its constructor; and
  - stores the reference (address) of the object into the reference variable `a`.
- We have studied the size and the layout of objects already.

> There is no way to allocate objects on stack (in Java *source/byte code*).

- Variable `a` is stored on the stack frame of the containing method.
- What if `a` is a class field?
  - It is part of an object of the enclosing class, and is stored on the heap.

1. Access paths for memory locations

- `a` points to the `new A` object
- `a.f` points to the `new B` object
- `a.f.g` points to the `new C` object
- `c` points to the `new C` object too
- `d` points to the `new A` object too
- `e` points to the `new B` object too
- `f` points to the `new A` object
  - Aiyo, which one?!
- We need some naming scheme for objects.

```
a = new A();
a.f = new B();
a.f.g = new C();
c = a.f.g;
d = a;
e = a.f;
f = new A();
```

Allocation-site abstraction

- An object-allocation site may instantiate many objects at runtime. How?
  - Loops.
  - Method invocations.
- Say all the objects allocated at line `l` are called O_l.
  - Called the allocation-site abstraction.
  - O_l is an *abstract object.*
- Now we have a finite number of objects in each program,
- *and a name for each object too!*
- If line numbers are not unique, qualify the object with method/class/package/file, etc.

```
1. a = new A();      //O1
2. a.f = new B();    //O2
3. a.f.g = new C();  //O3
4. c = a.f.g;
5. d = a;
6. e = a.f;
7. f = new A();      //O7
```

2. May versus must information

- Which variables *may* get assigned (some value) in this program?
  - `a, b, c`
- Which variables *must* get assigned in this program?
  - `a`

```
if (d) {
   a = ...
   b = ...
} else {
   a = ...
   c = ...
}
```

- May analysis: the computed information should hold in at least one execution of the program.
- Must analysis: the computed information should hold in all the executions of the program.

We usually compute **may-point-to** relationships.

3. Points-to relationships

- A reference variable on the stack may point to one or more object(s) in its lifetime.
  - Let's store such points-to relationships in a map *Stack*.
- Similarly, each reference field of a heap object may point to one or more object(s).
  - Let's store such points-to relationships in a two-level map *Heap*.
- Note that the points-to values are **sets**.
- Thus:

```
Stack[a] = {O1}
Stack[c] = {O3}
Stack[d] = {O1}
Stack[e] = {O2}
Stack[f] = {O7}
```

```
Heap[O1,f] = {O2}
Heap[O2,g] = {O3}
```

PCQ: `Heap[O7,f]?`

```
1. a = new A();     //O1
2. a.f = new B();   //O2
3. a.f.g = new C(); //O3
4. c = a.f.g;
5. d = a;
6. e = a.f;
7. f = new A();     //O7
```

- In fact, points-to updates after processing each statement can be stored in a database and retrieved using SQL/Datalog queries!

Intraprocedural updates

```
L: v = new T(); // Alloc
   Stack[v] = {OL}
```

```
v = w; // Copy
   Stack[v] = Stack[w]
```

These can also be done using a "points-to graph" —
a topic learnt in detail in the COOOL course, but we would see today as well.

```
v = w.f; // Field load
   Stack[v] = {}
   forall Ow in Stack[w]:
      Stack[v] U= Heap[Ow,f]
```

```
v.f = w; // Field store
   forall Ov in Stack[v]:
      Heap[Ov,f] = Stack[w]
```

Practice

- What should be `Stack[s]`?
    - $\{O_1, O_3\}$
- For which all objects X does $O_{12} \in$ `Heap[X,g]`?
    - O_5 and O_8
- The merge operation is union.

```
a = new A();      //O1
a.f = new B();    //O2
b = new A();      //O3
if (*) {
    b.f = new B();//O5
    r = a;
} else {
    b.f = new B();//O8
    r = b;
}
s = r;
b.f.g = new A();//O12
```

Flow (in)sensitivity

- Flow-sensitive results:
  - `Stack[a] = {O1}` from lines 1-3, `{O2}` afterwards
  - `Stack[b] = {O2}` from lines 2-4; `{O3}` afterwards
  - `Stack[c] = {O3}` from lines 3-5; `{O2}` afterwards

```
1. a = new A(); //O1
2. b = new A(); //O2
3. c = new A(); //O3
4. a = b;
5. b = c;
6. c = a;
```

- Single "summary" at the end:

```
Stack[a] = {O1,O2}
Stack[b] = {O2,O3}
Stack[c] = {O3,O2}
```

Flow-insensitivity is *very* fast, but loses precision.

- Flow-insensitive results:

```
Stack[a] = {O1,O2,O3}
Stack[b] = {O1,O2,O3}
Stack[c] = {O1,O2,O3}
```