CS614: Advanced Compilers

Alias Analysis

Manas Thakur
CSE, IIT Bombay

Autumn 2023

Remember this example?

- 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[O_7,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, these can be stored in a database and retrieved using SQL/Datalog queries!

Intraprocedural updates

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

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

These can also be done using a "points-to graph", a topic we would see more during presentations.

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

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

```
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
```

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}
```

Handling method calls

- Points-to relations before the call to `foo`:
  - `a -> {`O_1`}; `O_1`.f -> {`O_2`}; b -> {`O_3`}`
- 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.

This will become crystal clear on the next slide.

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) {
   ...
}
```

- 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 -> {O1}; O1.f -> {O2}`
  - `b -> {O3}; O3.f -> null`
- Points-to relations at the end of `foo`:
  - `p -> {O7}; O7.f -> {O8}`
  - `q -> {O3}; O3.f -> {O9}`
- Points-to relations after the call to `foo`:
  - `a -> {O1}; O1.f -> {O2}`
  - `b -> {O3}; O3.f -> {O9}`

```
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
}
```

- **Much** more precise than handling method calls conservatively.

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}
```

```
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(x,y) == true iff
ptsto(x) ∩ ptsto(y) != Φ
```

- `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.

```
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
```

```
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}
```

- 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).

What else can this 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.