Applied Static Analysis

An Introduction to Points-to and Alias Analysis

Software Technology Group
Department of Computer Science
Technische Universität Darmstadt
Dr. Michael Eichberg

> If you find any issues, please directly report them: GitHub

Some of the images on the following slides are inspired by slides created by Eric Bodden.

Points-to analysis vs. alias analysis

- Points-to analysis computes for each variable the allocation sites whose objects the variable *may*/*must* point to: $\text{points-to}(v) = \{a1, a2, \ldots\}$
- Alias analysis determines which variables *may* or *must* alias, i.e., point to the same objects:
  - $\text{may-alias}(v1, v2) = true/false$
  - $\text{must-alias}(v1, v2) = true/false$

In case of a *may* analysis $true$ means **maybe**. I.e., if two variables may alias then they may point to the same object, but they don't have to. If the answer is $false$, they definitively never alias.
In case of a *must* analysis $false$ (only) means *maybe not*.

May vs. must alias analysis

```
a = new A();
if(..) {
    b = a;
}
c = new C();
d = c;
```

$\text{may-alias}(a, b) = true$

$\text{must-alias}(a, b) = false$

$\text{may-alias}(a, c) = false$

$\text{must-alias}(c, d) = true$

May vs. must alias analysis

```
a = new A();
if(..) {
    b = a;
}
c = new C();
d = c;
d = a;  // <= NEW
```

$\text{may-alias}(a, b) = true$

$\text{must-alias}(a, b) = false$

$\text{may-alias}(a, c) = false$

$\text{must-alias}(c, d) = false$

Over the lifetime of an entire execution, two variables (practically) never always alias. Thus must-alias analysis typically needs to take control flow into account (they have to be flow-sensitive).

Flow-sensitive must analysis

```
      b = null;
      d = null;
s1:   a = new A();
      if(..) {
          b = a;
      }
s2:   c = new C();
      b = c;
s3:   d = a;
```

$\text{must-alias}(variable\ v1, variable\ v2; \text{after execution of } s_x) \rightarrow \{true, false\}$

$\text{must-alias}(a, d; s2) = false$

$\text{must-alias}(a, d; s3) = true$

$\text{must-alias}(b, c; s2) = false$

$\text{must-alias}(b, c; s3) = true$

Flow-insensitive must analysis

In a flow-insensitive analysis the order in which the instructions will be evaluated is ignored.

```
     b = null;
     d = null;
s1:  a = new A();
     if(..) {
         b = a;
     }
s2:  c = new C();
     b = c;
s3:  d = a;
```

$$\text{must-alias}(variable\ v1, variable\ v2) \rightarrow \{true, false\}$$

$$\text{must-alias}(a, d) = false$$

$$\text{must-alias}(a, d) = false$$

$$\text{must-alias}(b, c) = false$$

$$\text{must-alias}(b, c) = false$$

Here, we always have to chose the save default answer: $false$. However, this observation is generally true: most program properties don't always hold and **therefore most must-analyses have to be flow sensitive**.

The above observation does not hold for may-analyses. They only determine whether a property may (if at all) hold somewhere in the program.

Points-to and alias analysis

Points-to analysis can answer alias-analysis queries:

$$\text{alias}(v1, v2) = (\text{points-to}(v1) \bigcap \text{points-to}(v2) \neq \emptyset)$$

This leads us to the question: *Is points-to analysis always more expressive than alias analysis*?

Points-to vs. alias analysis

- Points-to analysis requires a notion of allocation sites: $\text{points-to}(v) = \{a1, a2, \dots\}$
- Alias analysis only talks about variables: $\text{may-alias}(v1, v2) = true/false$ or $\text{must-alias}(v1, v2) = true/false$

Important in real world: What if we have *incomplete knowledge about allocation sites*?

Points-to and alias analysis for incomplete programs

```
void readProp(String id, String default) {
    String s = Properties.read(id);
    if(s==null) s = default;
    return s;
}
```

Assume that `Properties.read` is a native method or that for some other reason we don't know its definition.

A may alias analysis will be able to derive: $\textbf{may-alias}(s, default) = true$.
We cannot compute the information by using $\textbf{points-to}$ information:
$\textbf{may-alias}(s, default) = (\textbf{points-to}(s) \bigcap \textbf{points-to}(default) \neq \emptyset)$.

The underlying issue is that $\textbf{points-to(s)}$ cannot be computed. Choosing either the empty set ($\emptyset$) or `any object` will both render the analysis practically unusable. Using the empty set is (potentially grossly) unsound and using `any object` is (grossly) imprecise.

Problem of points-to analysis for incomplete programs

Summary

- Points-to analysis associates variables with allocation sites.
- If allocation sites are unknown then this association is necessarily either unsound or imprecise, depending on the analysis design.
- In comparison, alias analysis can recover precision by analyzing the relationship between variables without caring which objects they point to.

A direct alias analysis

(We are not using points-to information!)

```
b = a
c = b
```

$\textbf{may-alias}(b, a) = \mathit{true}$

$\textbf{may-alias}(c, b) = \mathit{true}$

What happens if `a = null`?

In this case (`a = null`) the variables do not alias. Hence, returning `true` is imprecise (but still sound).

When to prefer points-to analyses?

```
l = new LinkedList(); // Allocation site a1
l.clear();
```

In the above case, points-to information ($\textbf{points-to}(l) = \{a1\}$) can be used to devirtualize the (`clear`) method call; it can only invoke `LinkedList.clear()` . Alias information ($\textbf{type-of}(\textbf{points-to}(l)) = \{LinkedList\}$) is (in general) of no use in this case.

Weak Updates

So-called weak updates are generally required if *only may-alias information is available*. **For aliases, information before the statement is retained, only new information is added**.
We cannot kill information; otherwise we would be unsound!

Weak Updates - example

We only know:

- `x.f ↦ 0`, `y.f ↦ 0` (the fields `f` are initialized to `0`)
- $\textbf{may-alias}(x, y)$

We see an update:

```
x.f = 3
```

Given that we only have may-alias information, we must retain the old value for field f of alias y:

- `x.f ↦ 3`, `y.f ↦ 3`, `y.f ↦ 0`
- $\textbf{may-alias}(x, y)$

A weak update is necessary, because $\textbf{may-alias}(x, y)$ tells us that there is a path along which x and y do alias and there may be a path along which they don't. But the value `y.f` must represent both possible truths at the same time; hence both `y.f↦3`, `y.f↦0` must be included!

Strong Updates

So-called strong updates require must-alias information and can *kill* analysis information associated with an alias.

Strong Updates - example

We (only) know:

- `x.f ↦ 0`, `y.f ↦ 0` (the fields `f` are initialized to `0`)
- $\textbf{must-alias}(x, y)$

We see an update:

```
x.f = 3
```

Given that we have must-alias information, we can safely kill `y.f↦0`.

- `x.f ↦ 3`, `y.f ↦ 3`
- $\textbf{must-alias}(x, y)$

Generally, we can never kill information on aliases without must-alias information.