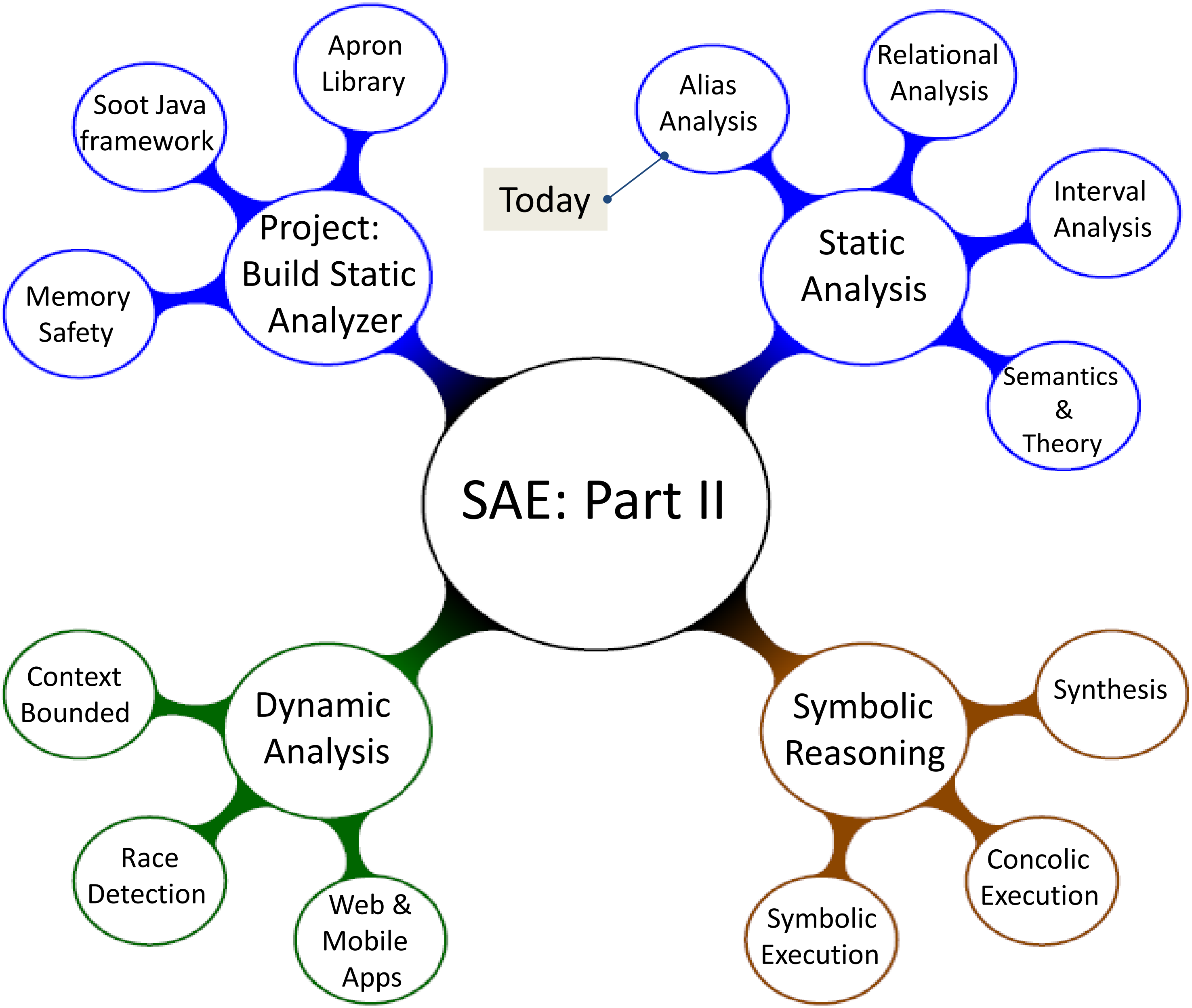
Software Architecture and Engineering: Part II

ETH Zurich, Spring 2014
Prof. Martin Vechev



SAE: Part II

- Project: Build Static Analyzer
  - Soot Java framework
  - Apron Library
  - Memory Safety
- Static Analysis
  - Alias Analysis
  - Relational Analysis
  - Interval Analysis
  - Semantics & Theory
- Dynamic Analysis
  - Context Bounded
  - Race Detection
  - Web & Mobile Apps
- Symbolic Reasoning
  - Synthesis
  - Concolic Execution
  - Symbolic Execution

Today

Pointer & Alias Analysis

Pointer and Alias Analysis is fundamental to reasoning about heap manipulating programs (pretty much all programs today). Virtually all practical static analysis tools (bug finding, verification, etc...) contain some form of pointer analysis.

Due to its importance, the topic has received much attention from the research and developer communities. In our lecture today, we will study the core concepts of such pointer analyses and illustrate them on examples. This will enable us to use (like in the course project) or to build/extend such analyzers.

Updated Language

```
x      ∈    Var          set of integer variables
p,q    ∈    PtrVar       set of variables pointing to objects
f      ∈    Field        set of field names

a ::= as defined in earlier lectures (arithmetic expressions)

b ::= p = q  | … as before          (boolean expressions)

s ::=
        p := newObject$^{\ell}$ T   create a new object of a given type
     |  p := q$^{\ell}$             pointer assignment
     |  p.f := a$^{\ell}$           integer heap store
     |  x := p.f$^{\ell}$           integer heap load
     |  p.f := q$^{\ell}$           pointer heap store
     |  p := q.f$^{\ell}$           pointer heap load
```

Let us define the concrete store

A little more elaborate than before:

- Objs : set of all possible objects
- PtrVal = Objs $\cup$ { null }

This is as before

- $\rho \in \text{PrimEnv} = \text{Var} \longrightarrow \text{Z}$
- $r \in \text{PtrEnv} = \text{PtrVar} \longrightarrow \text{PtrVal}$
- $h \in \text{Heap} = \text{Objs} \longrightarrow (\text{Field} \longrightarrow \{\text{PtrVal} \cup \text{Z}\})$

A store is now: $\sigma = \langle \rho, r, h \rangle \in \text{Store} = \text{PrimEnv} \times \text{PtrEnv} \times \text{Heap}$

Some Common Terms

- Aliases
  - Two pointers p and q are aliases if they point to the same object

- Points-to pair
  - (p, A) means p holds the address of object A

- Points-to pairs and aliases
  - if (p, A) and (r, A) then p and r are aliases

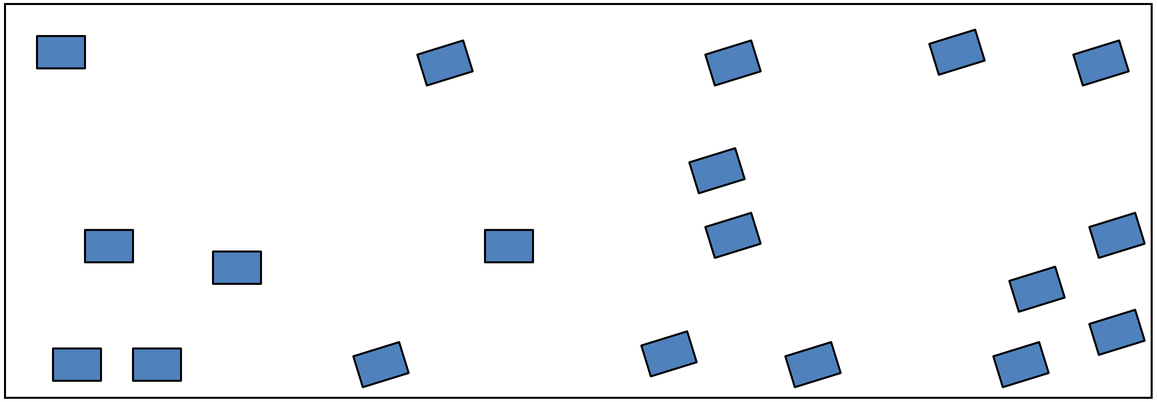
(May) Points-to Analysis

What to do with `newObject` ? A program can create an unbounded number of objects.

We need to again use abstraction. That is, we need some static naming scheme for dynamically allocated objects

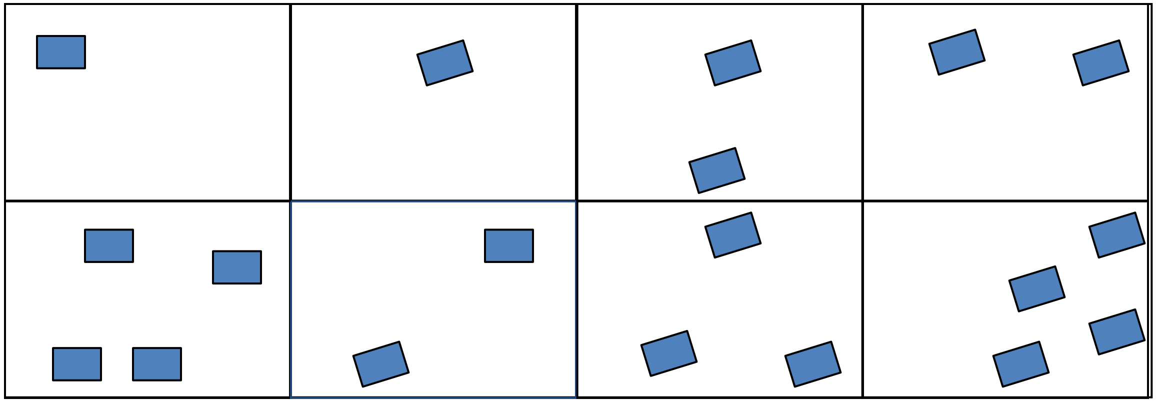
Abstraction: Allocation Sites

- Divide heap into a fixed partition based on allocation site (the statement label)
- All objects allocated at the same program point (label) get represented by a single "abstract object"



Abstraction: Allocation Sites

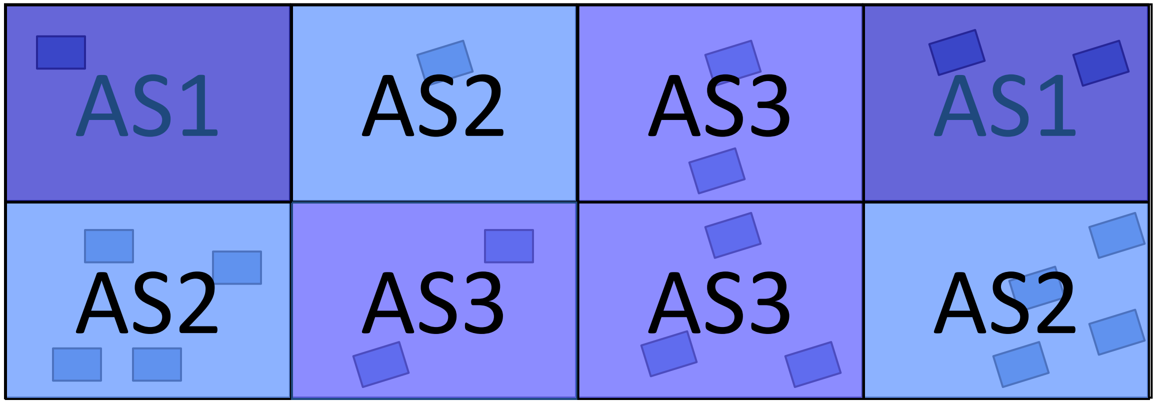
- Divide heap into a fixed partition based on allocation site (the statement label)
- All objects allocated at the same program point (label) get represented by a single "abstract object"



Abstraction: Allocation Sites

- Divide heap into a fixed partition based on allocation site (the statement label)

- All objects allocated at the same program point (label) get represented by a single “abstract object”



Abstract Objects

The (static) abstract objects can just be the allocations sites (labels of statements in our simple language) of the program. If this is too imprecise, we can also use the calling context. This is for instance common in library frameworks where the allocation site inside the library is not useful as we need to know where the library was called from. Naturally, bigger calling context will lead to more abstract objects.

If we use allocation sites (labels), we can now define the abstract objects as follows:

$$\texttt{AbsObj = }\{\ell \mid \texttt{stmt}(\ell)\ \texttt{is p := newObject}^{\ell}\ \texttt{T}\}$$

Pointer Analysis: two kinds

- Flow sensitive: respects the program control flow
  - a separate set of points-to pairs for every program point
  - the set at a program point represents possible may-aliases on some path from entry to the program point
- Flow insensitive: assume all execution orders are possible, abstracts away order between statements
  - good for concurrency (if not too imprecise)

Pointer Analysis: two kinds

Let us first take a look at the flow sensitive analysis and to define its abstract domain, discuss the abstraction and its abstract transformers.

Step 1: Define Domain

The abstract domain is a complete lattice:

$$\texttt{Labs} \rightarrow (\ (\texttt{PtrVar} \rightarrow \wp(\texttt{AbsObj})) \times \\ (\texttt{AbsObj} \times \texttt{Field} \rightarrow \wp(\texttt{AbsObj}))\ )$$

That is, the abstract domain keeps two maps at every program label. The first map contains a mapping from a pointer variable to a set of abstract objects. The second map contains a mapping from the fields of abstract objects to the set of abstract objects they point to.

Note that this lattice is of finite height. We have a finite number of abstract objects (i.e. `AbsObj`), finite number of field names (i.e. `Field`), and a finite number of pointer variables (i.e. `PtrVar`), and labels (i.e. `Lab`). Therefore, we will not need widening here.

Step 1: Define Domain

The abstract domain is a complete lattice:

$$\texttt{Labs} \rightarrow (\ (\texttt{PtrVar} \rightarrow \wp(\texttt{AbsObj})) \times (\texttt{AbsObj} \times \texttt{Field} \rightarrow \wp(\texttt{AbsObj}))\ )$$

Example of an element in the domain:

$1 \rightarrow (\ p \rightarrow \{a_5, a_{10}\},\ a_5.f \rightarrow \{a_6, a_9\}\ )$

…

$43 \rightarrow$ …

We read this as follows: at program label 1, pointer p points to 2 abstract objects a_5 and a_{10} . Field f of abstract object a_5 points to two abstract objects a_6 and a_9. In this element, we have other program labels (43 of them), where there are many such pointer maps, but we did not write them explicitly here.

Step 1: Define Domain

The abstract domain is a complete lattice:

$$\texttt{Labs} \rightarrow (\ (\texttt{PtrVar} \rightarrow \wp(\texttt{AbsObj})) \times (\texttt{AbsObj} \times \texttt{Field} \rightarrow \wp(\texttt{AbsObj}))\ )$$

What are $\sqsubseteq, \sqcup, \sqcap, \perp, \mathrm{T}$?

Example:

$$1 \rightarrow (\, p \rightarrow \{a_5, a_{10}\}, a_5.f \rightarrow \{a_6, a_9\}\,)$$
$$\sqsubseteq$$
$$1 \rightarrow (\, p \rightarrow \{a_5, a_{10}, a_{15}\}, a_5.f \rightarrow \{a_6, a_9, a_{52}\}\,)$$

Essentially, everything is based on $\subseteq$, $\cup$, $\cap$, lifted appropriately.
It is a good exercise to define them formally.

Step 2: Define Abstraction

$$\alpha\text{:}\ \wp(\Sigma) \to (\texttt{Labs} \to (\ (\texttt{PtrVar} \to \wp(\texttt{AbsObj})) \times (\texttt{AbsObj} \times \texttt{Field} \to \wp(\texttt{AbsObj}))\ ))$$

$$\gamma\text{:}\ (\texttt{Labs} \to (\ (\texttt{PtrVar} \to \wp(\texttt{AbsObj})) \times (\texttt{AbsObj} \times \texttt{Field} \to \wp(\texttt{AbsObj}))\ )) \to \wp(\Sigma)$$

Using α, we abstract a set of states into the two kinds of maps. Similarly, using γ, we concretize the pointer maps to a set of states.

The formal definition of α and γ is left as an exercise.

Let us consider an example to give an intuition.

Example of Abstraction

$$\begin{array}{l}
\alpha\ (\\
\{\ \langle\ 5,\ \_\ ,\ \{p\mapsto o_1, q\mapsto o_2\}\ ,\ \{o_1.k\mapsto o_3,\ o_2.v\mapsto o_6\}\ \rangle, \\
\quad \langle\ 5,\ \_\ ,\ \{p\mapsto o_2, q\mapsto o_3\}\ ,\ \{o_1.k\mapsto o_3,\ o_2.v\mapsto o_3\}\ \rangle \\
\}\ )
\end{array}$$

Here, by _ we mean that the program has no integer variables.

Suppose that: object o_1 is allocated at site a_3 (program label 3)
object o_2 is allocated at site a_4 (program label 4)
object o_3 is allocated at site a_9 (program label 9)
object o_6 is allocated at site a_{31} (program label 31)

What is the result ?

Example of Abstraction

α (
$\{ \langle\ 5,\ \_\ ,\ \{p \mapsto o_1, q \mapsto o_2\}\ ,\ \{o_1.k \mapsto o_3,\ o_2.v \mapsto o_6\}\ \rangle,$
$\langle\ 5,\ \_\ ,\ \{p \mapsto o_2, q \mapsto o_3\}\ ,\ \{o_1.k \mapsto o_3,\ o_2.v \mapsto o_3\}\ \rangle$
$\}$)

Here, by _ we mean that the program has no integer variables.

Suppose that: object o_1 is allocated at site a_3 (program label 3)
object o_2 is allocated at site a_4 (program label 4)
object o_3 is allocated at site a_9 (program label 9)
object o_6 is allocated at site a_{31} (program label 31)

$$5 \rightarrow (\{p \mapsto \{a_3, a_4\},\ q \mapsto \{a_4, a_9\}\},\ \{a_3.k \mapsto \{a_9\},\ a_4.v \mapsto \{a_{31}, a_9\}\})$$

Step 3: Define Abstract Transformers

We now need to define the effect of program statements manipulating pointers on the abstract domain. That is, creation of objects, pointer assignment and conditionals:

`p = q`	`compare two pointers`
`p := newObject`$^{\ell}$ `T`	`create new object`
`p := q`$^{\ell}$	`assign pointers`
`p.f := q`$^{\ell}$	`pointer heap store`
`p := q.f`$^{\ell}$	`pointer heap load`

Lets us take a look at the most tricky one (pointer heap store). The rest are just direct assignments. The formal definitions are left as an exercise.

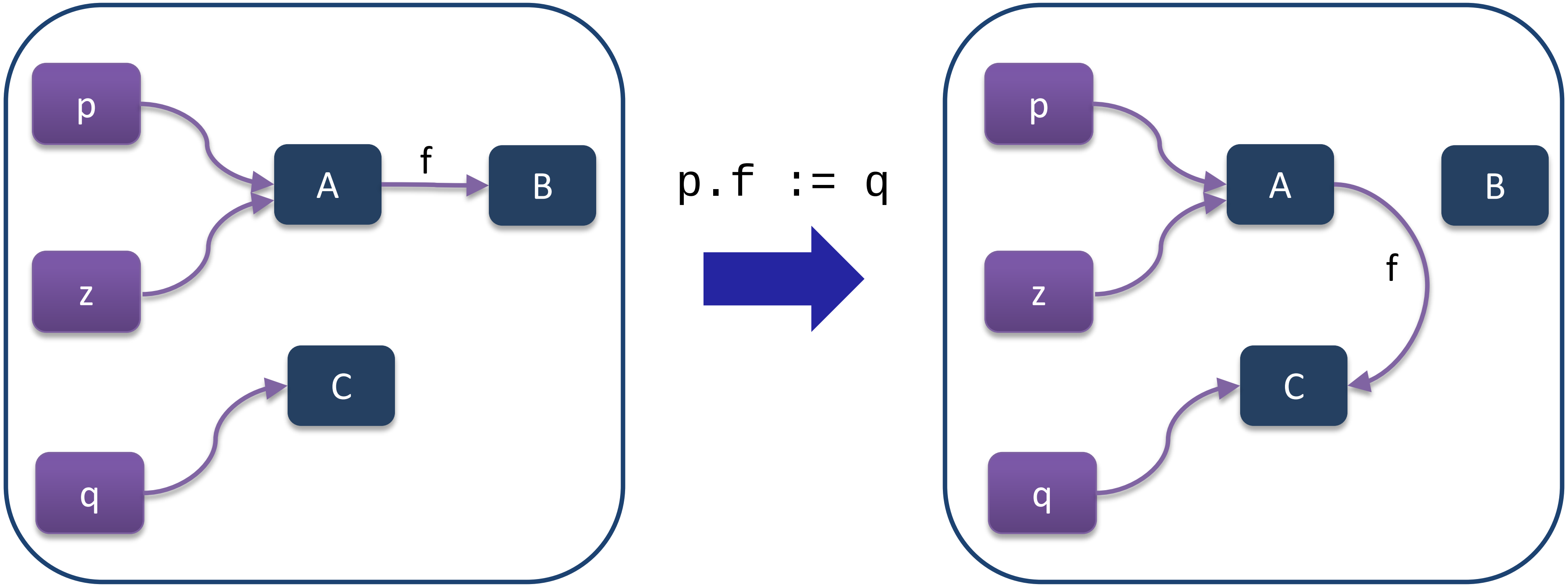
What about `p.f := q` ?

Say **p** ↦ **{A}, where A.f** ↦ **{B}, and q** ↦ **{C}**. Can we have **A.f** ↦ **{C}** as a result?

Abstract Element AE1

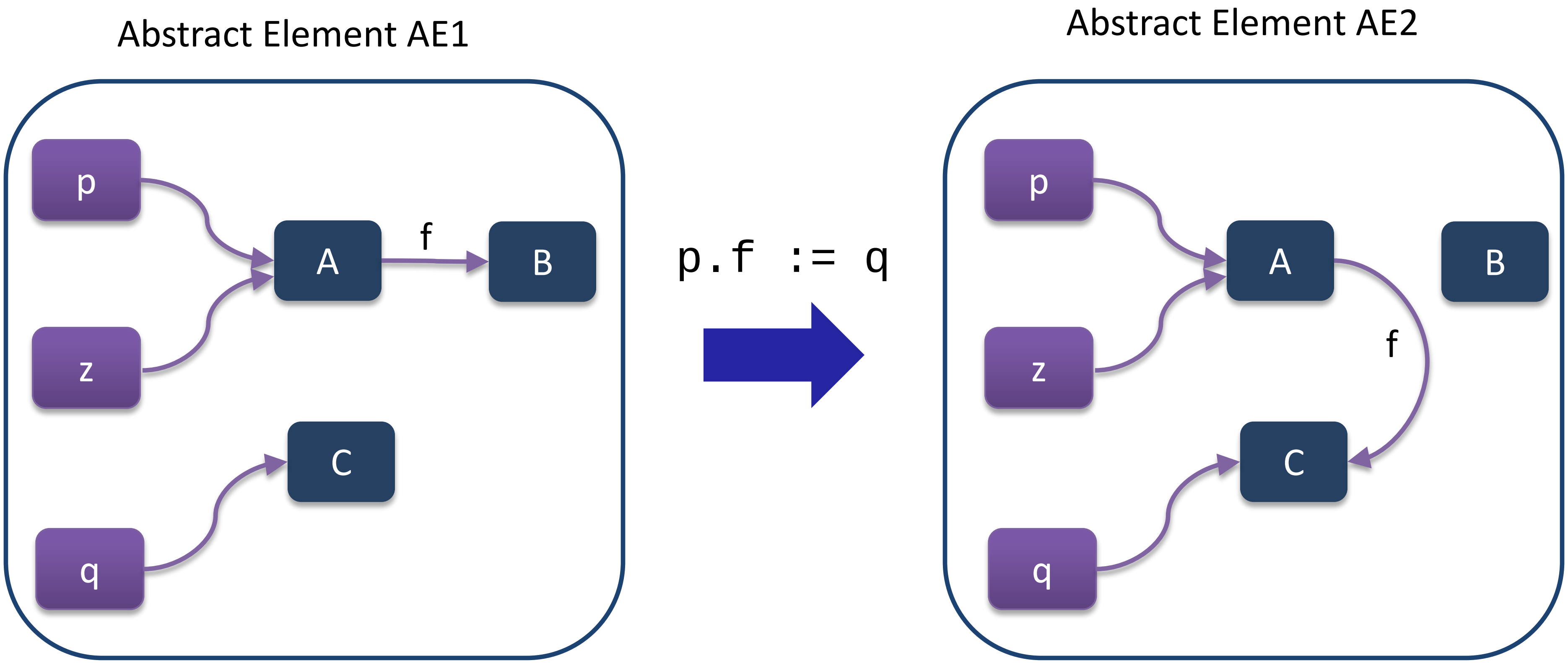
Abstract Element AE2

Is this result correct ?



What about `p.f := q` ?

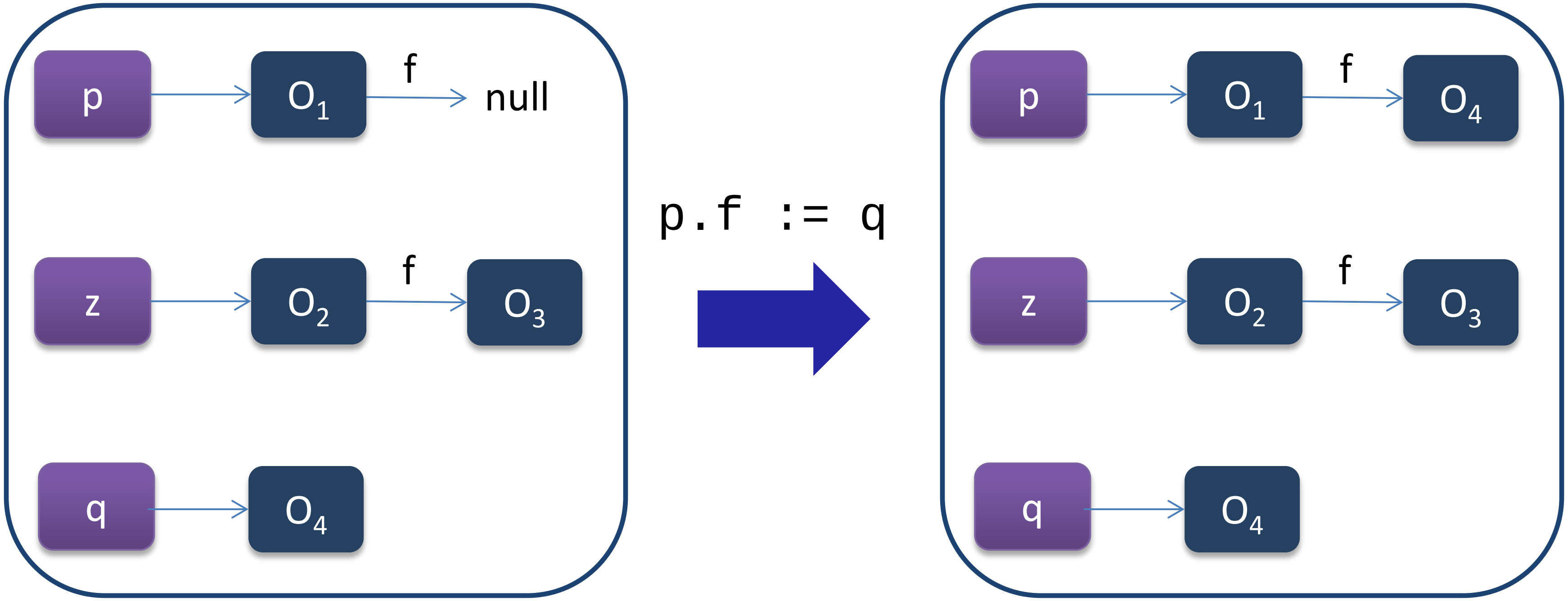
To see why this is **not correct**, we need to think what the left side means in the **concrete** and what the right side means in the **concrete**.



A Counter-Example in the Concrete

Possible Concrete Structure CE of AE1

Possible Concrete Structure **not captured** by Abstract Element AE2

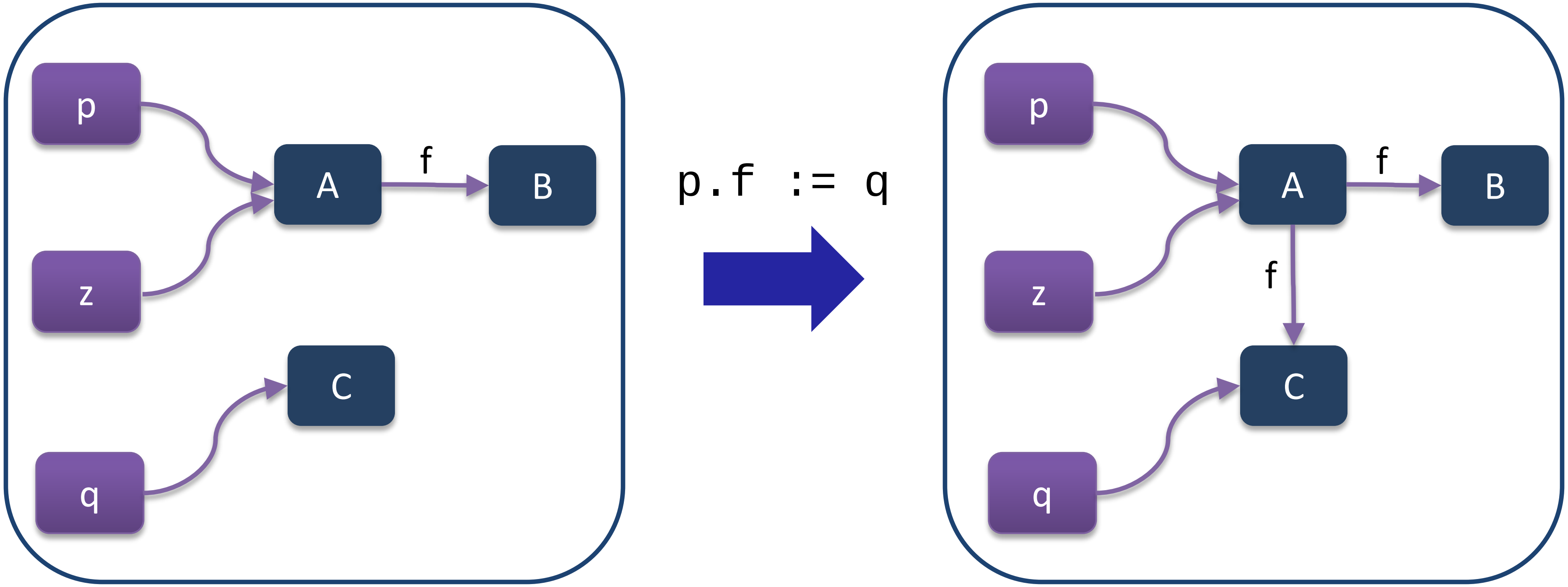


Concrete object O_1 allocated at site A
Concrete object O_2 allocated at site A
Concrete object O_3 allocated at site B
Concrete object O_4 allocated at site C

The reason this structure is not captured by AE2 is because in AE2 we can never reach an object allocated at site B via pointer z, while here, this is possible

What about `p.f := q` ?

A **correct solution** is to apply union on the contents of A.f and q, thereby obtaining that **A.f ↦ {B, C}.** This is called **weak updates**. There are techniques to perform strong updates, but we will not study them in this course.



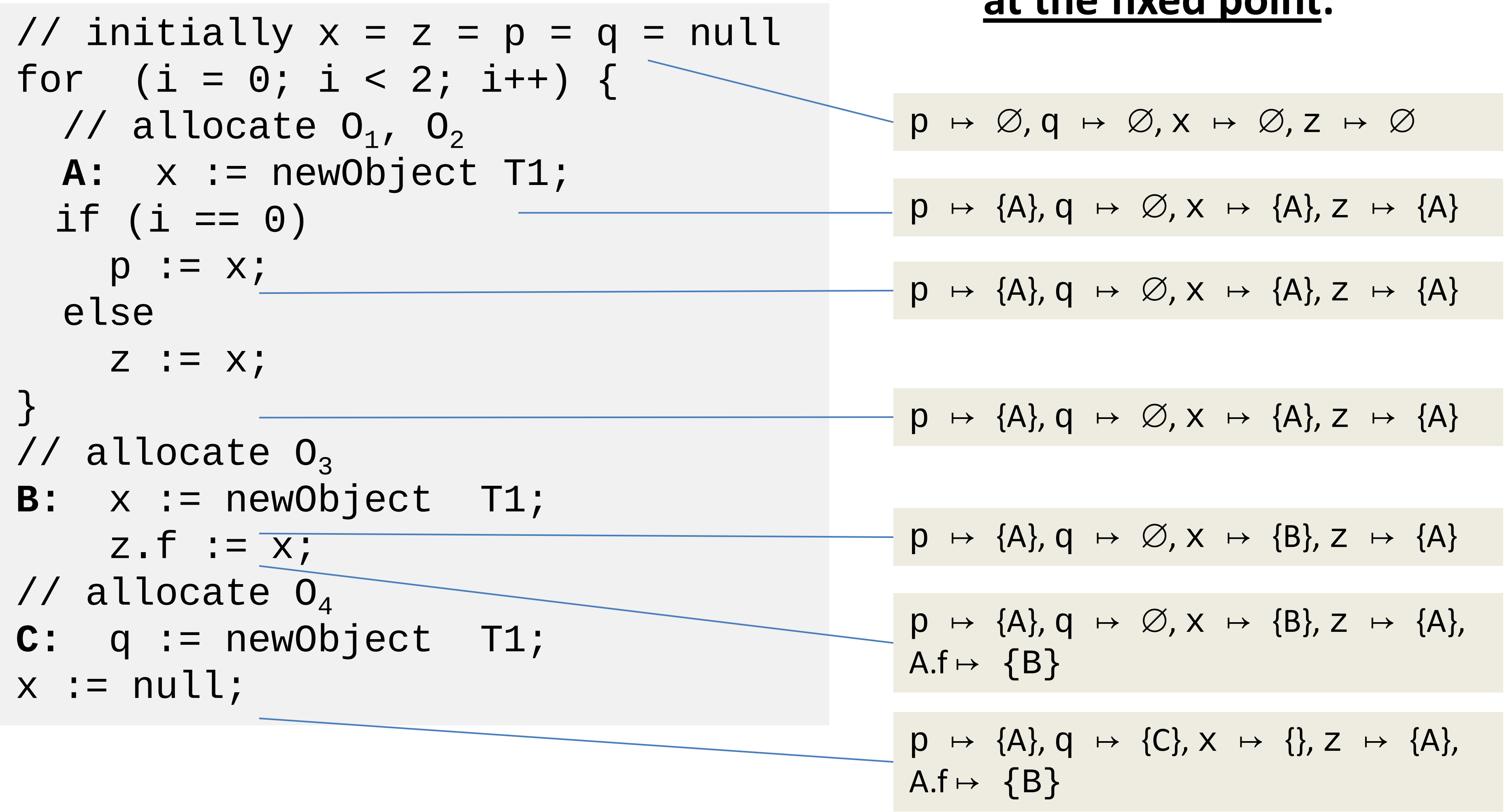
A program which produces structure CE

```
// initially x = z = p = q = null
for  (i = 0; i < 2; i++) {
  // allocate O1, O2
  A:  x := newObject T1;
  if (i == 0)
    p := x;
  else
    z := x;
}
// allocate O3
B:  x := newObject  T1;
    z.f := x;
// allocate O4
C:  q := newObject  T1;
x := null;
```

There could be many programs which produce the structure CE

Lets apply pointer analysis to the program

The result of pointer analysis
<u>at the fixed point</u>:



```
// initially x = z = p = q = null
for  (i = 0; i < 2; i++) {
  // allocate O1, O2
  A:   x := newObject T1;
  if (i == 0)
     p := x;
  else
     z := x;
}
// allocate O3
B:   x := newObject  T1;
     z.f := x;
// allocate O4
C:   q := newObject  T1;
x := null;
```

$p \mapsto \emptyset, q \mapsto \emptyset, x \mapsto \emptyset, z \mapsto \emptyset$

$p \mapsto \{A\}, q \mapsto \emptyset, x \mapsto \{A\}, z \mapsto \{A\}$

$p \mapsto \{A\}, q \mapsto \emptyset, x \mapsto \{A\}, z \mapsto \{A\}$

$p \mapsto \{A\}, q \mapsto \emptyset, x \mapsto \{A\}, z \mapsto \{A\}$

$p \mapsto \{A\}, q \mapsto \emptyset, x \mapsto \{B\}, z \mapsto \{A\}$

$p \mapsto \{A\}, q \mapsto \emptyset, x \mapsto \{B\}, z \mapsto \{A\}, A.f \mapsto \{B\}$

$p \mapsto \{A\}, q \mapsto \{C\}, x \mapsto \{\}, z \mapsto \{A\}, A.f \mapsto \{B\}$

Notes on the pointer analysis

The pointer analysis simply applies the transformers of the pointer manipulating statements from slide 20 on the **control-flow graph**. The function is the same shape as Interval domain, except applied to the pointer relevant statements:

$$F^{pointer}: (\mathrm{Lab} \to A) \to (\mathrm{Lab} \to A)$$

Here, Lab → A denotes the pointer analysis domain from slide 14.

$$F^{pointer}(m)\ell = \begin{cases} \top & \text{if } \ell \text{ is initial label} \\ \bigsqcup_{(\ell',\text{action},\ \ell)} [\![\text{action}]\!](m(\ell')) & \text{otherwise} \end{cases}$$

Example

```
p :=newObject$^1$ T1; // A1
q :=newObject$^2$ T2; // A2
if p=q $^3$ then
    z:=p $^4$
else
    z:=q $^5$
```

Allocation-site based naming (using A_{lab} instead of just “lab” for clarity)

Result of Pointer Analysis

```
p :=newObject$^1$ T1; // A1
q :=newObject$^2$ T2; // A2
if p=q $^3$ then
    z:=p $^4$
else
    z:=q $^5$
```

$p \mapsto \varnothing, q \mapsto \varnothing, z \mapsto \varnothing$

$p \mapsto \{A1\}, q \mapsto \varnothing, z \mapsto \varnothing$

$p \mapsto \{A1\}, q \mapsto \{A2\}, z \mapsto \varnothing$

$p \mapsto \varnothing, q \mapsto \varnothing, z \mapsto \varnothing$

$p \mapsto \varnothing, q \mapsto \varnothing, z \mapsto \varnothing$

$p \mapsto \{A1\}, q \mapsto \{A2\}, z \mapsto \varnothing$

$p \mapsto \{A1\}, q \mapsto \{A2\}, z \mapsto \{A2\}$

Allocation-site based naming (using A_{lab} instead of just “lab” for clarity)

A note on handling `null`

In our domain so far: `p` ↦ {A1} is interpreted to mean that pointer `p` can point to some concrete object allocated at allocation site A1 or that p can point to `null`.

This means that if our program performs `p.f := q`, this interpretation requires the analysis to consider the case `null.f := q`, meaning that the program can trigger a segmentation fault (in C) or an exception (in Java) .

Practical analyzers often ignore `null` dereferences and simply continue the analysis, which in theory, leads to over-approximation (as they execute more than what the program would execute in the concrete). Some analyzers do include the `null` element explicitly in the abstract domain to recover more precision.

Many practical analyzers (for say Java) do not however track the control flow triggered by the null pointer exception (which can be caught in Java). This means that in practice, the analysis actually computes an under-approximation.

A note on handling `null`

For our interpretation, when we have the abstract state:

$$p \mapsto \{A1\}, q \mapsto \{A2\}, z \mapsto \{A1000\}$$

and evaluate statement `if(p=q){L: //}` on that state, at label `L`, we get:

$$p \mapsto \emptyset, q \mapsto \emptyset, z \mapsto \{A1000\}$$

The reason is that the only way for `p` and `q` to be equal is when the two pointers are `null`, which in turn abstracts to $\emptyset$.

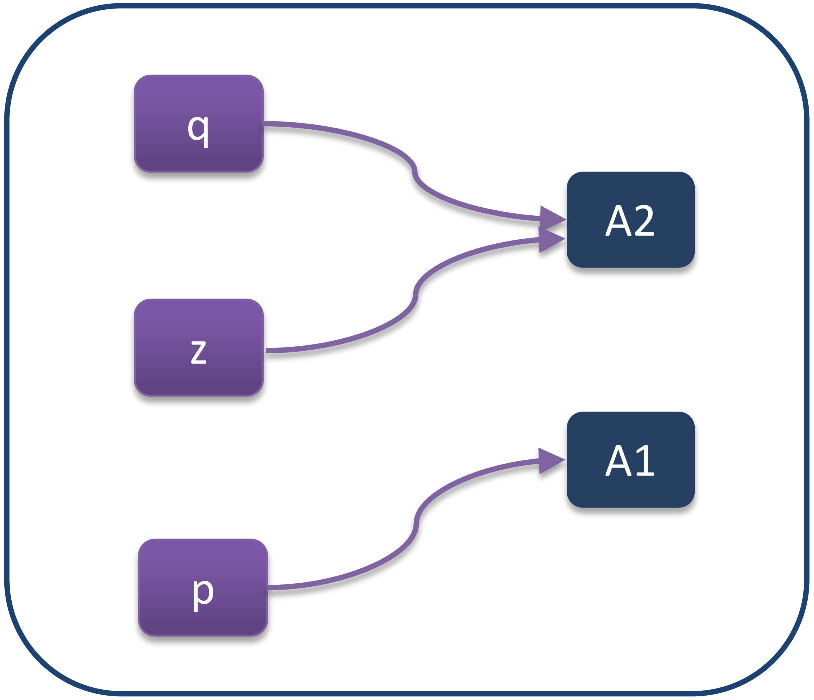
However, if we had `null` in the abstract domain, then $p \mapsto \{A1\}$ would mean that p cannot point to `null` and therefore, the result at label `L` would be:

$$p \mapsto \emptyset, q \mapsto \emptyset, z \mapsto \emptyset$$

meaning that essentially, the label is unreachable.

Flow-Sensitive: Output

Showing results at the end of the program:



3 points-to pairs

z and p do not alias

z and q alias

Pointer Analysis: two kinds

- Lets now take a look at the flow insensitive analysis.
  - Scalable points-to analysis is typically flow-insensitive

- Soot implements a few flow-insensitive analyses

Flow Insensitive Abstract Domain

$$(\texttt{PtrVar} \rightarrow \wp(\texttt{AbsObj})) \times$$
$$(\texttt{AbsObj} \times \texttt{Field} \rightarrow \wp(\texttt{AbsObj}))$$

This abstract domain does not keep information per label, essentially **ignoring the control flow** of the program.

Flow-Insensitive Analysis

```
p :=newObject$^1$ T1; // A1
q :=newObject$^2$ T2; // A2
if p=q $^3$ then
    z:=p $^4$
else
    z:=q $^5$
```

Allocation-site based naming (using A_{lab} instead of just “lab” for clarity)

Flow-Insensitive Analysis

```
p :=newObject¹ T1; // A1
q :=newObject² T2; // A2

    z:=p ⁴


    z:=q ⁵
```

Allocation-site based naming (using A_{lab} instead of just “lab” for clarity)

Flow-Insensitive Analysis

p :=newObject^1 T1; // A1

q :=newObject^2 T2; // A2

z:=p ^{4}

z:=q ^{5}

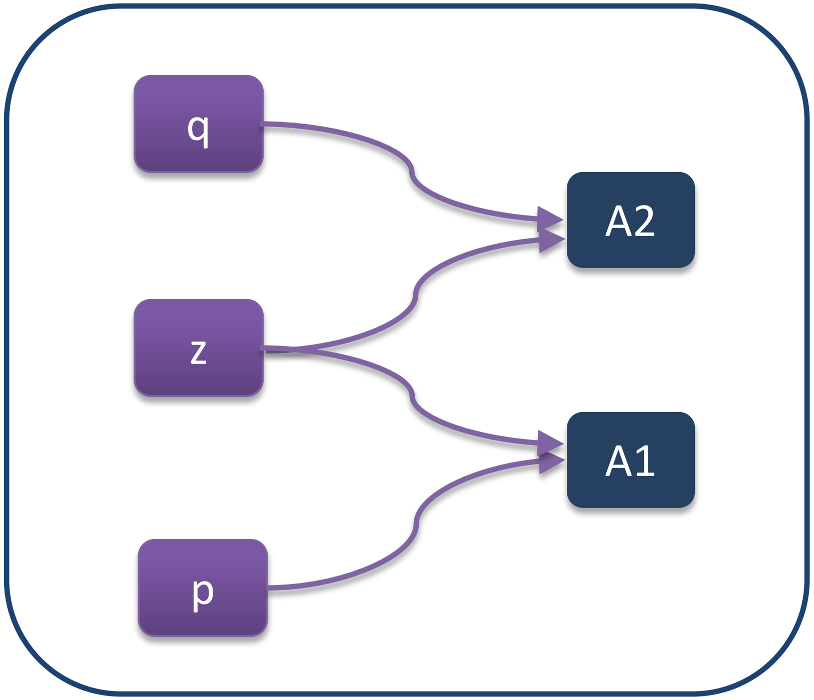
Output of Analysis:

```
p ↦ {A1},q ↦ {A2},z ↦ {A1, A2}
```

Allocation-site based naming (using A_{lab} instead of just “lab” for clarity)

Flow-Insensitive Output

At any program point we have:



4 points-to pairs

z and q alias
z and p alias

Alias Analysis

(this is a particular client of the pointer analysis)

- Once we have performed the pointer analysis, it is trivial to compute alias analysis
  - but not vice versa

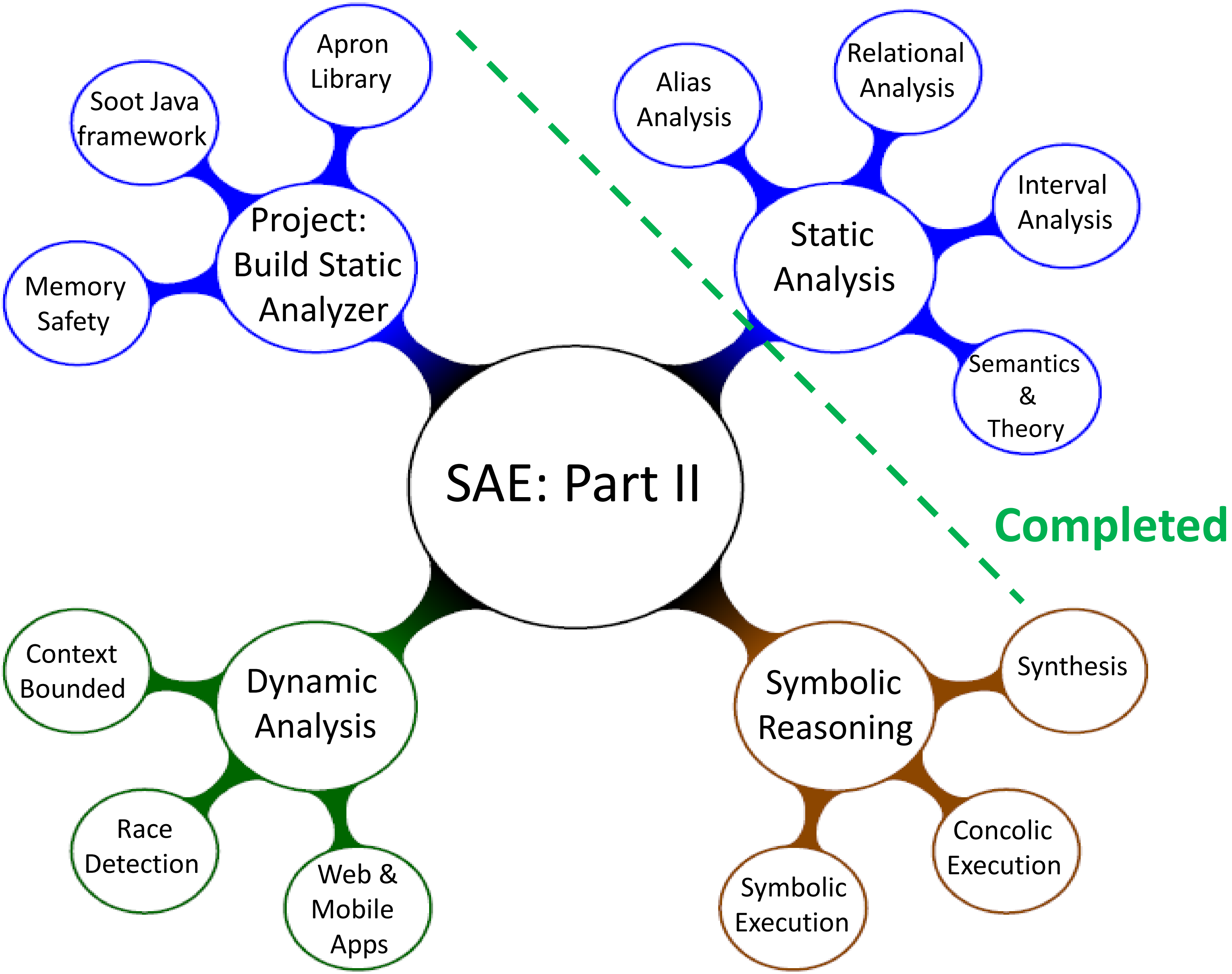
- A function points-to (p) returns the set of all abstract objects that a pointer p can point to
  - Practically, frameworks like Soot contain similar call to points-to, where one can obtain the abstract objects a pointer points to.

- Two pointers p and q may alias if:
  - $\text{points-to}(a) \cap \text{points-to}(b) \neq \emptyset$

Static Analysis

In our study of static analysis, we have studied and seen how to work with both numerical domains as well as heap domains (like pointer analysis). Both of these are very popular domains when it comes to analysis of real-world programs.

This concludes our study of static analysis and over-approximation.


SAE: Part II
Project: Build Static Analyzer
Apron Library
Soot Java framework
Memory Safety
Static Analysis
Alias Analysis
Relational Analysis
Interval Analysis
Semantics & Theory
Completed
Dynamic Analysis
Context Bounded
Race Detection
Web & Mobile Apps
Symbolic Reasoning
Synthesis
Concolic Execution
Symbolic Execution