Program Analysis

Call Graphs

Prof. Dr. Michael Pradel

Software Lab, University of Stuttgart

Winter 2021/2022

Slides adapted from Eric Bodden

Warm-up Quiz

What does this Java code print?

```
class Reflection {
  static class Car {
    private String color;
    protected void getColor() {
      System.out.println("A "+color+" car");
    }
  }
  public static void main(String[] args)
      throws Exception {
    Class clazz = Class.forName("Reflection$Car");
    Car car = (Car) clazz.newInstance();
    Method getColor = clazz.getDeclaredMethod("getColor");
    getColor.invoke(car);
  }
}
```

Warm-up Quiz

What does this Java code print?

```
class Reflection {
  static class Car {
    private String color;
    protected void getColor() {
      System.out.println("A "+color+" car");
    }
  }
  public static void main(String[] args)
      throws Exception {
    Class clazz = Class.forName("Reflection$Car");
    Car car = (Car) clazz.newInstance();
    Method getColor = clazz.getDeclaredMethod("getColor");
    getColor.invoke(car);
  }
}
```

Result: A null car

Call Graph Analysis

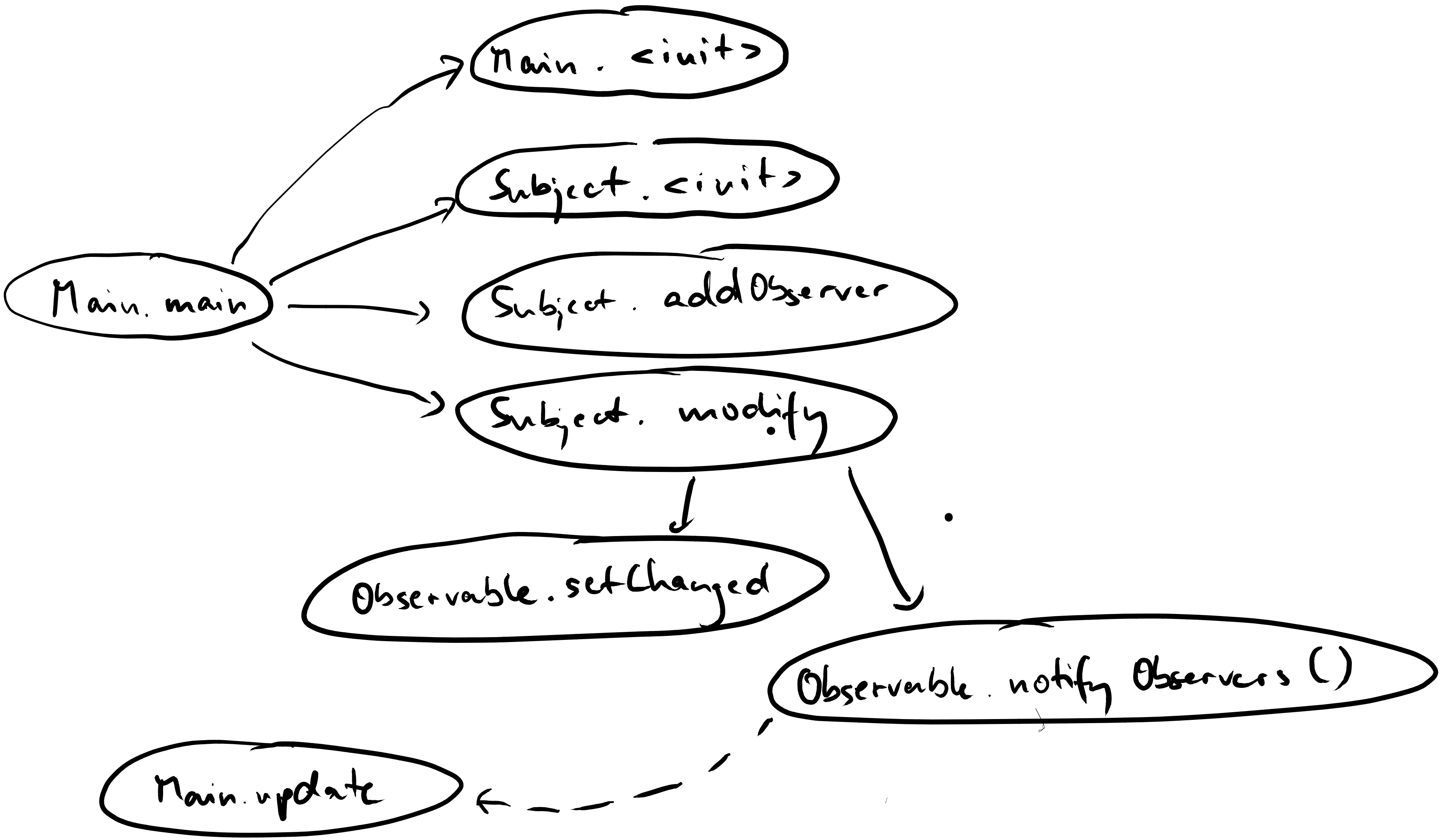
- **Call graph: Abstraction of all method calls in a program**
  - Nodes: Methods
  - Edges: Calls
  - Flow-insensitive: No execution order
- **Here: Static call graph**
  - Abstract of all calls that may execute

Example

```
public class Main implements Observer {
  public static void main(String[] args) {
    Main m = new Main();
    Subject s = new Subject();
    s.addObserver(m);
    s.modify();
  }

  public void update(Observable o, Object arg) {
    System.out.println(o+" notified me!");
  }

  static class Subject extends Observable
    public void modify() {
      setChanged();
      notifyObservers();
    }
}
```



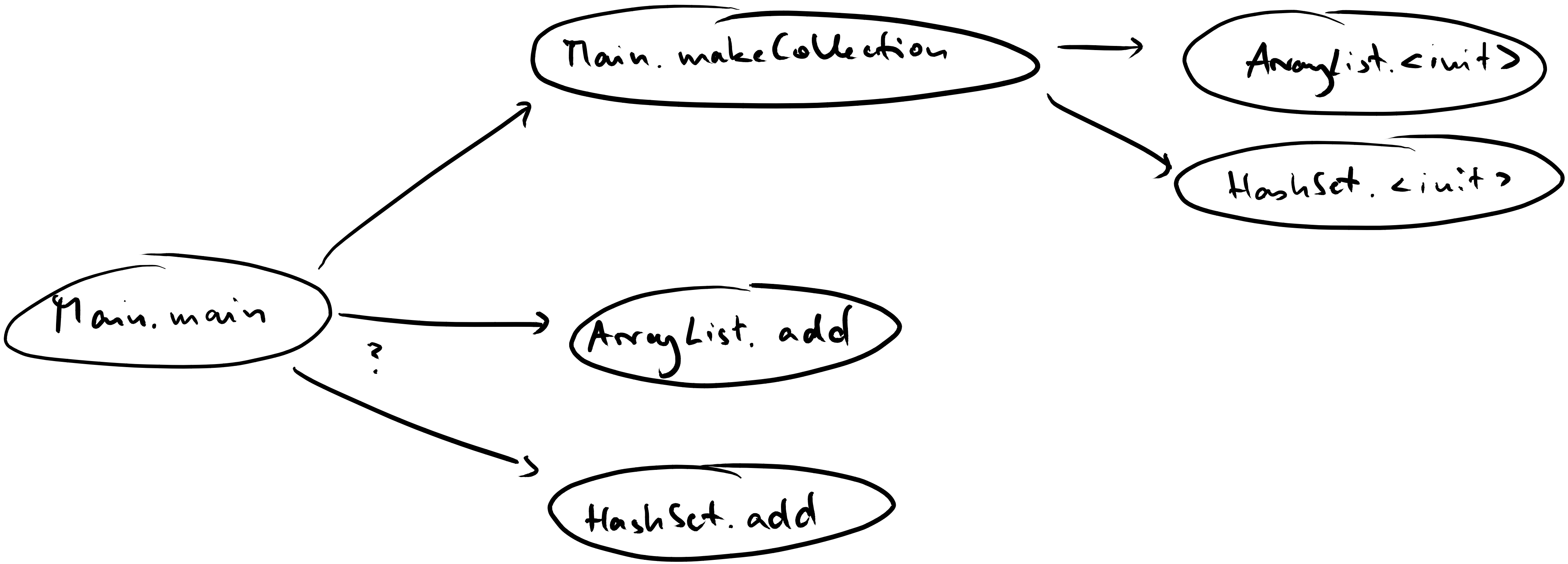
Main.<init>
Subject.<init>
Main.main
Subject.addObserver
Subject.modify
Observable.setChanged
Observable.notifyObservers()
Main.update

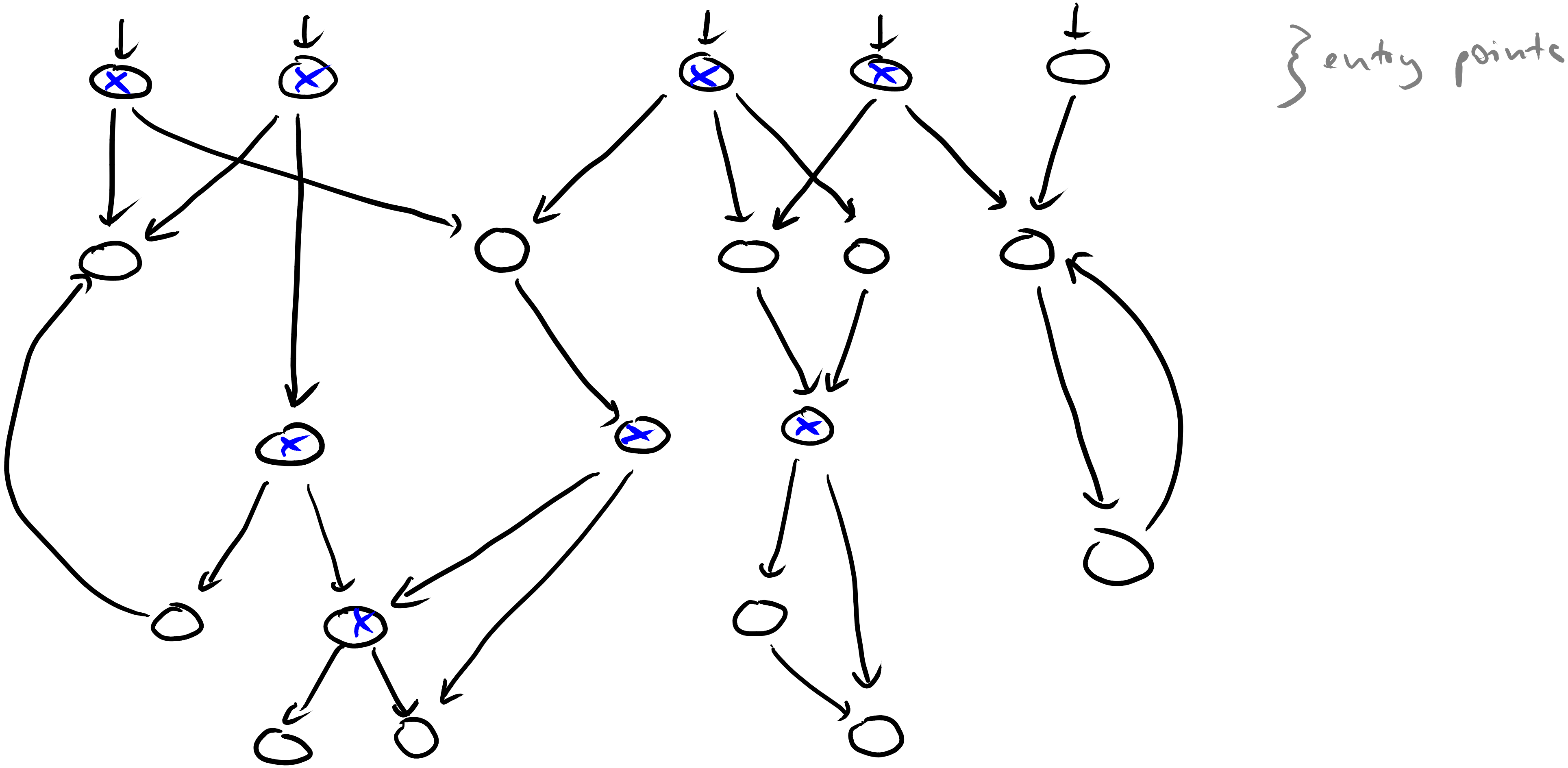
Problem: Polymorphic Calls

```
import java.util.*;

public class Main {
  public static void main(String[] args) {
    Collection c = makeCollection(args[0]);
    c.add("hello");
  }

  static Collection makeCollection(String s) {
    if(s.equals("list")) {
      return new ArrayList();
    } else {
      return new HashSet();
    }
  }
}
```


Main.makeCollection
ArrayList.<init>
HashSet.<init>
Main.main
?
ArrayList.add
HashSet.add



x .. possibly polymorphic call site

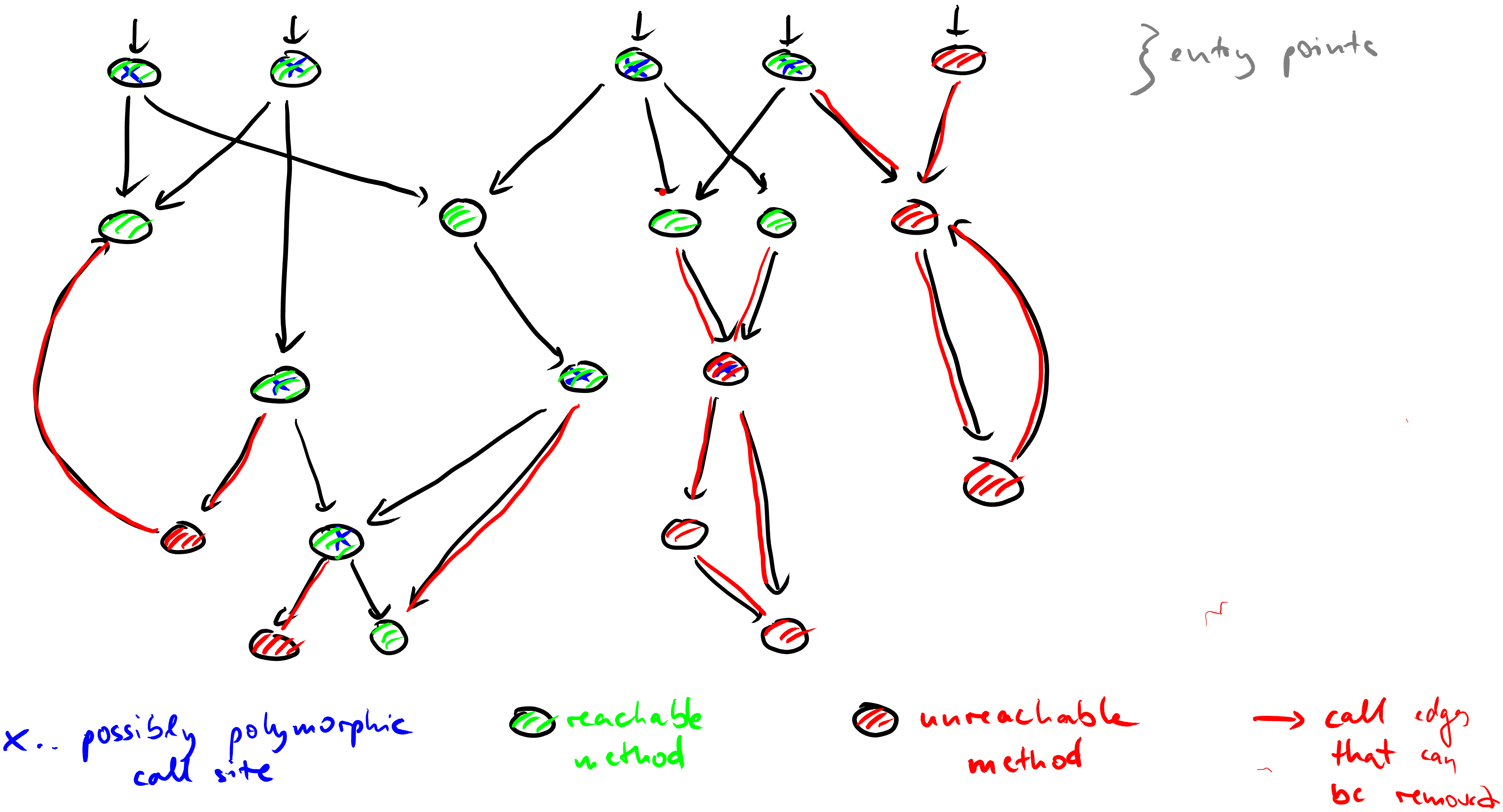
reachable method

unreachable method

→ call edges that can be removed

Improving the Call Graph

- **Prune graph:**
  Focus on feasible behavior
- **Want to minimize**
  - Reachable methods
  - Call edges
  - Potentially polymorphic call sites


entry points
x.. possibly polymorphic call site
reachable method
unreachable method
call edges that can be removed

Overview

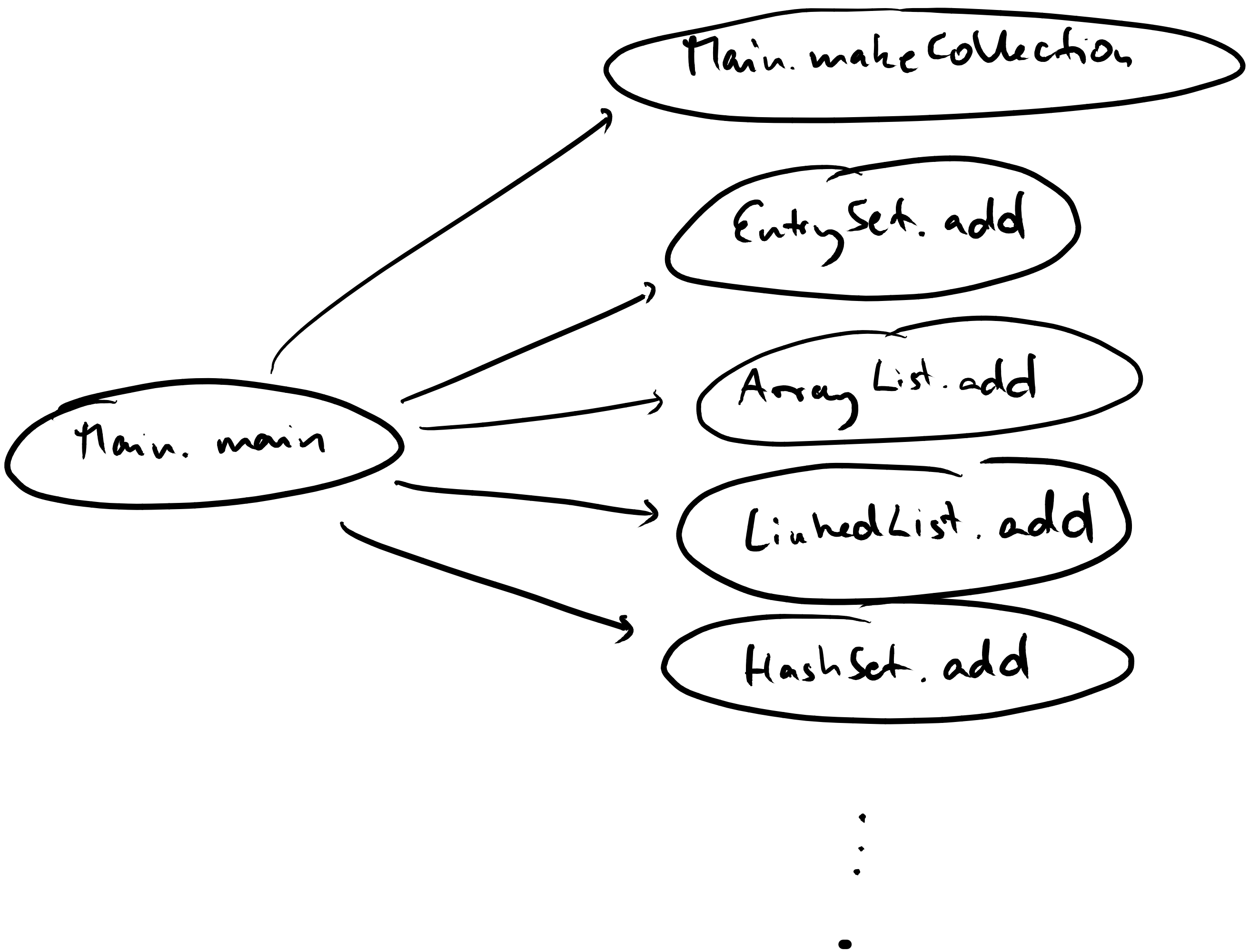
- Introduction
- Single & efficient: CHA, RTA
- Analyzing assignments: VTA, DTA
- Call graphs and points-to analysis: Spark

Five Algorithms

- **Many algorithms for call graph construction**
  - Class hierarchy analysis (CHA)
  - Rapid type analysis (RTA)
  - Variable type analysis (VTA)
  - Declared type analysis (DTA)
  - General construction framework: Spark

Class Hierarchy Analysis (CHA)

- **Most simple analysis**
- **For a polymorphic call site `m()` on declared type `T`:**
  Call edge to `T.m` and any subclass of `T` that implements `m`


Main.makeCollection
EntrySet.add
ArrayList.add
LinkedList.add
HashSet.add
Main.main

Class Hierarchy Analysis (CHA)

- **Pros**
  - Very simple
  - Correct: Contains edges for all calls that the program may execute
  - Few requirements: Needs only hierarchy, no other analysis information
- **Cons**
  - Very imprecise: Most edges will never be executed

Rapid Type Analysis (RTA)

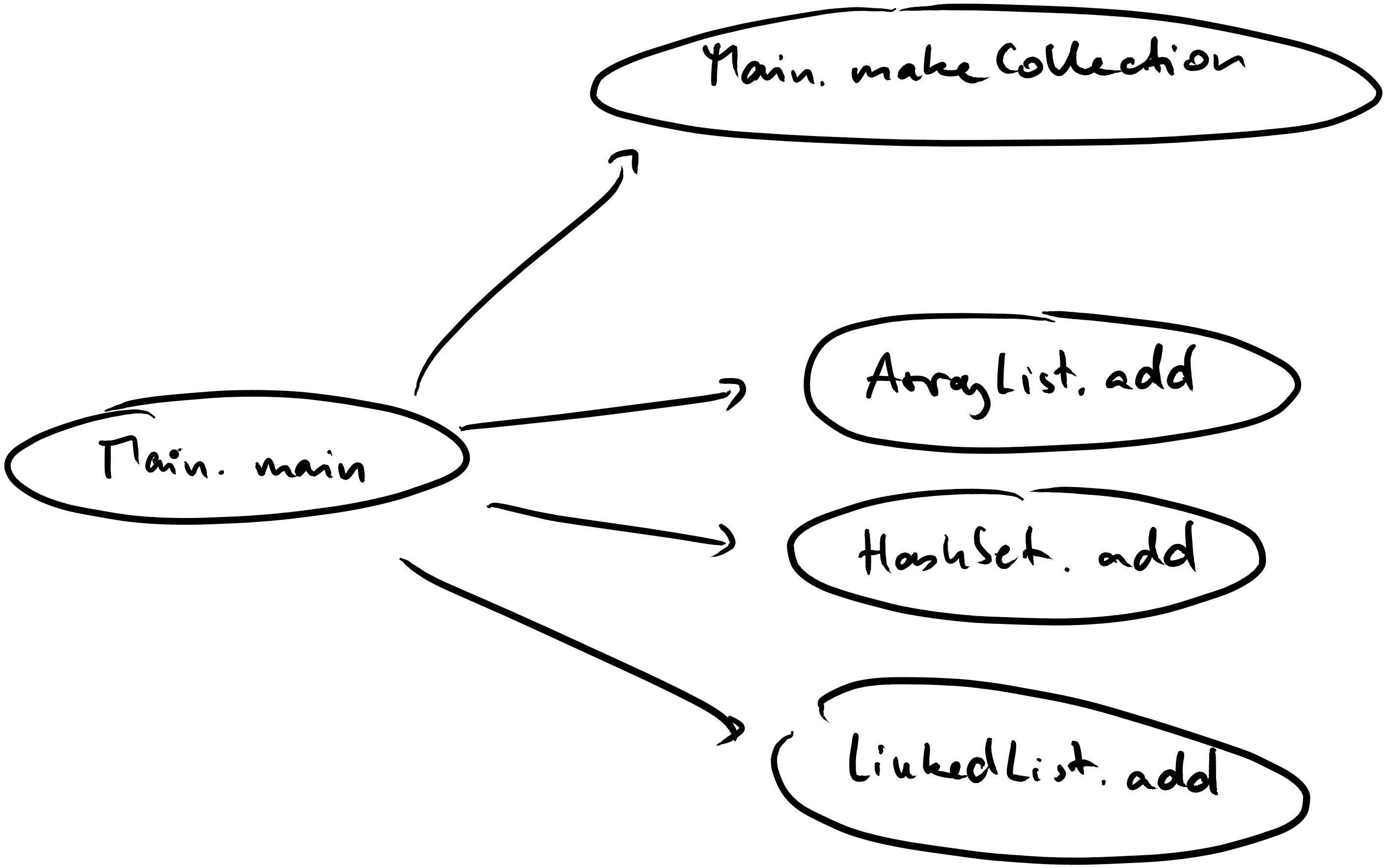
- Like CHA, but:
  Take into account only those types that the program actually instantiates

Problem: Polymorphic Calls

```
import java.util.*;

public class Main {
  public static void main(String[] args) {
    Collection c = makeCollection(args[0]);
    c.add("hello");
    new LinkedList();
  }

  static Collection makeCollection(String s) {
    if(s.equals("list")) {
      return new ArrayList();
    } else {
      return new HashSet();
    }
  }
}
```


Main. makeCollection
ArrayList. add
Main. main
HashSet. add
LinkedList. add

Rapid Type Analysis (RTA)

- **Pros**
  - Still pretty fast: Complexity is $\mathcal{O}(|Program|)$
  - Correct
  - Much more precise than CHA:

    Many unnecessary nodes and edges pruned
- **Cons**
  - Doesn't reason about assignments

Overview

- Introduction
- Single & efficient: CHA, RTA
- Analyzing assignments: VTA, DTA
- Call graphs and points-to analysis: Spark

Variable Type Analysis (VTA)

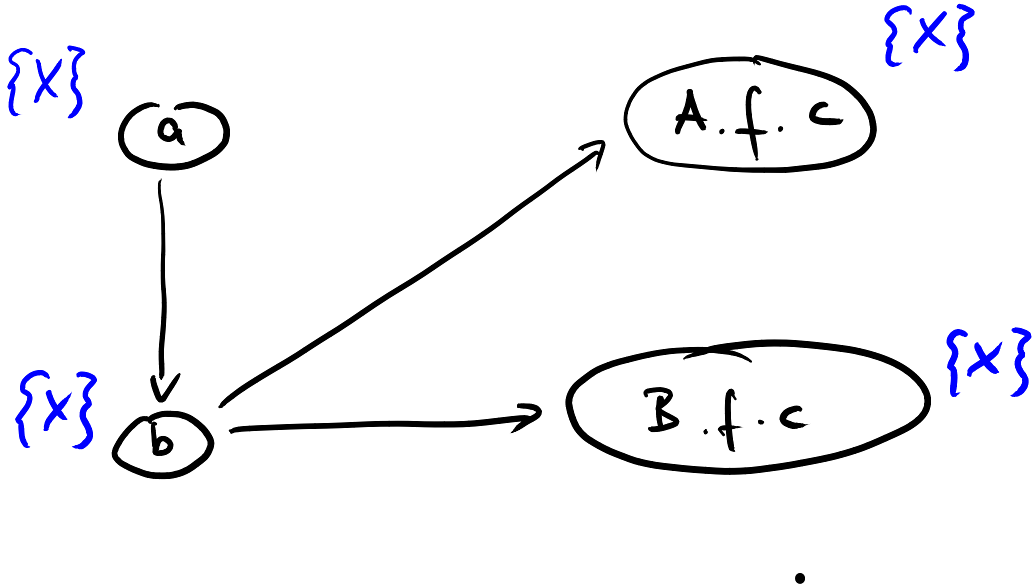
- Reason about **assignments**
- Infer what **types the objects involved in a call may have**
- Prune calls that are infeasible based on the inferred types

Example

```
a = new X();
...
b = a;
...
o.f(b);
```

```
public class A {
  public void f(C c) {
    c.m();
  }
}

public class B {
  public void f(C c) {
    c.m();
  }
}
```



{x}
a
{x}
b
A.f.c
{x}
B.f.c
{x}

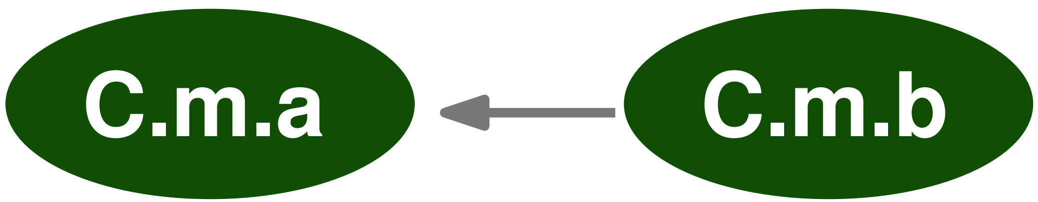
Type Propagation

Four steps:

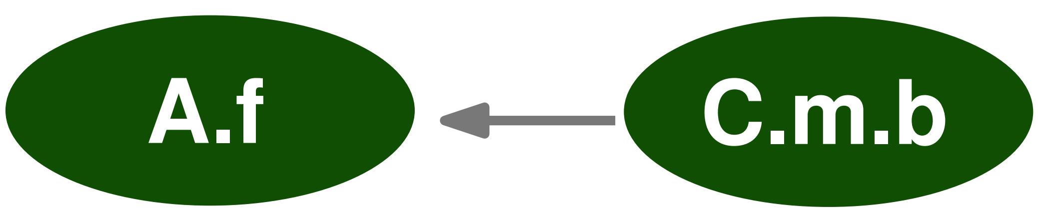
- Form initial conservative call graph
  - E.g., using CHA or RTA
- Build type-propagation graph
- Collapse strongly connected components
- Propagate types in one iteration

Building Type Propagation Graph

- Assume statement `a = b;` is in method `C.m`



- Assume another statement `a.f = b;` where field `f` is declared in `A`

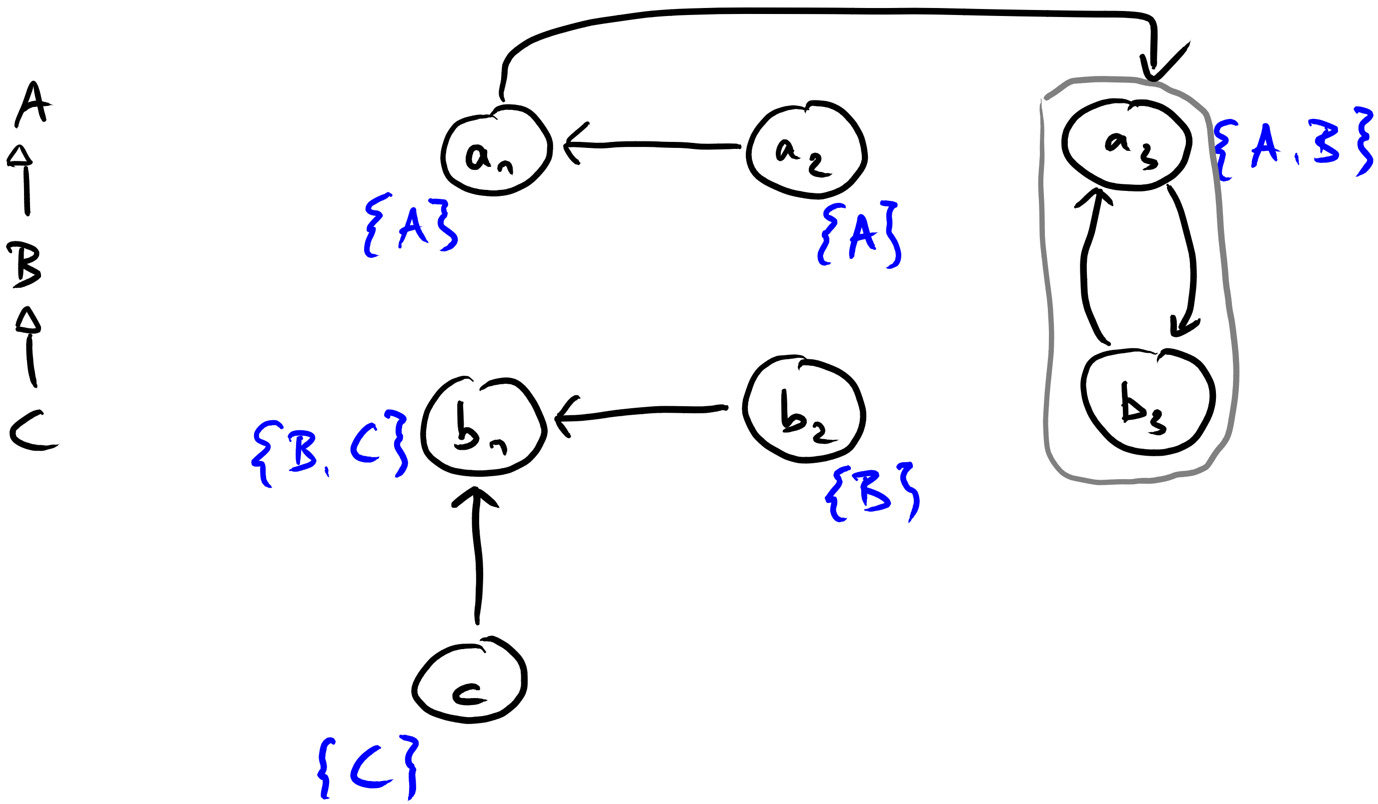


Example

```
A a1, a2, a3; B b1, b2, b3; C c;

a1 = new A();
a2 = new A();
a3 = new B();
b1 = new B();
b2 = new B();
b3 = new B();
c = new C();

a1 = a2;
b3 = (B) a3;
a3 = b3;
a3 = a1;
b1 = b2;
b1 = c;
```



A
B
C
a1
a2
a3
b1
b2
b3
c
{A}
{A}
{A, B}
{B, C}
{B}
{C}

Side Note: Field Representations

How does the analysis represent `a.f`?

- Field-sensitive: Represented as `a.f`
- Field-insensitive: Represented as `a.*` or `a`
- Field-based: Represented as `A.f`, where `A` is class of `a`

Side Note: Field Representations

How does the analysis represent `a.f`?

- Field-sensitive: Represented as `a.f`
- Field-insensitive: Represented as `a.*` or `a`
- Field-based: Represented as `A.f`, where `A` is class of `a`

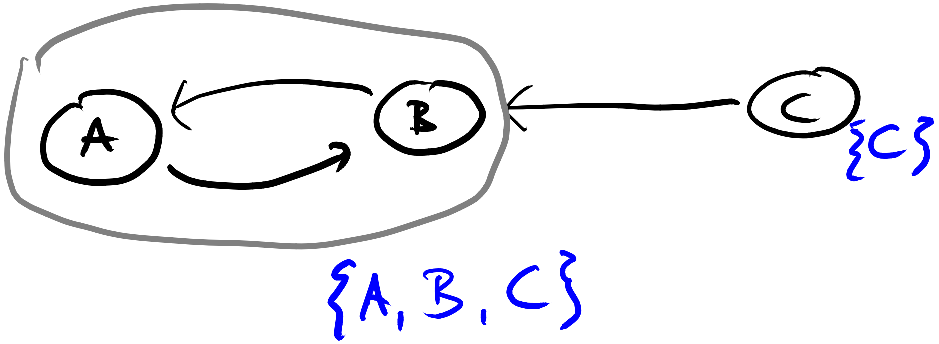
VTA is field-based

Variable Type Analysis (VTA)

- **Pros**
  - More precise than RTA: Considers only those types that may actually reach the call site
  - Still relatively fast
- **Cons**
  - Requires initial call graph (i.e., actually a refinement algorithm)
  - Some imprecision remains, e.g., because of field-based analysis

Declared-Type Analysis (DTA)

- "Small brother of VTA"
- Also reasons about assignments and how they propagate types
- But: Not per variable, but per type


A
B
C
{C}
{A, B, C}

Declared-Type Analysis (DTA)

- **Pros**
  - Faster than VTA: Graph is smaller, propagation is faster
  - More precise than RTA
- **Cons**
  - Less precise than VTA: Does not distinguish variables of same type

Overview

- Introduction
- Single & efficient: CHA, RTA
- Analyzing assignments: VTA, DTA
- Call graphs and points-to analysis: Spark

Spark: Idea

- **RTA, DTA, and VTA: Instances of one single unifying framework**
- **General recipe**
  - First, built pointer-assignment graph (PAG)
  - Propagate information through graph
- **Combine call graph construction with points-to analysis**
  - Reason about objects a variable may refer to

Pointer-Assignment Graph (PAG)

- **Nodes**
  - Allocation
  - Variable
  - Field reference
- **Edges**
  - Allocation
  - Assignment
  - Field store
  - Field load

Pointer-Assignment Graph (PAG)

- **Nodes**
  - Allocation
  - Variable
  - Field reference
- **Edges**
  - Allocation
  - Assignment
  - Field store
  - Field load

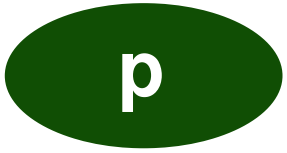
- **One for each `new A()`**
- **Represents a set of objects**
- **Has an associated type, e.g., `A`**

$alloc_1$

Pointer-Assignment Graph (PAG)

- **Nodes**
  - Allocation
  - Variable
  - Field reference
- **Edges**
  - Allocation
  - Assignment
  - Field store
  - Field load

- **One for each local variable, parameter, static field, and thrown exception**
- **Represents a memory location holding pointers to objects**
- **May be typed (depends on setting)**



Pointer-Assignment Graph (PAG)

- **Nodes**
  - Allocation
  - Variable
  - Field reference
- **Edges**
  - Allocation
  - Assignment
  - Field store
  - Field load

- **One for each `p.f`**
- **Represents a pointer dereference**
- **Has a variable node as its base, e.g., `p`**
- **Also models contents of arrays: `a.<elements>`**

Pointer-Assignment Graph (PAG)

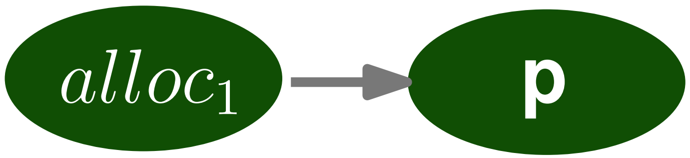
- **Nodes**
  - Allocation
  - Variable
  - Field reference
- **Edges**
  - Allocation →
  - Assignment
  - Field store
  - Field load

- **Represents allocation of an object assigned to a variable**
- **E.g., for**

```
p = new HashMap();
```

or

```
s="foo";
```

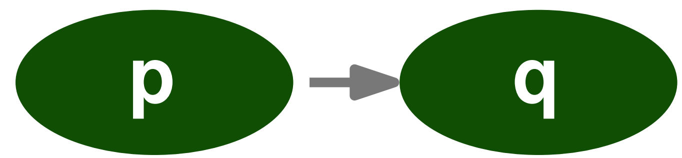


Pointer-Assignment Graph (PAG)

- **Nodes**
  - Allocation
  - Variable
  - Field reference
- **Edges**
  - Allocation
  - Assignment
  - Field store
  - Field load

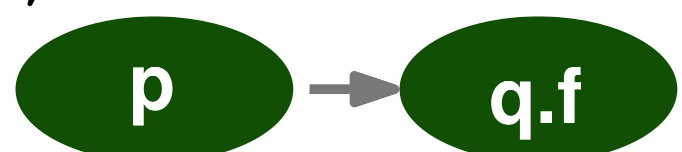
- **Represent assignments among variables and fields**
- **E.g., for**

`q = p;`



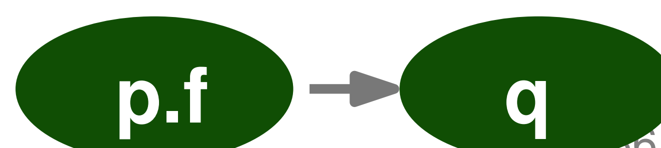
or

`q.f = p;`



or

`q = p.f;`



Example

```
static void foo() {
  p = new A(); // alloc₁
  q = p;
  r = new B(); // alloc₂
  p.f = r;
  t = bar(q);
  t.m();
}

static C bar(C s) {
  return s.f;
}
```

Points-to Sets

- **For each variable, set of objects the variable may refer to**
  - Objects represented as allocation nodes
- **Example:**

```
a = new X(); // alloc1
...
a = new Y(); // alloc2
```

$$pts(a) = \{alloc_1, alloc_2\}$$

Subset-based Analysis

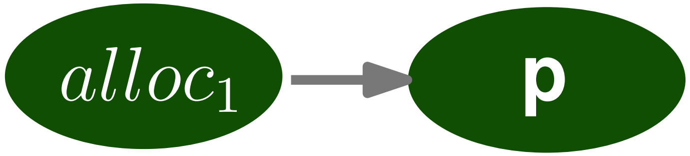
- **Allocation and assignment edges induce subset constraints**
  - Reason: Just because we know that

    ```
    p = new 1;
    ```

    does not mean that later we cannot see

    ```
    p = new 2;
    ```

- **Example:**



induces constraint

$$\{alloc_1\} \subseteq pts(p)$$

Subset-based Analysis

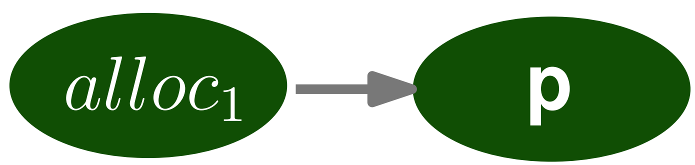
- **Allocation and assignment edges induce subset constraints**
  - Reason: Just because we know that

    ```
    p = new 1;
    ```

    does not mean that later we cannot see

    ```
    p = new 2;
    ```

- **Example:**



induces constraint

$$\{alloc_1\} \subseteq pts(p)$$

Note: Analysis is flow-insensitive, i.e., values are never assumed to be overwritten

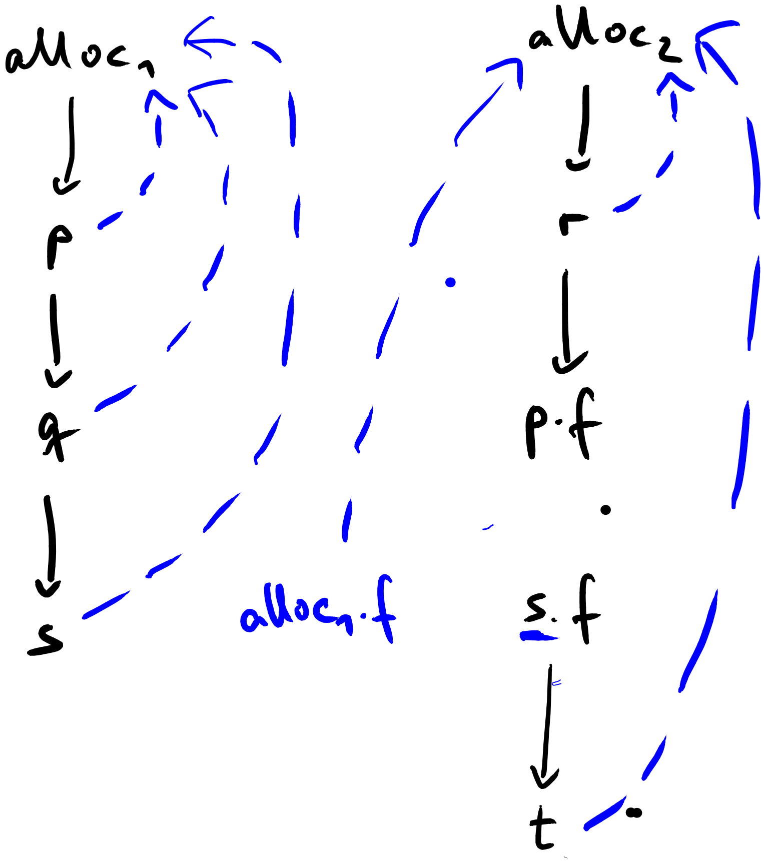
Computing Points-to Sets

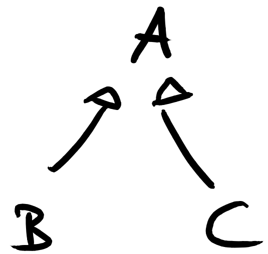
- **New helper node: Concrete fields**
- **Represents all objects pointed to by field `f` of all objects created at allocation site**
  - E.g., $alloc_1.f$

Computing Points-to Sets (2)

Iterative propagation algorithm

- Initialize $pts(v)$ according to allocation edges
- Repeat until no changes
  - Propagate sets along assignment edges $a \rightarrow b$
  - For each load edge $a.f \rightarrow b$:
    - For each $c \in pts(a)$, propagate $pts(c.f)$ to $pts(b)$
  - For each store edge $a \rightarrow b.f$:
    - For each $c \in pts(b)$, propagate $pts(a)$ to $pts(c.f)$





Call t.m() goes to B.m()

- - -> points-to

Simpler Variants

- **Spark framework supports many variants**
  - Just one allocation site per type
  - Fields simply represented by their signature
  - Equality instead of subsets for assignments
  - Etc.

Spark

- **Pros**
  - Generic algorithm where precision and efficiency can be tuned
  - Jointly computing call graph and points-to sets increases precision
- **Cons**
  - Still flow-insensitive
  - Can be quite expensive to compute