A Tour of Pointer Analysis

Ondřej Lhoták

University of Waterloo

Pointer analysis

Pointer Analysis: Haven't We Solved This Problem Yet?

Michael Hind
IBM Watson Research Center
30 Saw Mill River Road
Hawthorne, New York 10532
hind@watson.ibm.com

ABSTRACT

1. INTRODUCTION

2. BACKGROUND

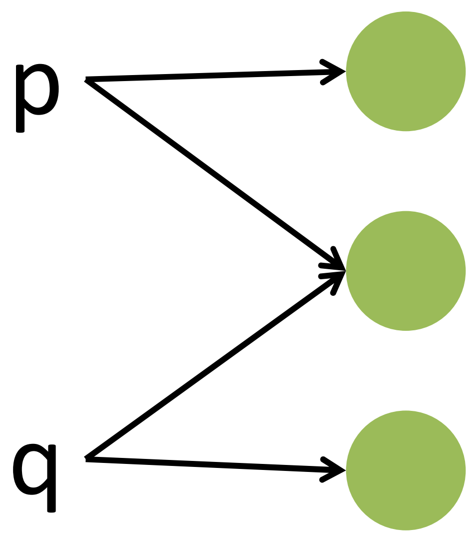
3. GENERAL ISSUES

3.1 Terminology

[PASTE 2001]

What does pointer analysis do?

For each pointer (reference) in the program, what memory locations (objects) does it point to?



Why pointer analysis?

```
a = 1
b = 2


c = a + b 3
```

Why pointer analysis?

```
a = 1
b = 2
*x = 4
c = a + b ?
```

Why pointer analysis?

```
a = 1
b = 2
*x = 4
c = a + b ?
```

If x == &a, then c = 6.

If x == &b, then c = 5.

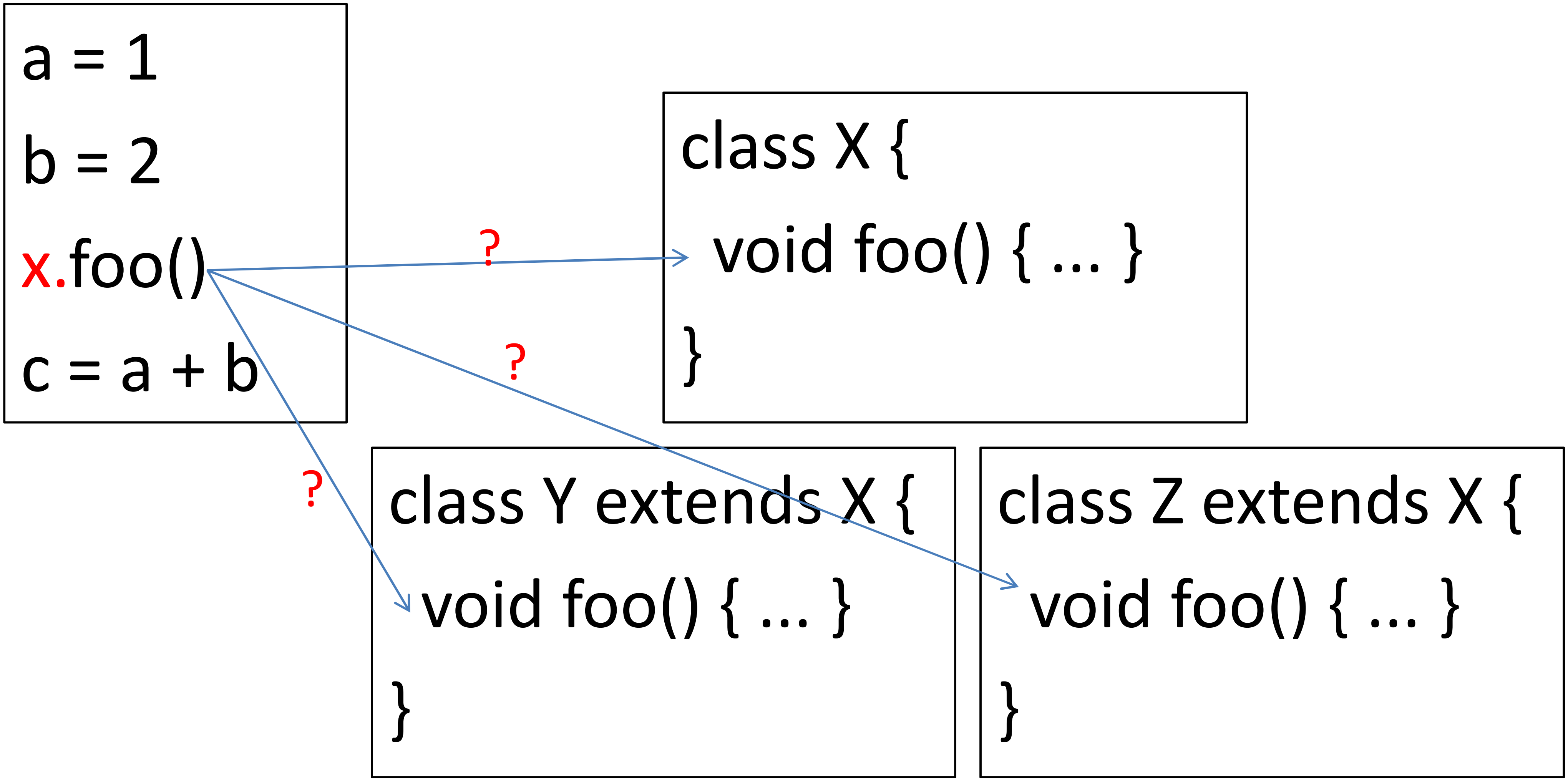
If x != &a && x != &b, then c = 3.

Why pointer analysis?

```
a = 1
b = 2
foo()
c = a + b
```

```
void foo() { ... }
```

Why pointer analysis?

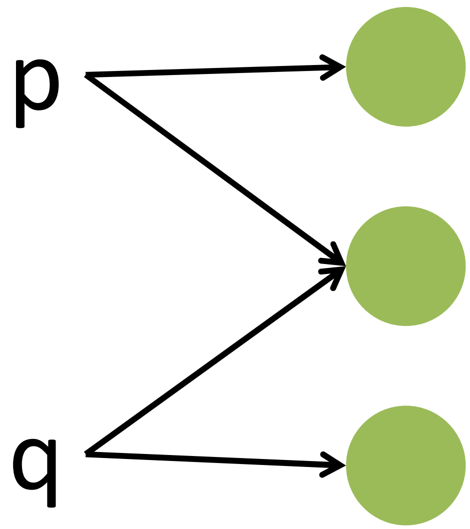


Applications of pointer analysis

- Call graph construction
- Dependence analysis and optimization
- Cast check elimination
- Side effect analysis
- Escape analysis
- Slicing
- Parallelization
- ...

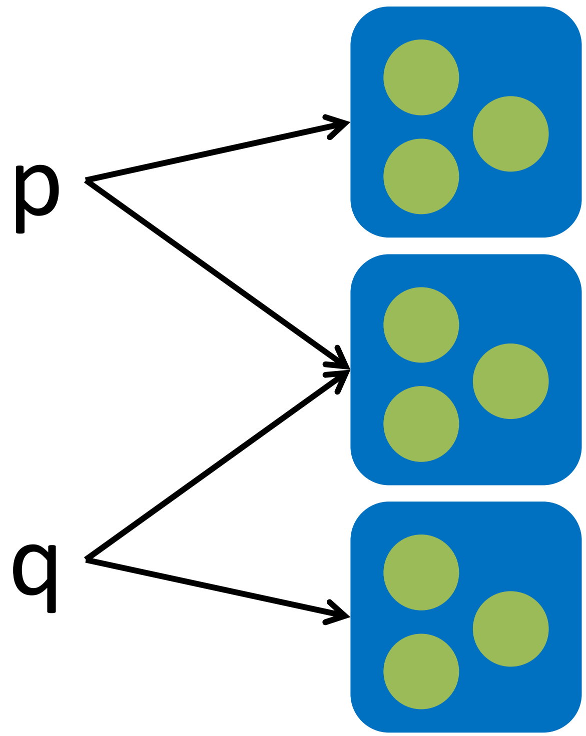
Pointer analysis as an abstraction

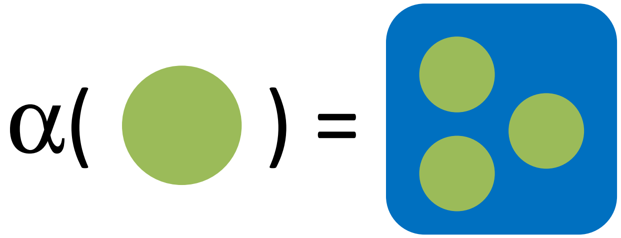
For each pointer (reference) in the program, what memory locations (objects) does it point to?



Pointer analysis as an abstraction

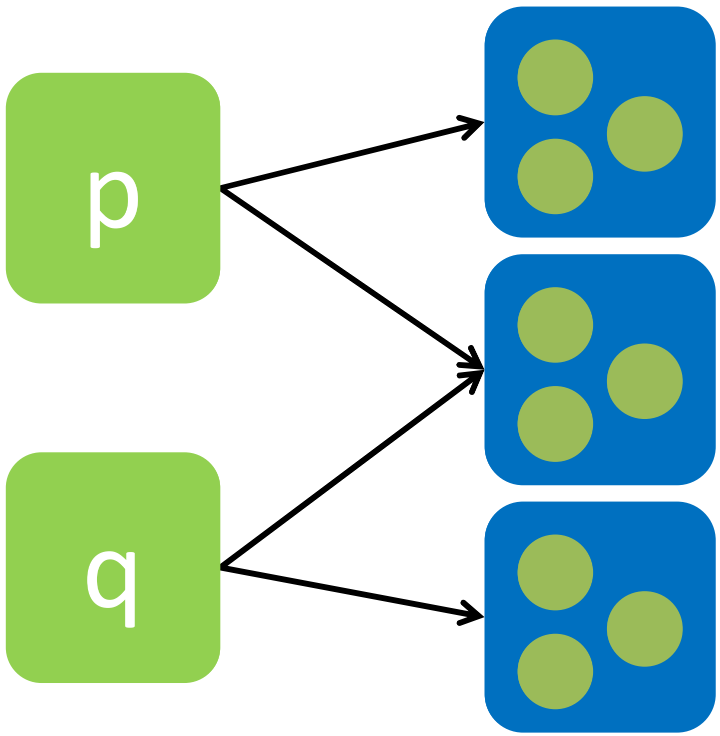
For each pointer (reference) in the program, what memory locations (objects) does it point to?





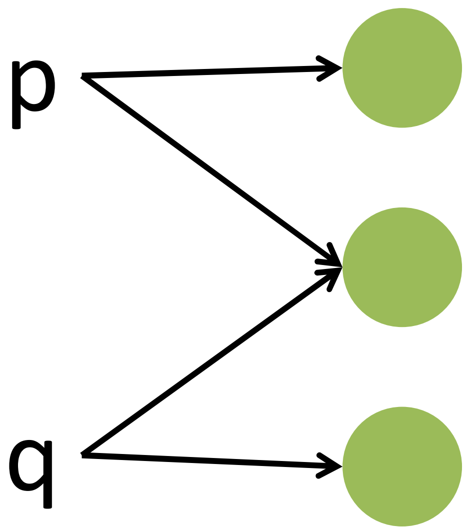
Pointer analysis as an abstraction

For each pointer (reference) in the program, what memory locations (objects) does it point to?

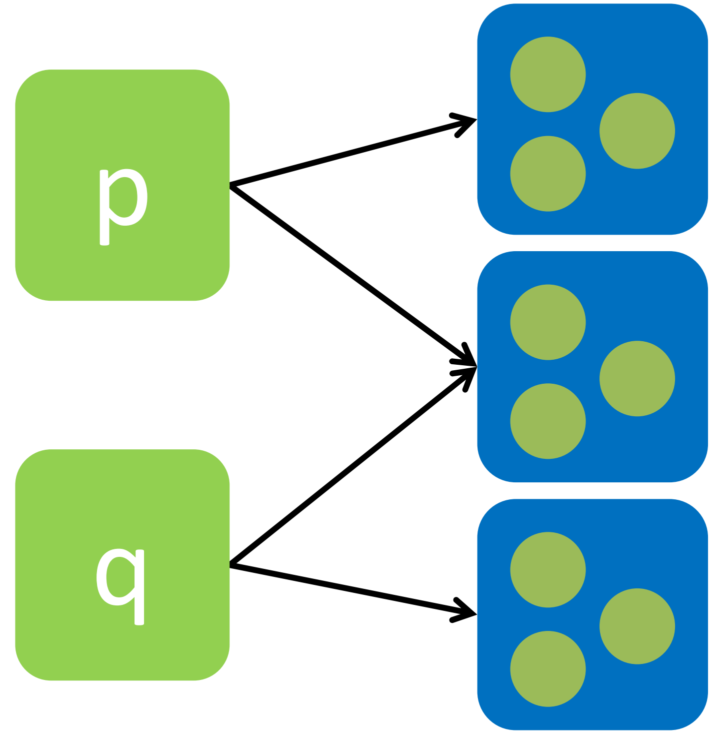


Pointer analysis as an abstraction

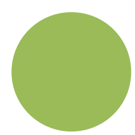
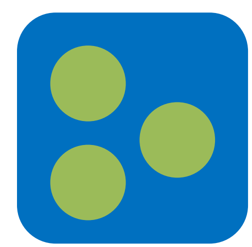
Concrete program execution



Abstract analysis



α(p) = 

α() = 

Precision of points-to sets

← more precise　　　　less precise →

unsound　　uncomputable　　　　conservative

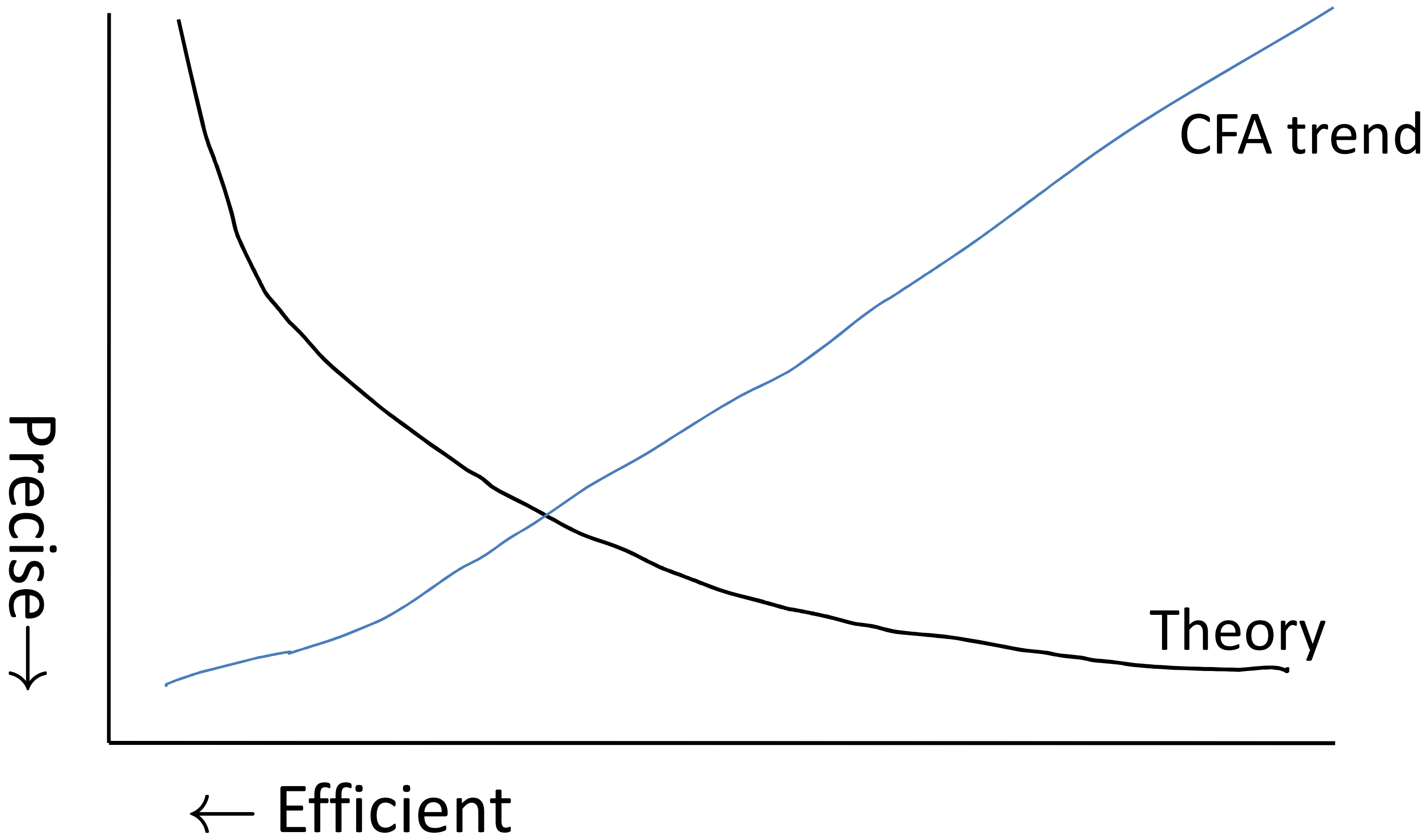
{L1} ⊂ {L1, L2} ⊂ {L1, L2, L3} ⊂ {L1, L2, L3, L4}

actual behaviour on **some** executions

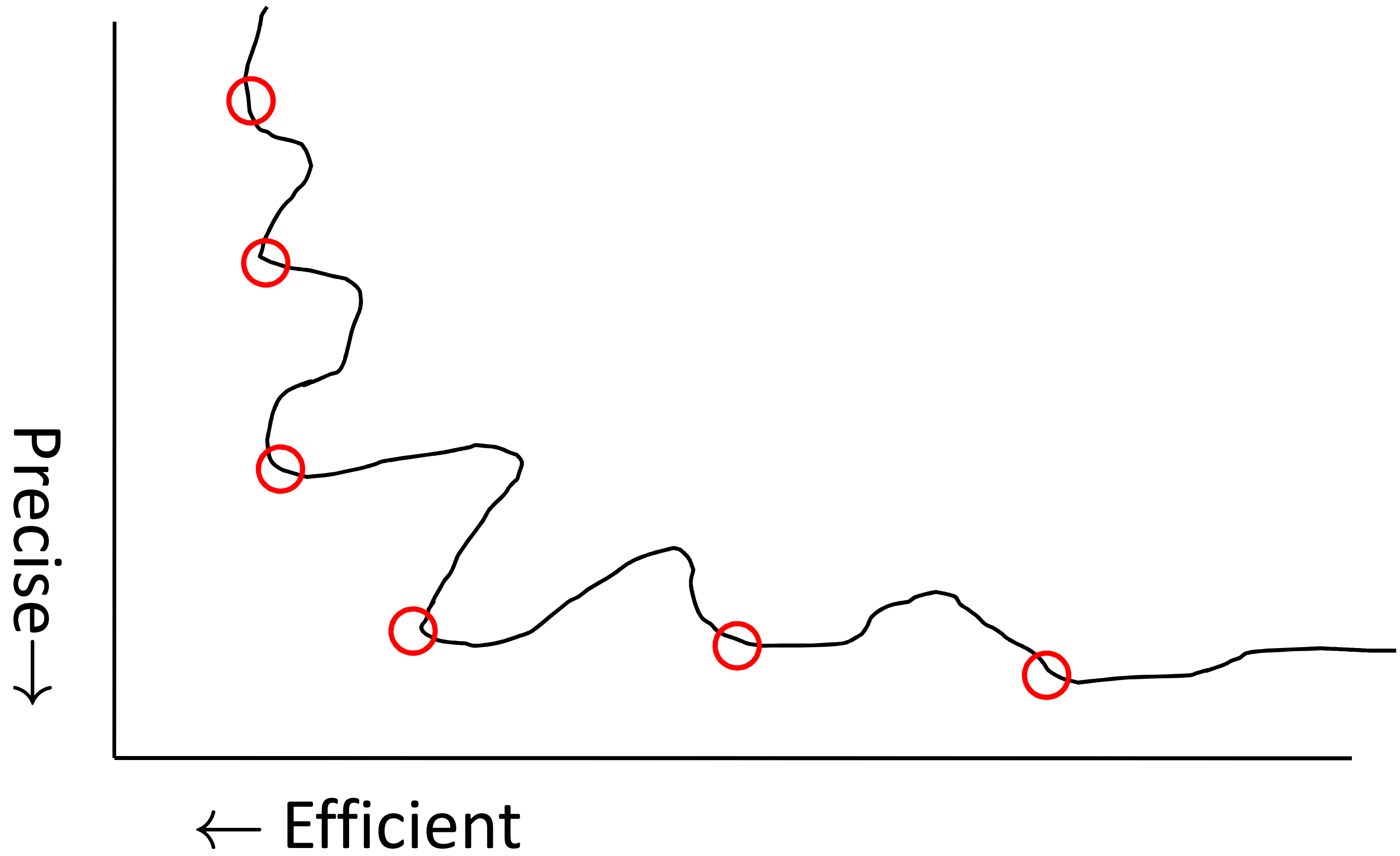
actual behaviour on **all** executions

possible analysis results

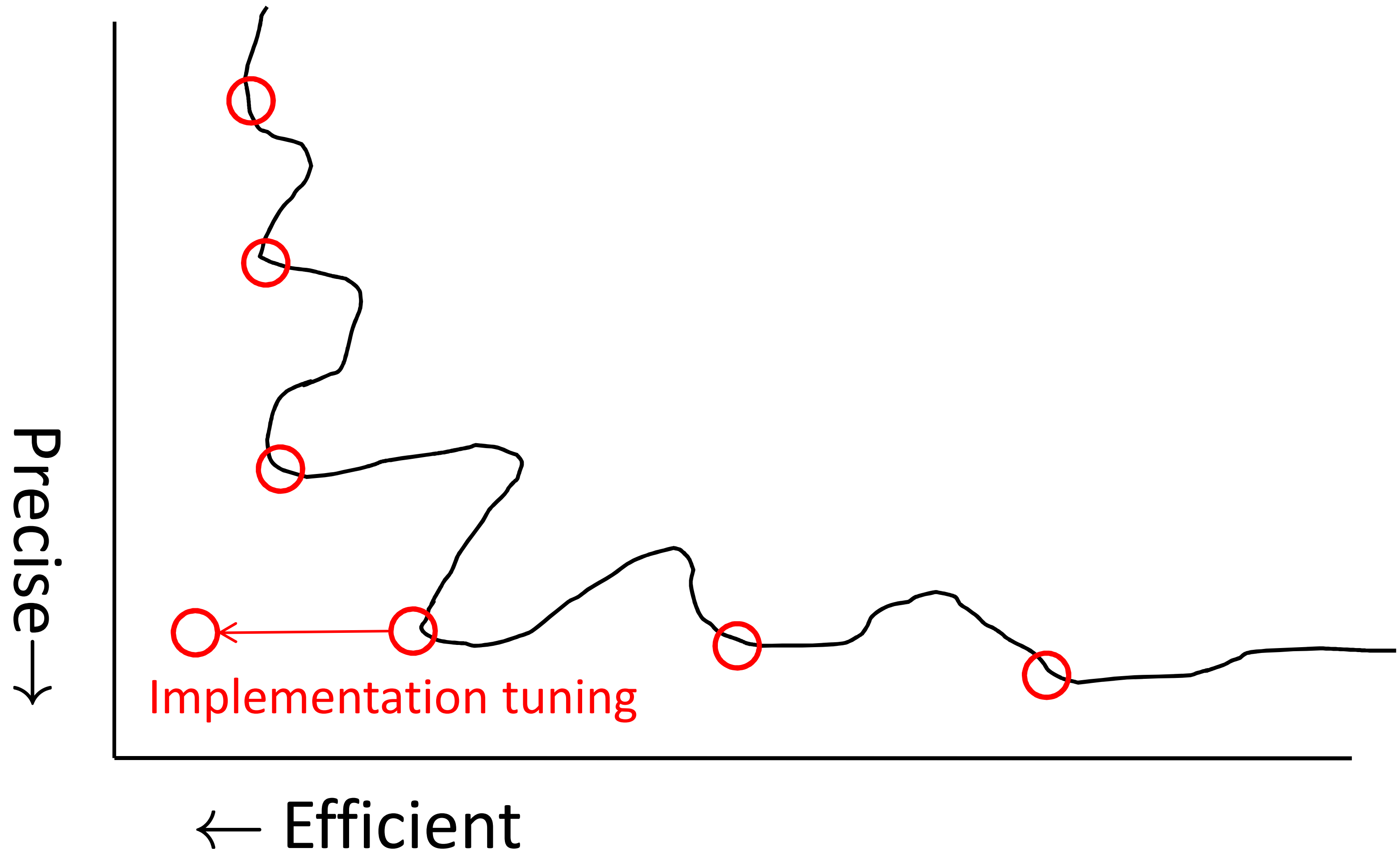
Precision vs. efficiency



Precision vs. efficiency



Precision vs. efficiency



Precision for a specific application

$\{L1, L2\} \subset \{L1, L2, L3, L4\}$

↑

This points-to set is more precise because it is smaller.

But suppose a particular application only cares whether L1 is in the set. Then <u>for that application</u>, both sets are equally precise.

Thus, precision/efficiency tradeoff must consider the application.

Design decisions for precision/efficiency

- The abstraction (affects precision and efficiency):
  - Type filtering
  - Field sensitivity
  - Directionality
  - Call graph construction
  - Context sensitivity
  - Flow sensitivity
- Algorithm and implementation (affects efficiency)
  - Propagation algorithm
  - Set implementation

An example abstraction and analysis

- The abstraction:
  - Type filtering
  - Field sensitivity
  - Directionality
  - Call graph construction
  - Context sensitivity
  - Flow sensitivity

First example:

- without type filtering
- field-sensitive
- subset-based
- ahead-of-time call graph
- context-insensitive
- flow-insensitive

Pointer Assignment Graph abstraction

Abstract object node:

Java:
L1: x = new Object()

C:
L1: x = malloc(42)

Represents some set of run-time objects (targets of pointers).
e.g. all objects allocated at a given allocation site
e.g. all objects of a given dynamic type

Pointer Assignment Graph abstraction

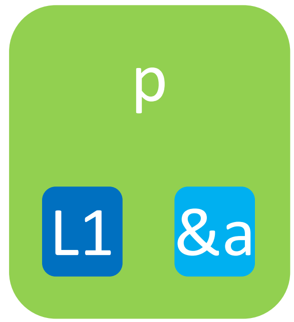
Address-of abstract object node:


```
C:
x = &a
```

Represents some set of run-time objects (targets of pointers).
e.g. the object whose address is &a.

Pointer Assignment Graph abstraction

Pointer variable node:



pt(p) = {L1, &a}

Points-to set

Represents some pointer-typed variable(s).
e.g. all instances of the local variable p in method m.

Pointer Assignment Graph abstraction

Pointer dereference node:

p.f

*p

Java:
y = p.f

C:
y = *p

Represents a dereference of some pointer (where the pointer is a pointer variable node).

Pointer Assignment Graph abstraction

Heap pointer node:

pt(L1.f) = {L2, L3}

Represents a pointer stored in some object in the heap.

State space (the analysis result)

pt(p) = { L1 , L2 , &q }

pt(L1.f) = { L1 , L2 , &q }

$$pt : (Var \cup (Obj \times Field)) \rightarrow \wp(Obj)$$

$$pt : (Var \times Obj) \quad \cup \quad (Obj \times Field \times Obj)$$

where Obj = Alloc $\cup$ AddrOf

State space (the analysis result)

pt(p) = { L1 , L2 , &q }

pt(L1.f) = { L1 , L2 , &q }

Pointer Assignment Graph edges

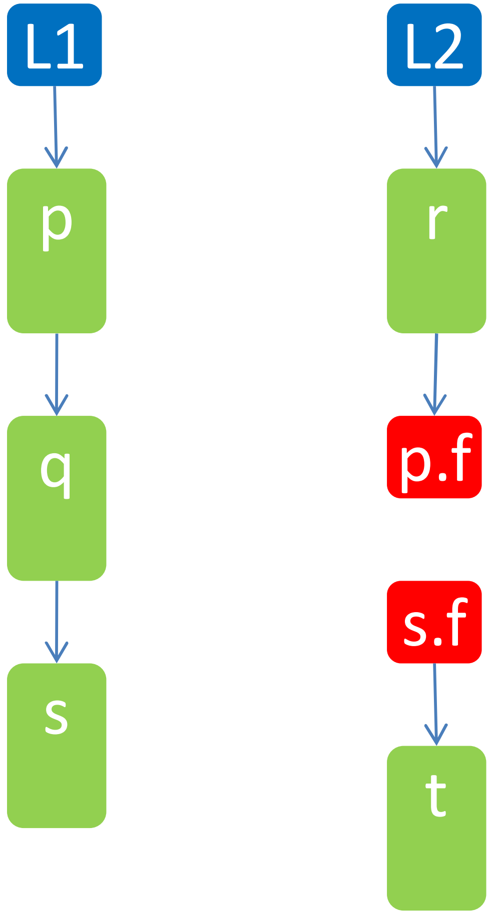
allocation	L1: x = new Object()	L1 → x	$$\frac{L1 \longrightarrow x}{\{L1\} \subseteq pt(x)}$$
assignment	x = y	y → x	$$\frac{y \longrightarrow x}{pt(y) \subseteq pt(x)}$$
store	y.f = x	x → y.f	$$\frac{x \longrightarrow y.f \qquad o \in pt(y)}{pt(x) \subseteq pt(o.f)}$$
load	x = y.f	y.f → x	$$\frac{y.f \longrightarrow x \qquad o \in pt(y)}{pt(o.f) \subseteq pt(x)}$$

Example

```
static void foo() {
L1: p = new O();
    q = p;
L2: r = new O();
    p.f = r;
    t = bar( q );
}

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

Example

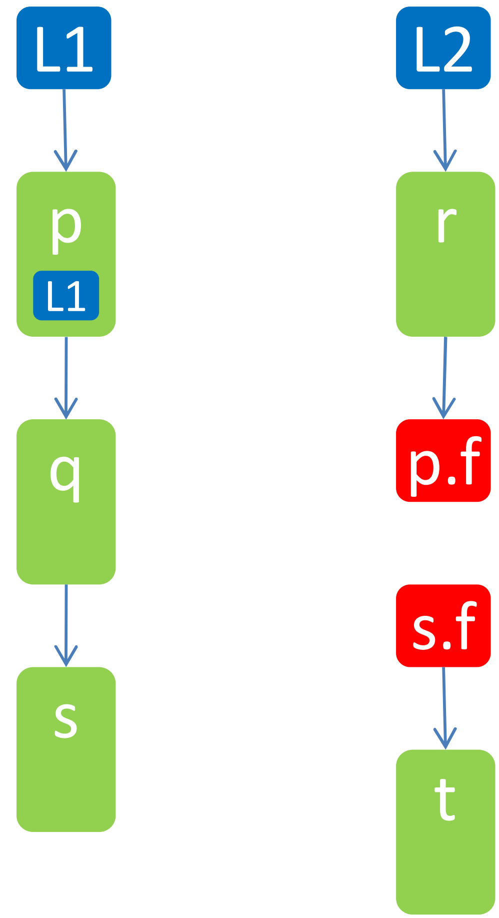
```
static void foo() {
L1: p = new O();
    q = p;
L2: r = new O();
    p.f = r;
    t = bar( q );
}

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



Generate points-to assignment graph.

Example

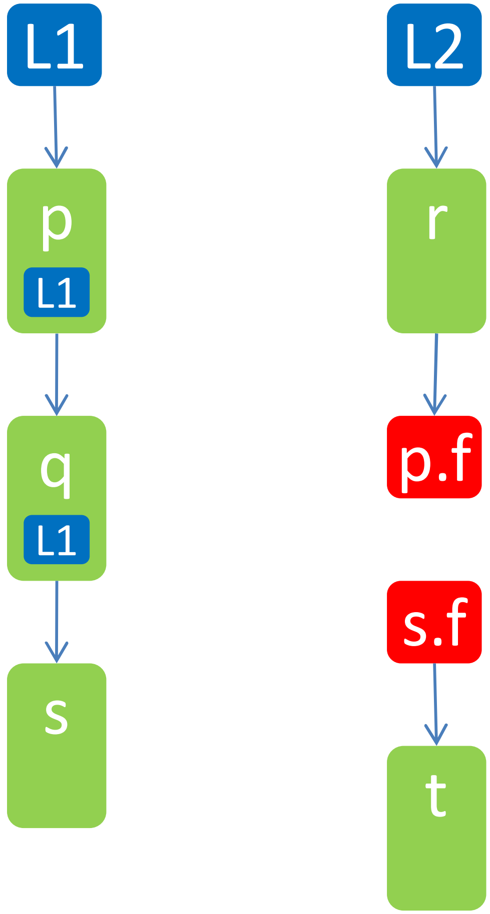
```
static void foo() {
L1: p = new O();
    q = p;
L2: r = new O();
    p.f = r;
    t = bar( q );
}

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



Propagate points-to sets.

Example

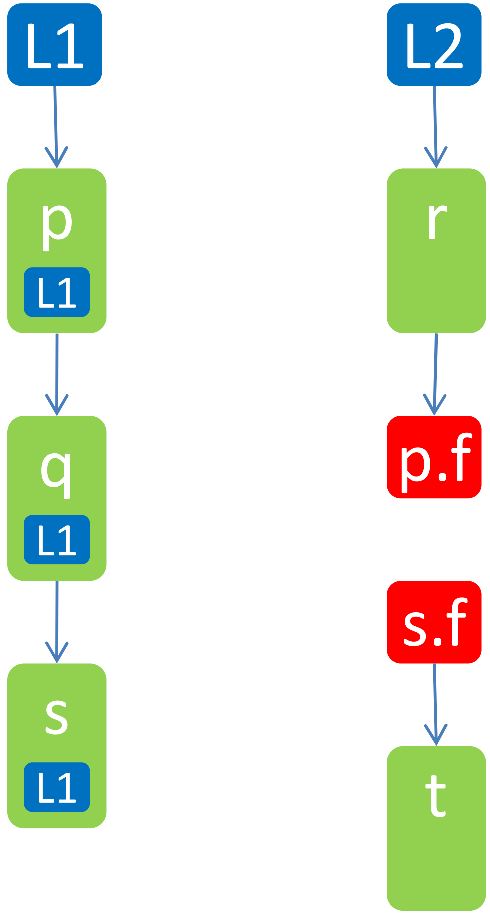
```
static void foo() {
L1: p = new O();
    q = p;
L2: r = new O();
    p.f = r;
    t = bar( q );
}

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



Propagate points-to sets.

Example

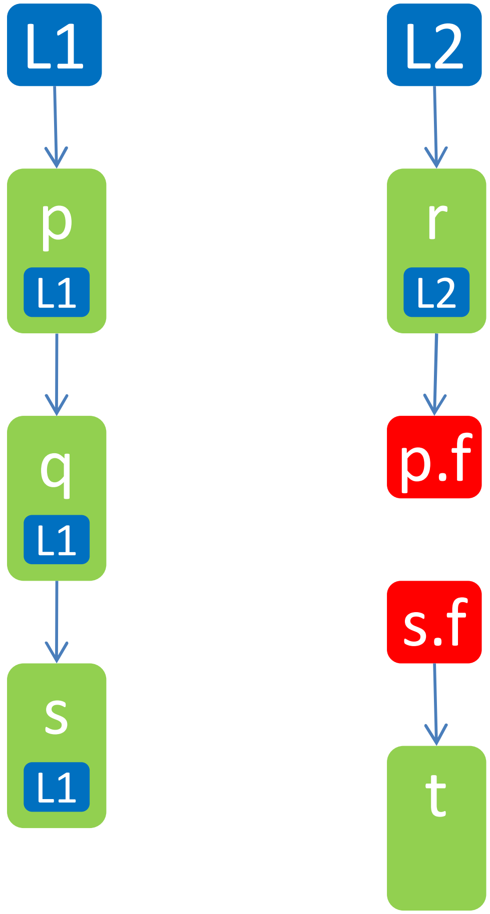
```
static void foo() {
L1: p = new O();
    q = p;
L2: r = new O();
    p.f = r;
    t = bar( q );
}

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



Propagate points-to sets.

Example

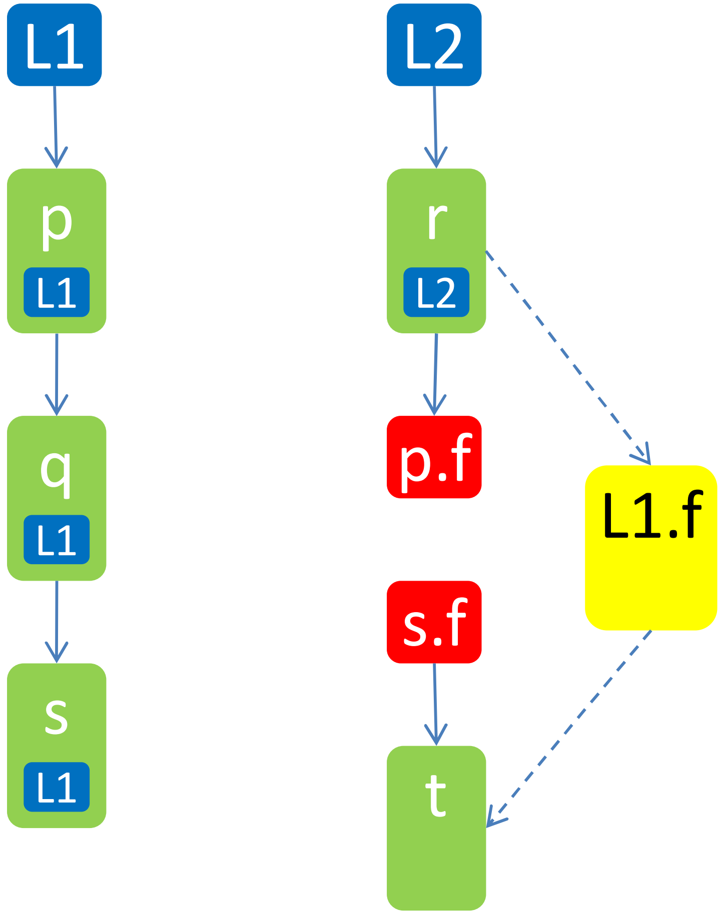
```
static void foo() {
L1: p = new O();
    q = p;
L2: r = new O();
    p.f = r;
    t = bar( q );
}

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



Propagate points-to sets.

Example

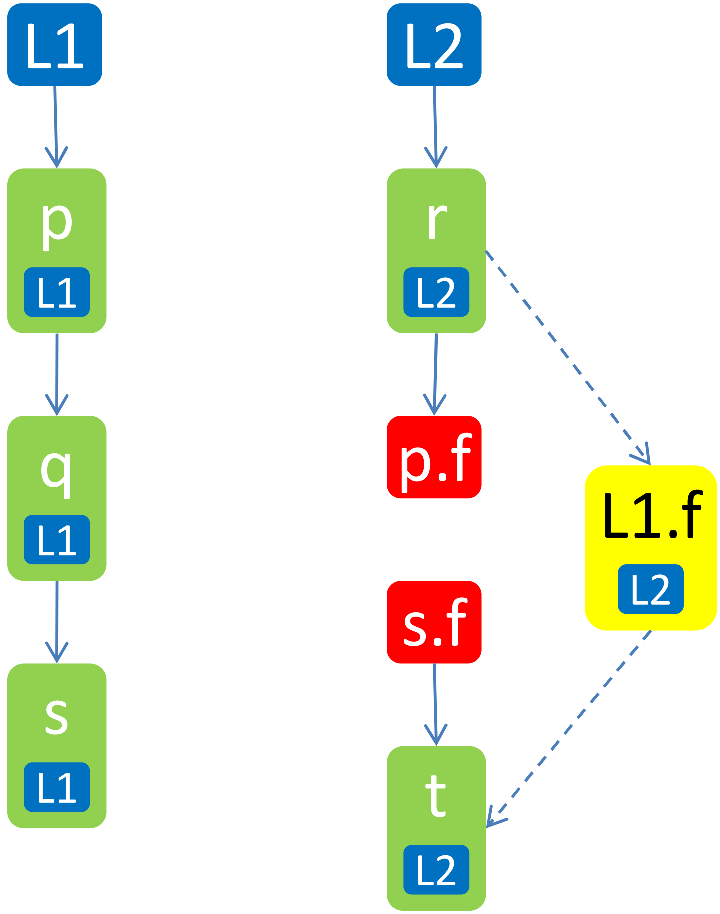
```
static void foo() {
L1: p = new O();
    q = p;
L2: r = new O();
    p.f = r;
    t = bar( q );
}

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



Add load/store edges.

Example

```
static void foo() {
L1: p = new O();
    q = p;
L2: r = new O();
    p.f = r;
    t = bar( q );
}

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



Re-propagate points-to sets.

Overall algorithm

Simple:

```
repeat until no change {
  propagate abstract objects along edges
  for each load/store, add indirect edges to heap ptr nodes
}
```

Detailed:

```
add all allocation nodes to worklist
while worklist not empty {
  remove node v1 from worklist
  for each edge v1 -> v2, propagate pt(v1) into pt(v2)
    if v2 changed, add v2 to worklist
  for each load v1.f -> v3 {
    for each a in pt(v1) {
      add edge a.f -> v3 to assignment graph
      add node a.f to worklist
    }
  }
  for each store v3 -> v1.f {
     ... (as above)
  }
}
```

Comparison with 0CFA

Field-sensitive subset-based points-to analysis:

pt : (Var $\cup$ (Obj $\times$ Field)) $\rightarrow \wp$(Obj)

0CFA:

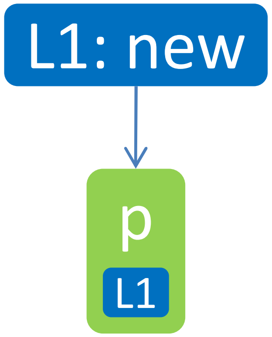
$$\hat{\varsigma} \in \hat{\Sigma} = \mathsf{Call} \times \widehat{Env}$$

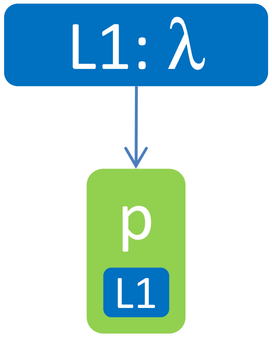
$$\hat{\rho} \in \widehat{Env} = \mathsf{Var} \rightharpoonup \mathcal{P}\left(\widehat{Clo}\right)$$

$$\widehat{clo} \in \widehat{Clo} = \mathsf{Lam}$$

$\Rightarrow$ Σ = Call $\times$ (Var $\rightarrow \wp$(Lam))

Comparison with 0CFA





Set implementation

- hash: Using java.util.HashSet
- array: Sorted array, binary search

a	b	d	g

- bit vector:

a	b	c	d	e	f	g	h	i	j
1	1	0	1	0	0	1	0	0	0

- hybrid:
  - array for small sets
  - bit vector for large sets
- sparse bit vector:

0 (ab)	1	1
1 (cd)	0	1
3 (gh)	1	0

- binary decision diagram:

Set implementation

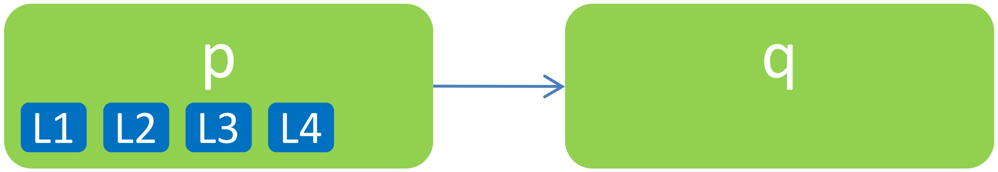
hash	slow	large
array	slow	small
bit vector	fast	large
hybrid	fast	small
sparse bit vector	fast	small
binary decision diagram	depends	depends

Slow vs. fast: up to 100x difference

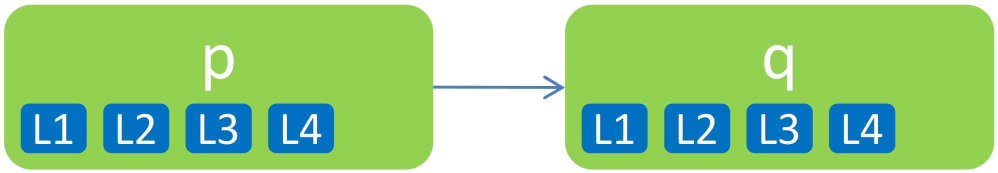
Large vs. small: up to 3x difference

Set implementation is very important.

Incremental propagation

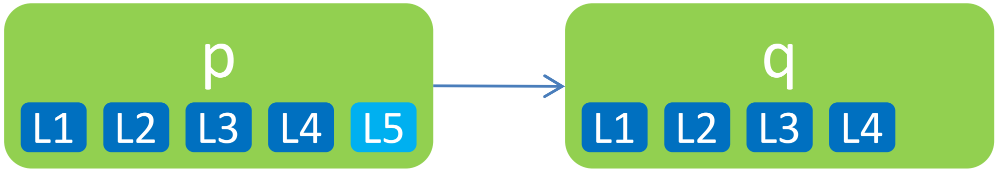


Incremental propagation



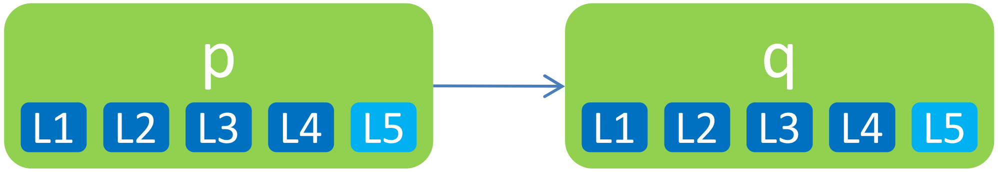
- 1st iteration: propagate {L1, L2, L3, L4}

Incremental propagation



- 1st iteration: propagate {L1, L2, L3, L4}
- add L5 to pt(p)

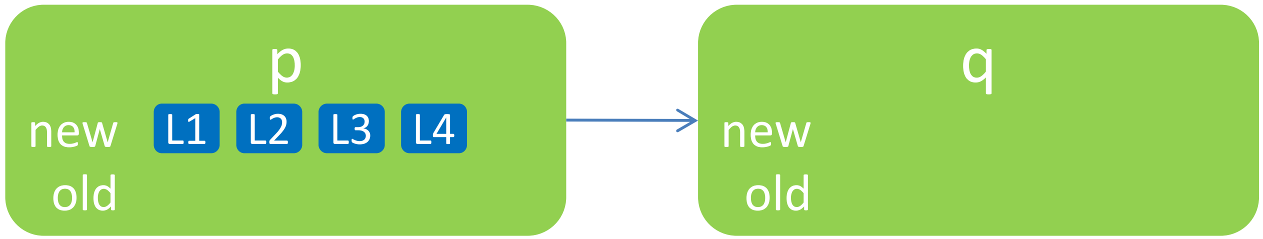
Incremental propagation



- 1st iteration: propagate {L1, L2, L3, L4}
- add L5 to pt(p)
- 2nd iteration: propagate {L1, L2, L3, L4, L5}

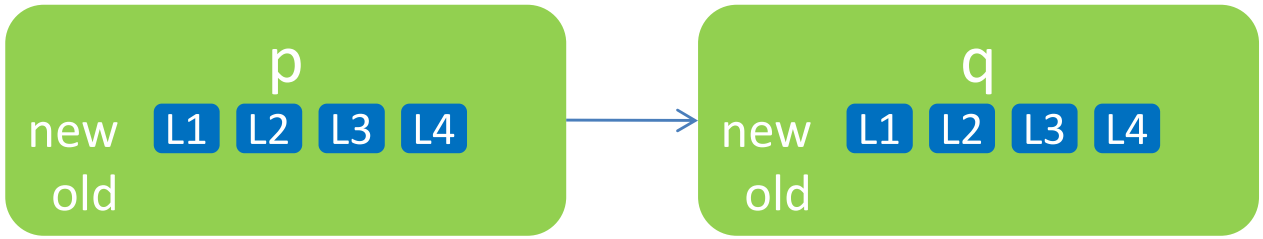
Incremental propagation

Idea: Split sets into old part and new part.



Incremental propagation

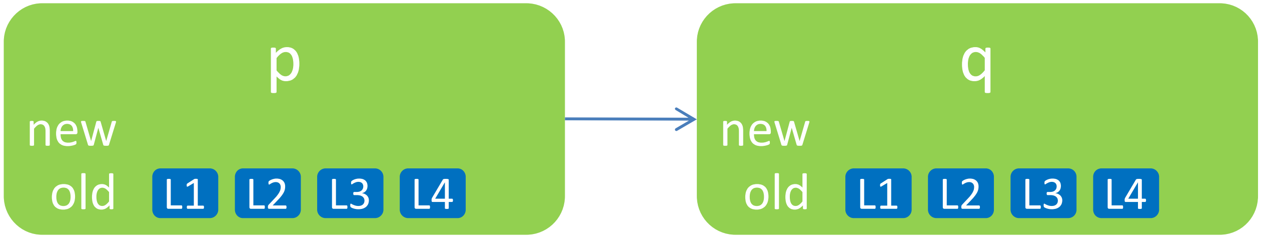
Idea: Split sets into old part and new part.



- 1st iteration: propagate {L1, L2, L3, L4}]

Incremental propagation

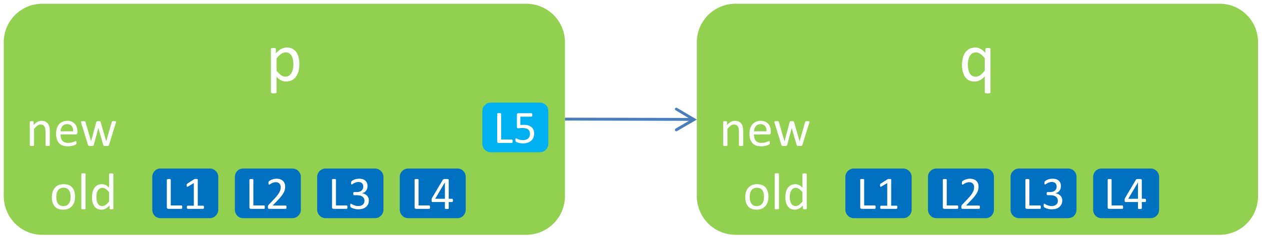
Idea: Split sets into old part and new part.



- 1st iteration: propagate {L1, L2, L3, L4}]
- flush new to old

Incremental propagation

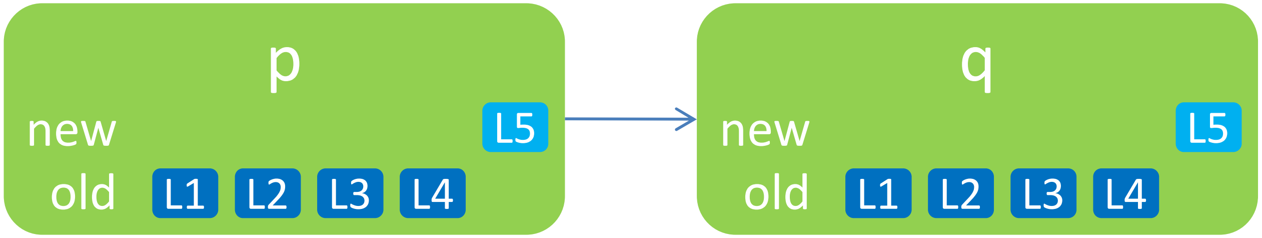
Idea: Split sets into old part and new part.



- 1st iteration: propagate {L1, L2, L3, L4}]
- flush new to old
- add L5 to new part of pt(p)

Incremental propagation

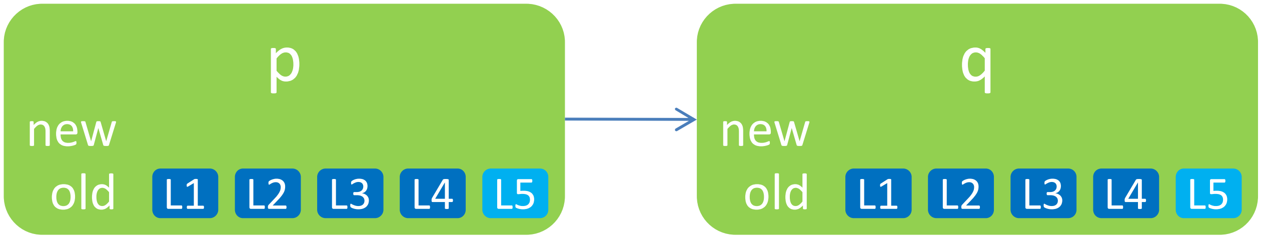
Idea: Split sets into old part and new part.



- 1st iteration: propagate {L1, L2, L3, L4}]
- flush new to old
- add L5 to new part of pt(p)
- 2nd iteration: propagate {L5}

Incremental propagation

Idea: Split sets into old part and new part.



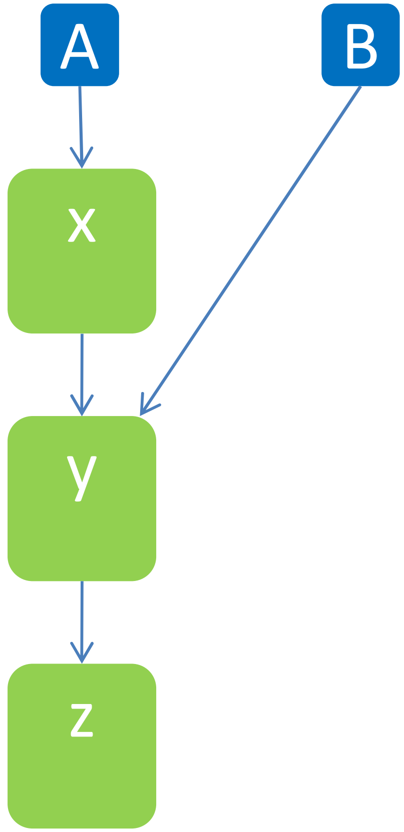
- 1st iteration: propagate {L1, L2, L3, L4}]
- flush new to old
- add L5 to new part of pt(p)
- 2nd iteration: propagate {L5}
- flush new to old

Design decisions for precision/efficiency

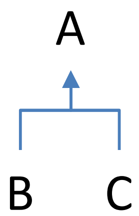
- The abstraction (affects precision and efficiency):
  - Type filtering
  - Field sensitivity
  - Directionality
  - Call graph construction
  - Context sensitivity
  - Flow sensitivity
- Algorithm and implementation (affects efficiency)
  - Propagation algorithm
  - Set implementation

Type filtering

```
   A x, z;
   B y;
A: x = new A();
B: y = new B();
   y = (B) x;
   z = y;
```



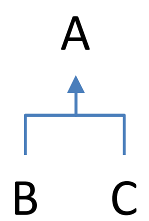
Inheritance hierarchy:

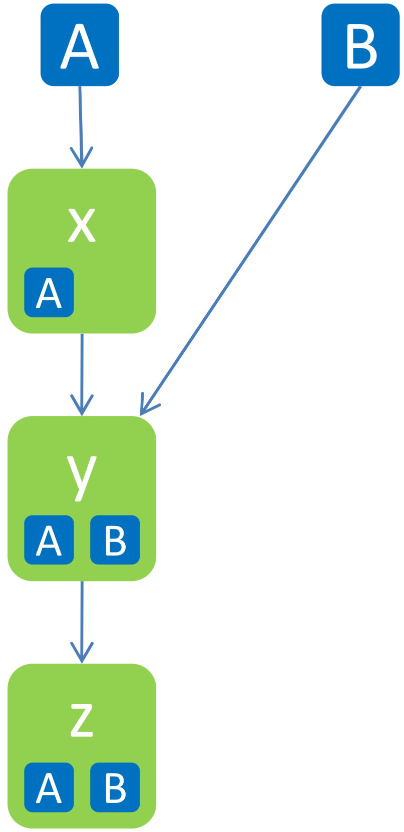


Type filtering: none

```
   A x, z;
   B y;
A: x = new A();
B: y = new B();
   y = (B) x;
   z = y;
```

Inheritance hierarchy:

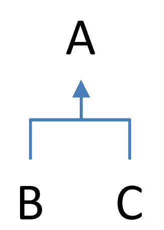


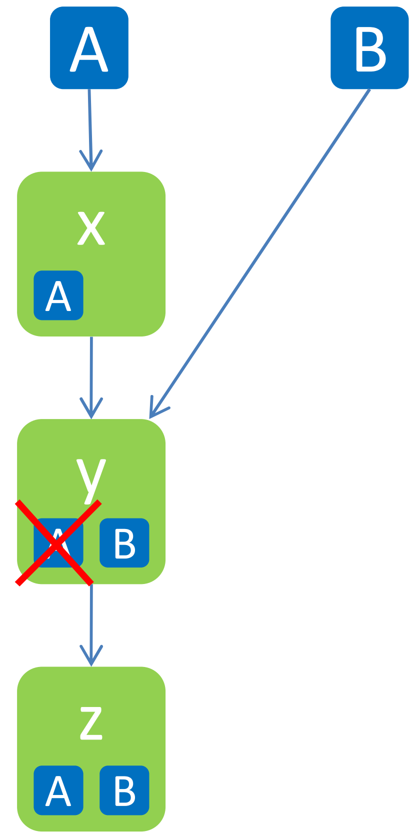


Type filtering: after analysis

```
    A x, z;
    B y;
A:  x = new A();
B:  y = new B();
    y = (B) x;
    z = y;
```

Inheritance hierarchy:

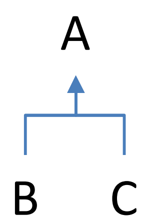


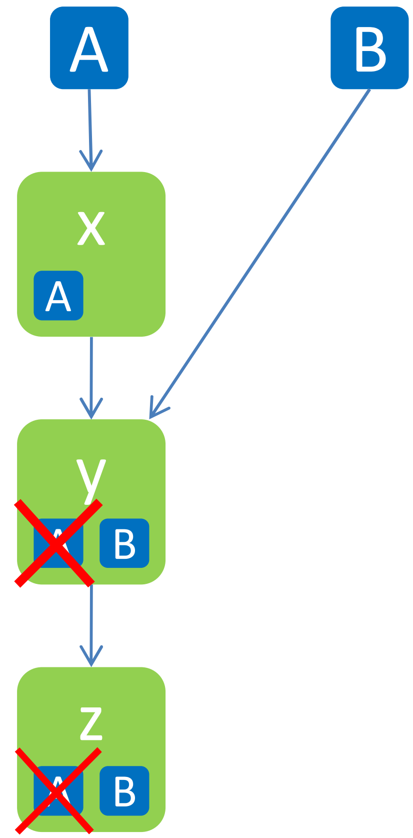


Type filtering: during analysis

```
   A x, z;
   B y;
A: x = new A();
B: y = new B();
   y = (B) x;
   z = y;
```

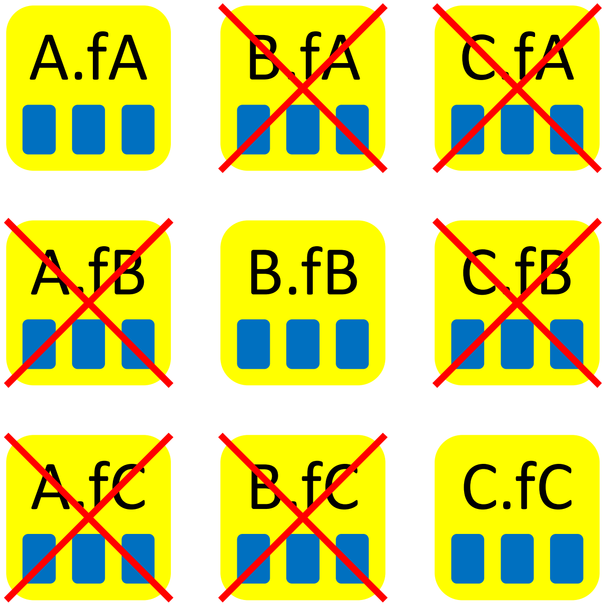
Inheritance hierarchy:





Type filtering: effect on heap nodes

```
class A {
  Object fA;
}
class B {
  Object fB;
}
class C {
  Object fC;
}
A: a = new A();
B: b = new B();
C: c = new C();
```



Type filtering

- Ignoring types yields many large points-to sets.
- Filtering after propagation is almost as precise as during propagation.
- Filtering during propagation is both most precise and most efficient.

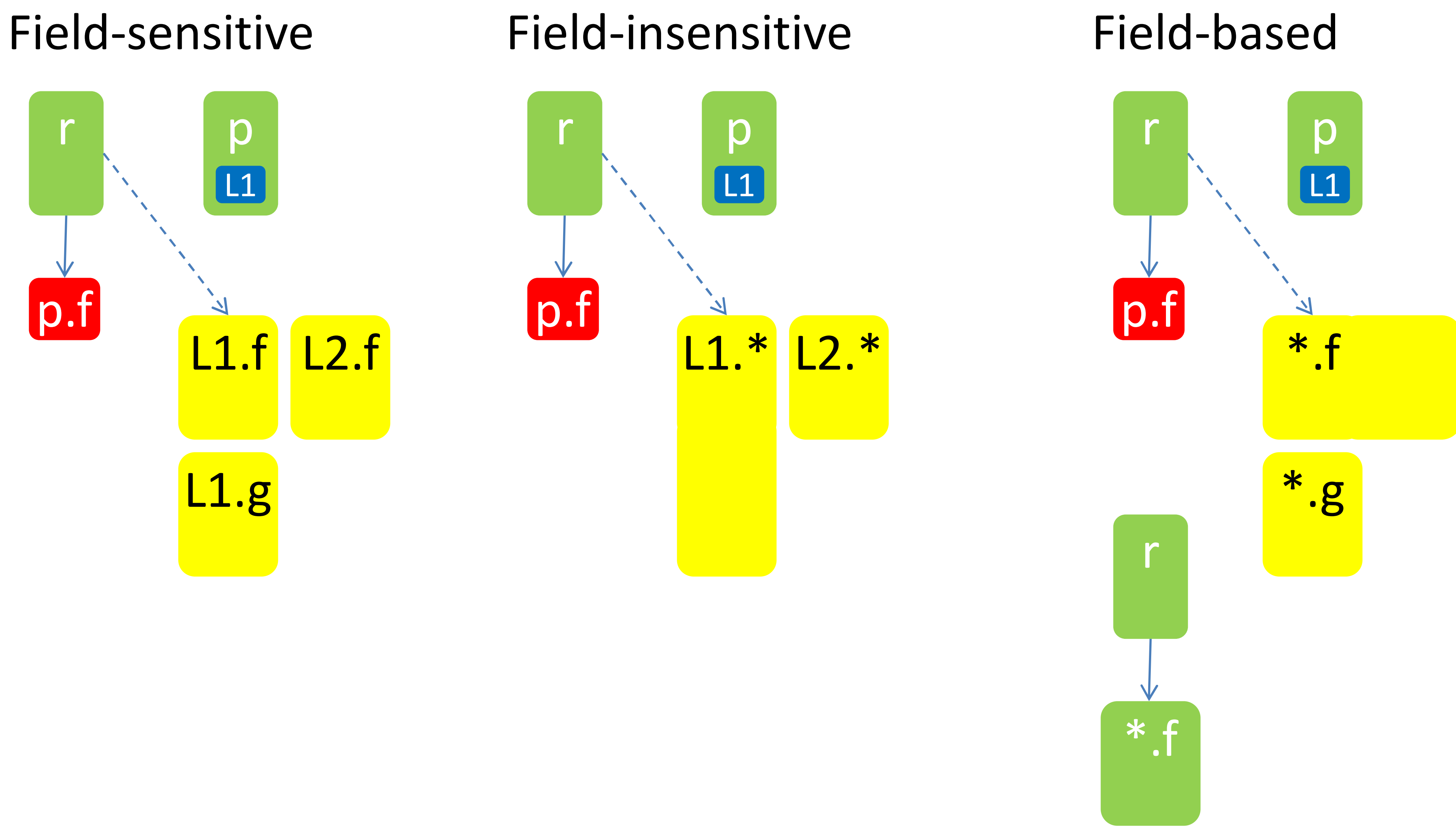
ignore	slow	imprecise
after propagation	slow	precise
during propagation	fast	precise

Design decisions for precision/efficiency

- The abstraction (affects precision and efficiency):
  - –Type filtering
  - –Field sensitivity
  - –Directionality
  - –Call graph construction
  - –Context sensitivity
  - –Flow sensitivity
- Algorithm and implementation (affects efficiency)
  - –Propagation algorithm
  - –Set implementation

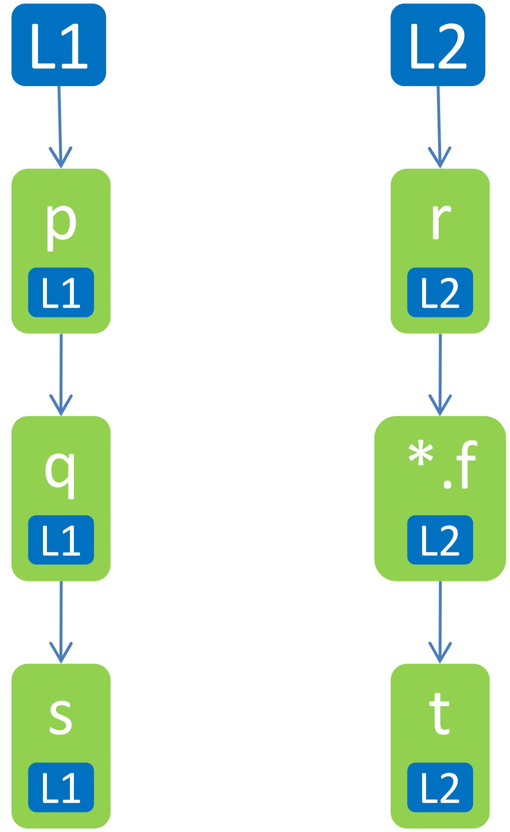
Field reference representation

Idea: merge yellow nodes with same abstract object (resp. same field).



Example (field-based)

```
static void foo() {
L1: p = new O();
    q = p;
L2: r = new O();
    p.f = r;
    t = bar( q );
}

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



Overall (field-based) algorithm

```
merge each SCC in assignment graph into a single node
topologically sort resulting DAG
for each node v1 in topological order {
  for each edge v1 -> v2 {
    propagate pt(v1) into pt(v2)
  }
}
```

Each edge is processed only once.

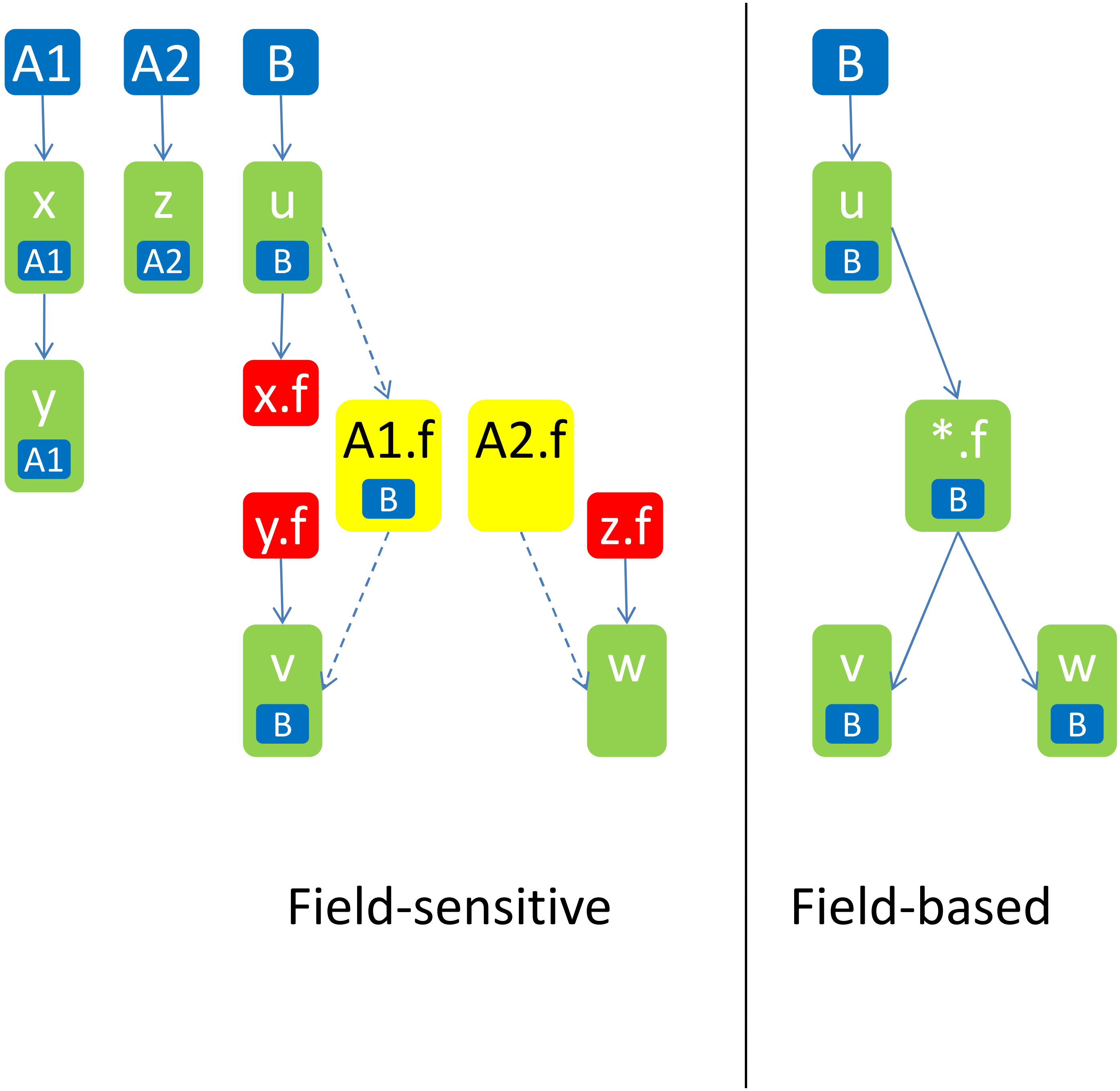
Worst-case O(n^2).

Also, worst-case is linear in size of output.

In contrast, field-sensitive algorithm is O(n^3).

Example of precision loss

```
     A x, y, z;
     B u, v, w;
A1:  x = new A();
     y = x;
A2:  z = new A();
B:   u = new B();
     x.f = u;
     v = y.f;
     w = z.f;
```



Field sensitivity summary

	Java	C
field-insensitive	sound slow imprecise	sound slow imprecise
field-based	sound fast imprecise	unsound
field-sensitive	sound slowest precise	sound slowest precise

Comparison: field-based PTA vs. 0CFA

Field-based subset-based points-to analysis:

pt : $\text{Var} \rightarrow \wp(\text{Obj})$

0CFA:

$$\hat{\varsigma} \in \hat{\Sigma} = \text{Call} \times \widehat{Env}$$

$$\hat{\rho} \in \widehat{Env} = \text{Var} \rightharpoonup \mathcal{P}\left(\widehat{Clo}\right)$$

$$\widehat{clo} \in \widehat{Clo} = \text{Lam}$$

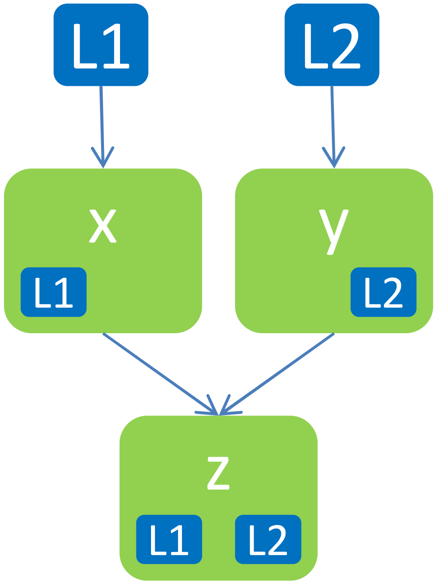
$\Rightarrow$ $\Sigma = \text{Call} \times (\text{Var} \rightarrow \wp(\text{Lam}))$

Design decisions for precision/efficiency

- The abstraction (affects precision and efficiency):
  - –Type filtering
  - –Field sensitivity
  - –Directionality
  - –Call graph construction
  - –Context sensitivity
  - –Flow sensitivity
- Algorithm and implementation (affects efficiency)
  - –Propagation algorithm
  - –Set implementation

Directionality

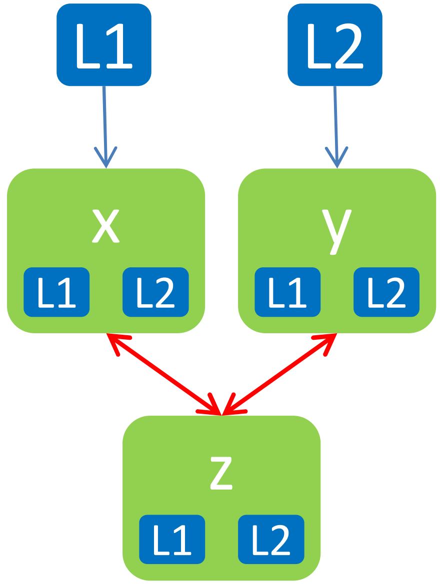
```
    Object x, y, z;
L1: x = new Object();
L2: y = new Object();
    if(*) {
      z = x;
    } else {
      z = y;
    }
```



Subset-based analysis
aka Inclusion-based analysis
aka Andersen's analysis

Directionality

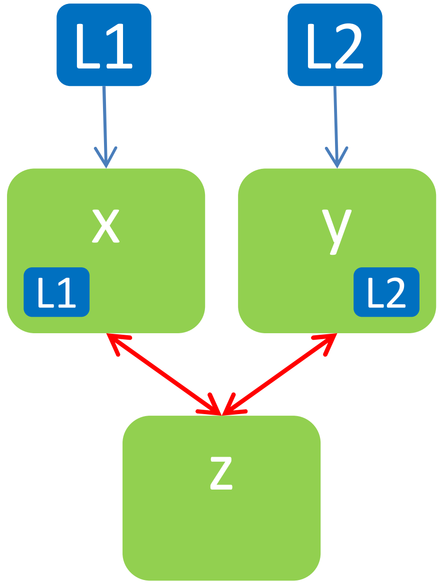
```
    Object x, y, z;
L1: x = new Object();
L2: y = new Object();
    if(*) {
      z = x;
    } else {
      z = y;
    }
```



Equality-based analysis
aka Unification-based analysis
aka Steensgaard's analysis

Implementation of unification

```
    Object x, y, z;
L1: x = new Object();
L2: y = new Object();
    if(*) {
      z = x;
    } else {
      z = y;
    }
```

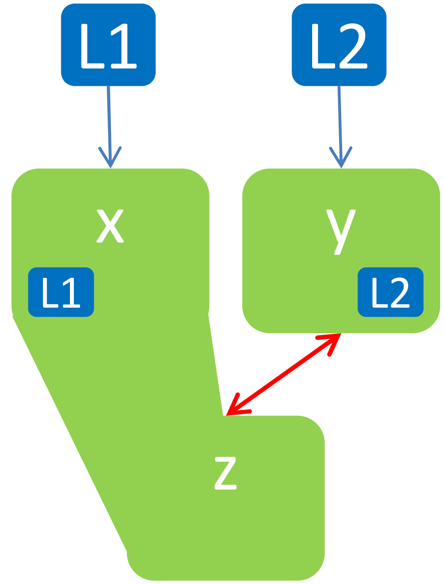


Step 1: Process allocation edges

Equality-based analysis
aka Unification-based analysis
aka Steensgaard's analysis

Implementation of unification

```
    Object x, y, z;
L1: x = new Object();
L2: y = new Object();
    if(*) {
      z = x;
    } else {
      z = y;
    }
```



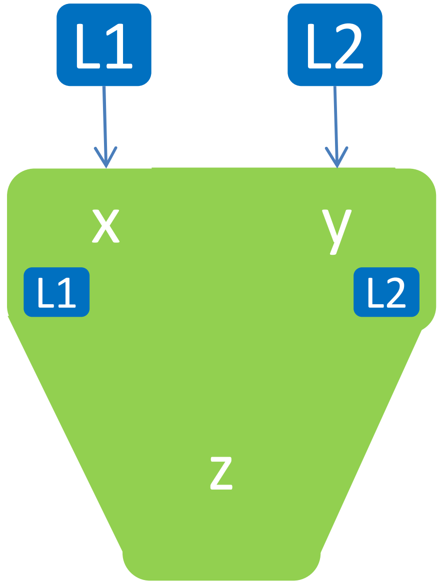
Equality-based analysis
aka Unification-based analysis
aka Steensgaard's analysis

Step 1: Process allocation edges
Step 2: Repeatedly unify nodes connected by assignments

Running time: almost linear

Implementation of unification

```
    Object x, y, z;
L1: x = new Object();
L2: y = new Object();
    if(*) {
      z = x;
    } else {
      z = y;
    }
```

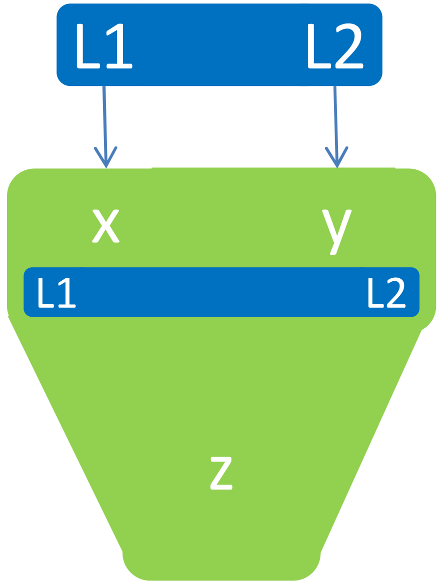


Equality-based analysis
aka Unification-based analysis
aka Steensgaard's analysis

Step 1: Process allocation edges
Step 2: Repeatedly unify nodes connected by assignments.

Implementation of unification

```
    Object x, y, z;
L1: x = new Object();
L2: y = new Object();
    if(*) {
      z = x;
    } else {
      z = y;
    }
```



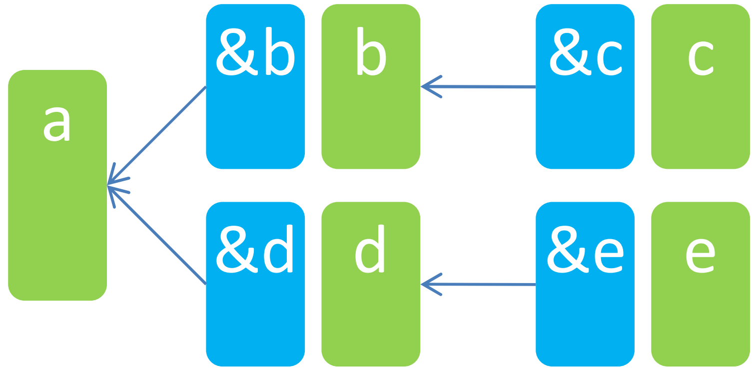
Equality-based analysis
aka Unification-based analysis
aka Steensgaard's analysis

Step 1: Process allocation edges
Step 2: Repeatedly unify nodes connected by assignments.
Also unify nodes pointed-to by same node.

Running time: almost linear

A C example

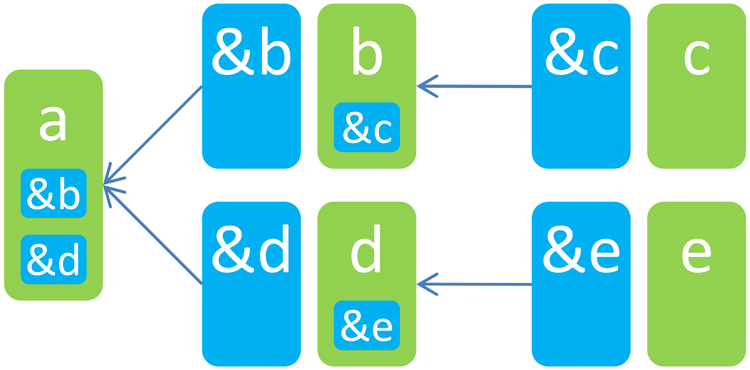
```
a = &b;
b = &c;
a = &d;
d = &e;
```



Subset-based analysis

A C example

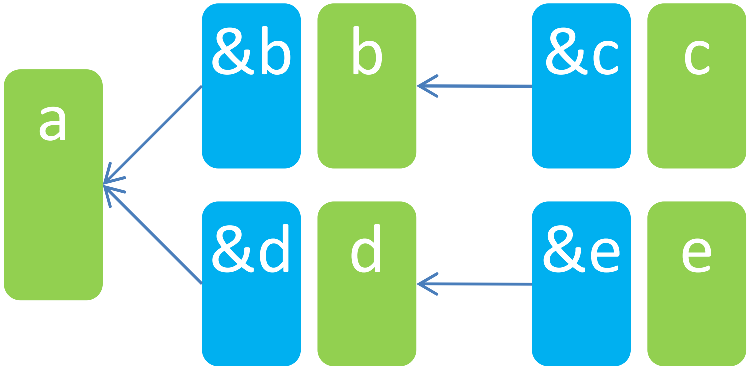
```
a = &b;
b = &c;
a = &d;
d = &e;
```



Subset-based analysis

A C example

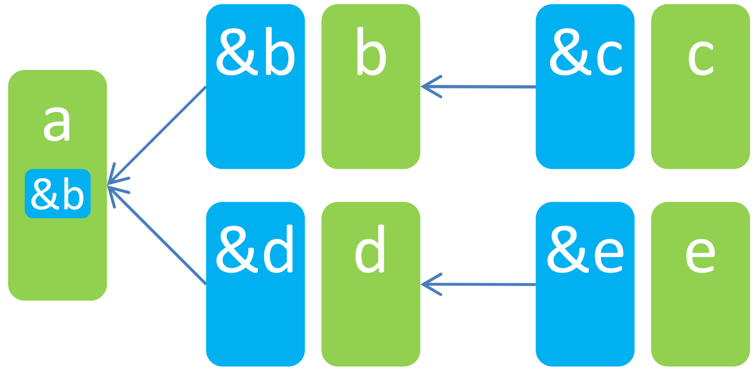
```
a = &b;
b = &c;
a = &d;
d = &e;
```



Equality-based analysis

A C example

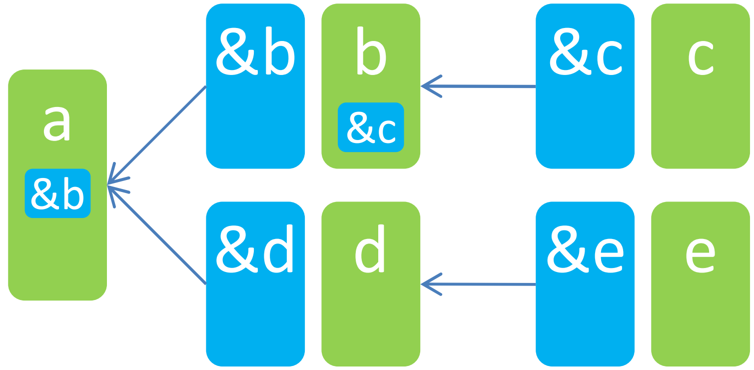
```
a = &b;
b = &c;
a = &d;
d = &e;
```



Equality-based analysis

A C example

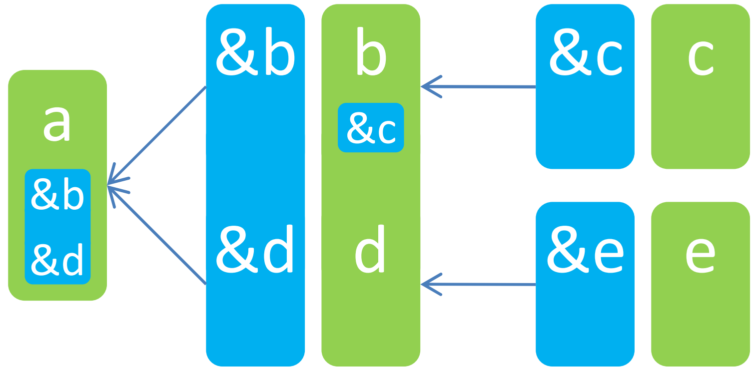
```
a = &b;
b = &c;
a = &d;
d = &e;
```



Equality-based analysis

A C example

```
a = &b;
b = &c;
a = &d;
d = &e;
```

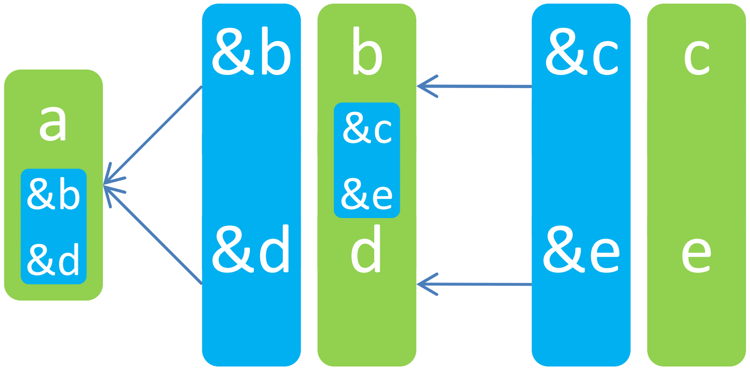


Equality-based analysis

Invariant: each node points to at most one other node.

A C example

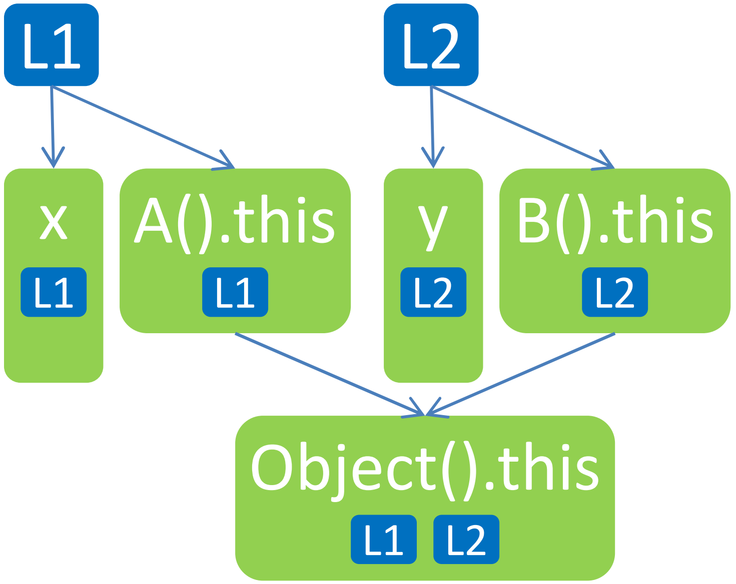
```
a = &b;
b = &c;
a = &d;
d = &e;
```



Equality-based analysis

A problem with unification and OOP

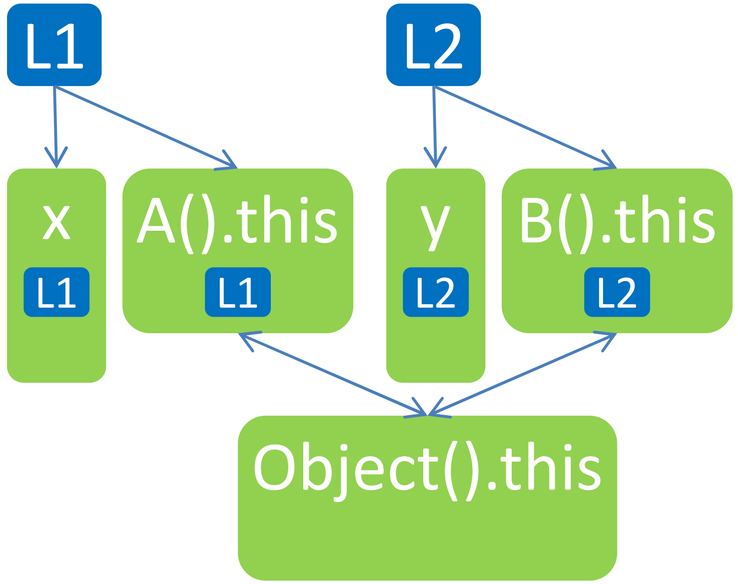
```
class A extends Object {
  public A() {
    super();
  }
}
class B extends Object {
  ...
}
L1: x = new A();
L2: y = new B();
```



Subset-based analysis

A problem with unification and OOP

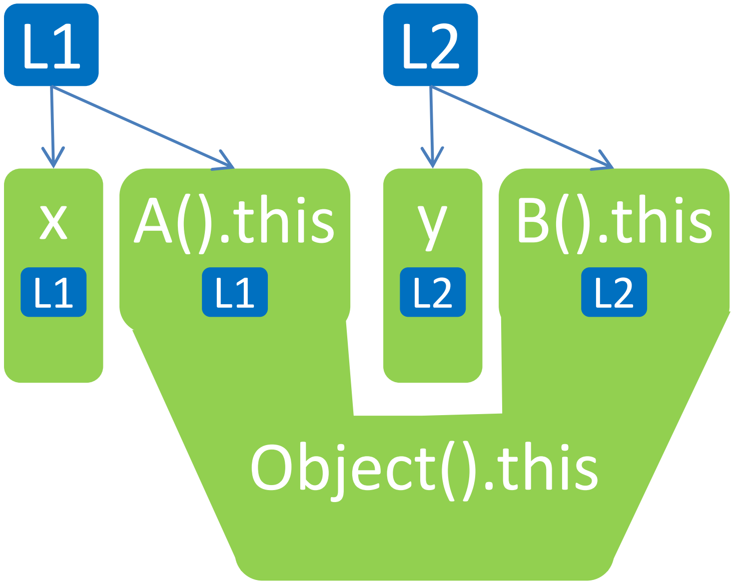
```
class A extends Object {
  public A() {
    super();
  }
}
class B extends Object {
  ...
}
L1: x = new A();
L2: y = new B();
```



Equality-based analysis

A problem with unification and OOP

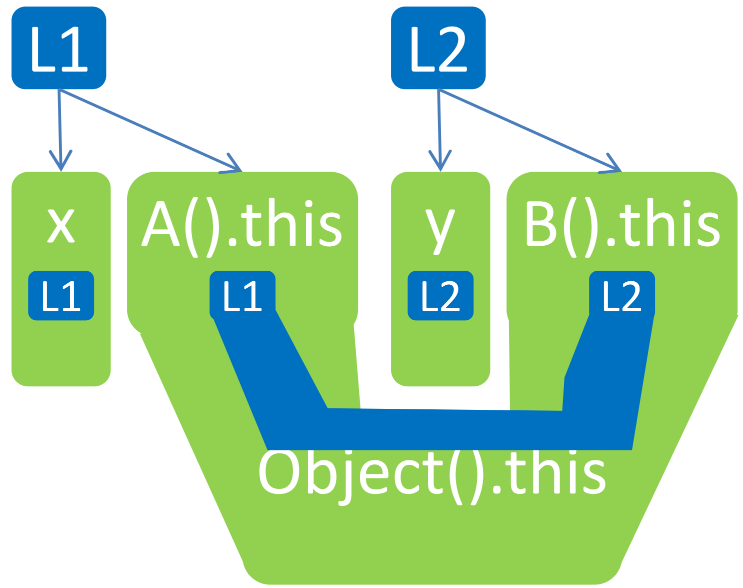
```
class A extends Object {
  public A() {
    super();
  }
}
class B extends Object {
  ...
}
L1: x = new A();
L2: y = new B();
```



Equality-based analysis

A problem with unification and OOP

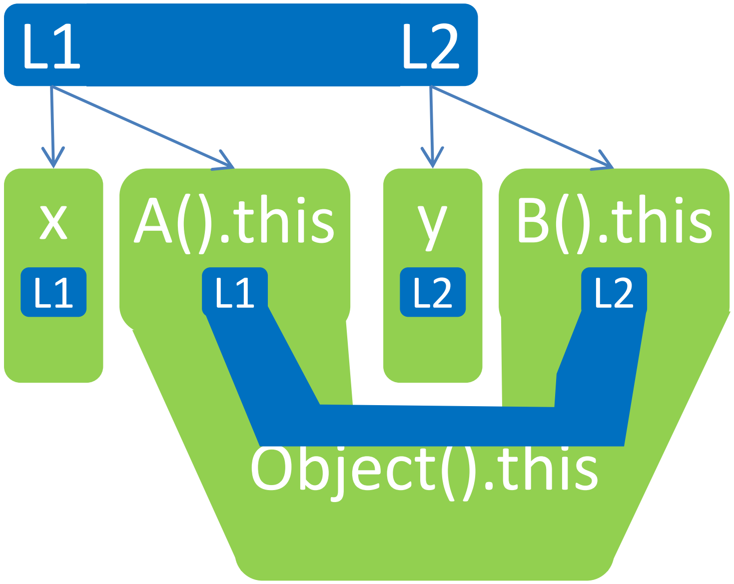
```
class A extends Object {
  public A() {
    super();
  }
}
class B extends Object {
  ...
}
L1: x = new A();
L2: y = new B();
```



Equality-based analysis

A problem with unification and OOP

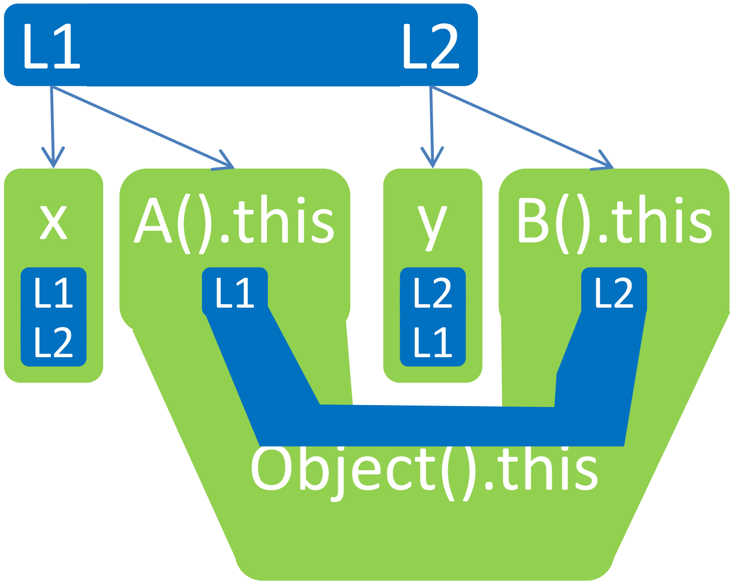
```
class A extends Object {
  public A() {
    super();
  }
}
class B extends Object {
  ...
}
L1: x = new A();
L2: y = new B();
```



Equality-based analysis

A problem with unification and OOP

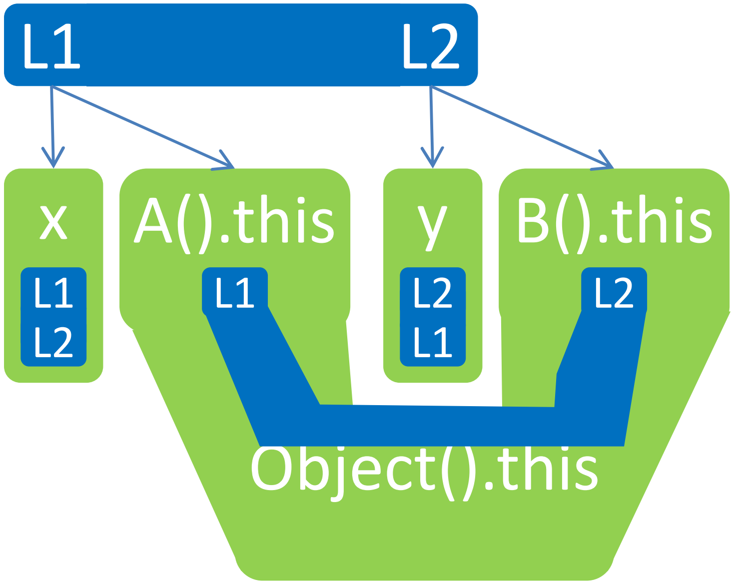
```
class A extends Object {
  public A() {
    super();
  }
}
class B extends Object {
  ...
}
L1: x = new A();
L2: y = new B();
```



Equality-based analysis

A problem with unification and OOP

```
class A extends Object {
  public A() {
    super();
  }
}
class B extends Object {
  ...
}
L1: x = new A();
L2: y = new B();
```



Equality-based analysis

Every pointer points to every object!

... but context sensitivity will fix this.

Design decisions for precision/efficiency

- The abstraction (affects precision and efficiency):
  - Type filtering
  - Field sensitivity
  - Directionality
  - Call graph construction
  - Context sensitivity
  - Flow sensitivity
- Algorithm and implementation (affects efficiency)
  - Propagation algorithm
  - Set implementation

How big is the call graph?

```
public class Hello {
  public static final void main(String[] args) {
    System.out.println("Hello");
  }
}
```

- Number of methods actually executed: ??
- Number of methods in static call graph: ??

How big is the call graph?

```
public class Hello {
  public static final void main(String[] args) {
    System.out.println("Hello");
  }
}
```

- Number of methods actually executed: 498
- Number of methods in static call graph: ??

How big is the call graph?

```
public class Hello {
  public static final void main(String[] args) {
    System.out.println("Hello");
  }
}
```

- Number of methods actually executed: 498
- Number of methods in static call graph: 3204

Call graph construction

To determine ...	... we need to know
points-to sets	
pointer assignment edges	
reachable methods	
call graph edges	

Call graph construction

To determine ...	... we need to know
points-to sets	pointer assignment edges
pointer assignment edges	
reachable methods	
call graph edges	

Call graph construction

To determine ...	... we need to know
points-to sets	pointer assignment edges
pointer assignment edges	reachable methods call graph edges
reachable methods	
call graph edges	

Call graph construction

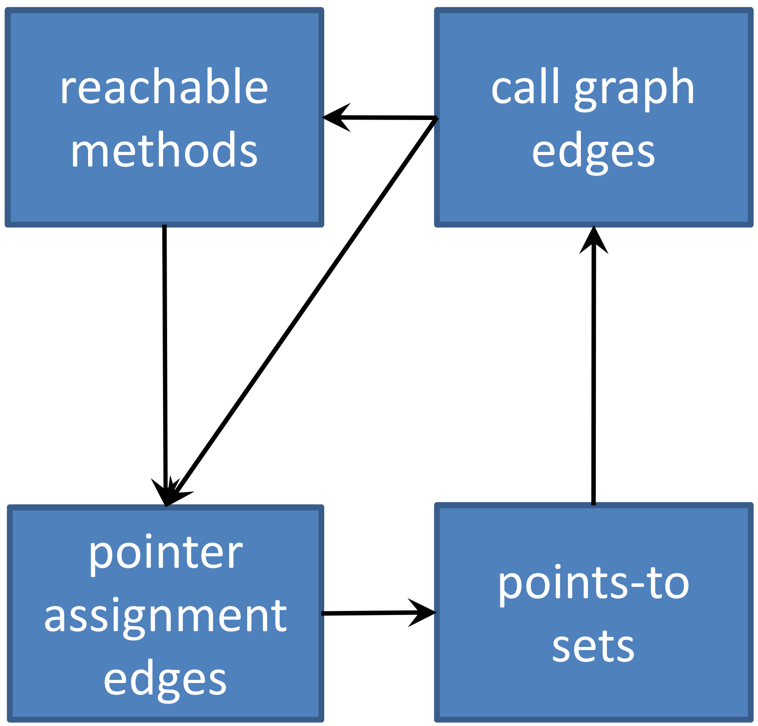
To determine ...	... we need to know
points-to sets	pointer assignment edges
pointer assignment edges	reachable methods call graph edges
reachable methods	call graph edges
call graph edges	

Call graph construction

To determine ...	... we need to know
points-to sets	pointer assignment edges
pointer assignment edges	reachable methods call graph edges
reachable methods	call graph edges
call graph edges	points-to sets

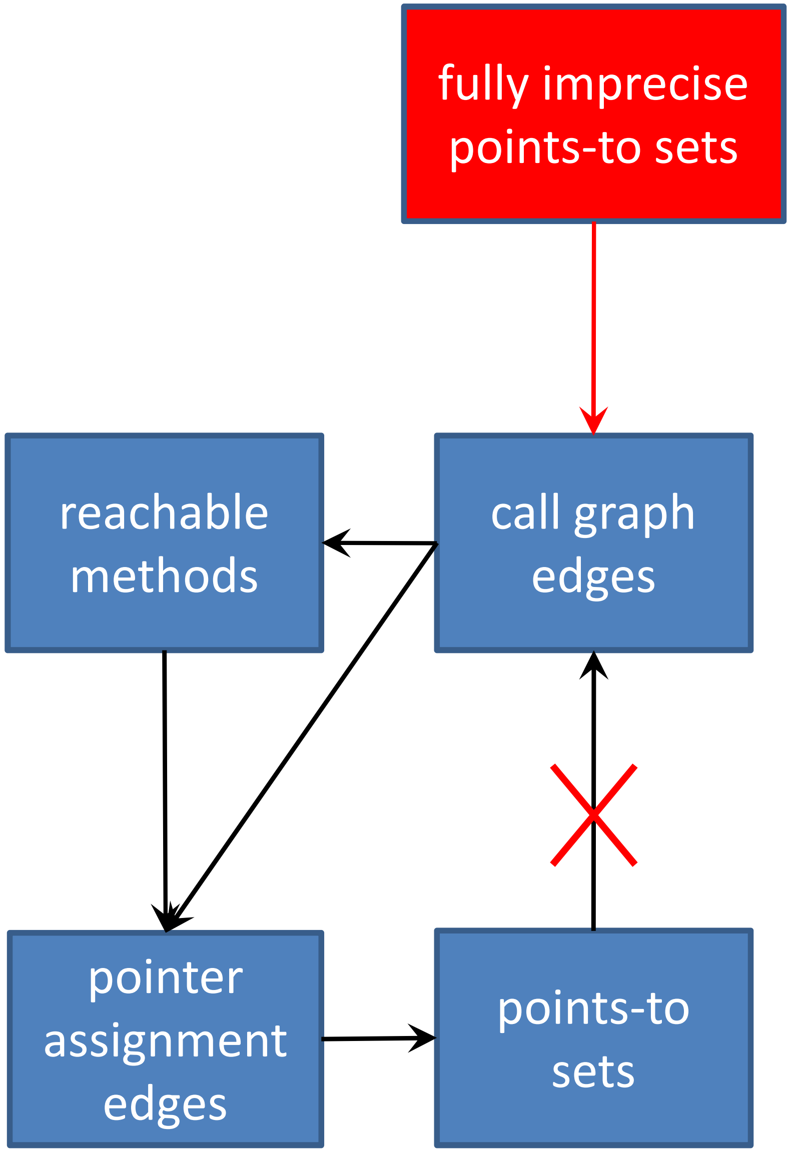
Call graph construction

To determine ...	... we need to know
points-to sets	pointer assignment edges
pointer assignment edges	reachable methods call graph edges
reachable methods	call graph edges
call graph edges	points-to sets



Ahead of time call graph construction

1. Assume every pointer can point to any object compatible with its declared type.
2. Explore call graph using this assumption, listing reachable methods (Class Hierarchy Analysis).
3. Generate pointer assignment graph using resulting call edges and reachable methods.
4. Propagate points-to sets along pointer assignment graph.



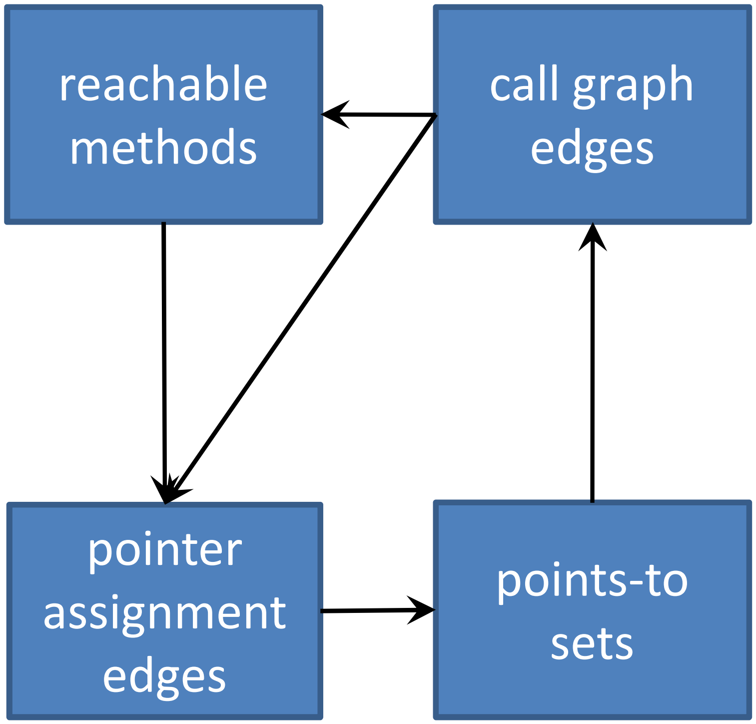
Ahead of time call graph construction

1. Assume every pointer can point to any object compatible with its declared type.
2. Explore call graph using this assumption, listing reachable methods (Class Hierarchy Analysis).
3. Generate pointer assignment graph using resulting call edges and reachable methods.
4. Propagate points-to sets along pointer assignment graph.

- no iteration
- very imprecise due to many reachable methods

On-the-fly call graph construction

1. Start with only initial reachable methods, no call edges, no pointer assignment edges, and no points-to sets.
2. Iteratively generate pointer assignment edges, points-to sets, call edges, and reachable methods implied by current information.
3. Stop when overall fixed point is reached.



On-the-fly call graph construction

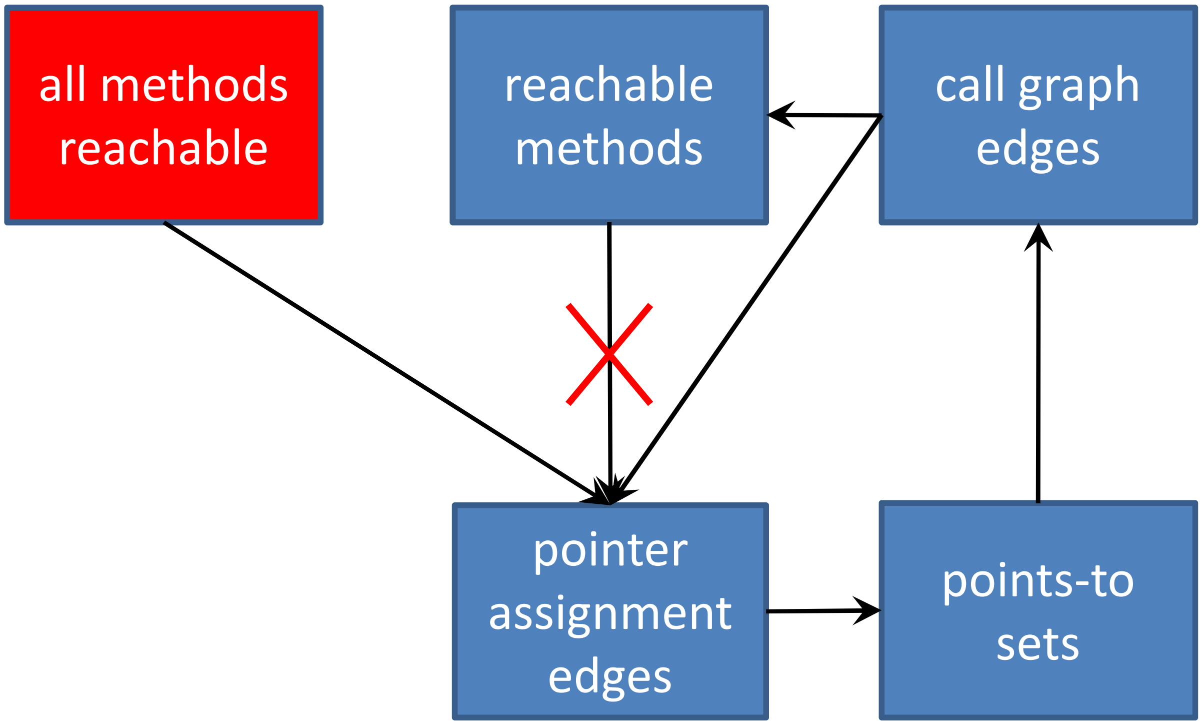
1. Start with only initial reachable methods, no call edges, no pointer assignment edges, and no points-to sets.
2. Iteratively generate pointer assignment edges, points-to sets, call edges, and reachable methods implied by current information.
3. Stop when overall fixed point is reached.

- requires iteration
- slower
- more complicated
- much more precise

due to fewer reachable methods

Partly on-the-fly call graph construction

1. Assume all methods are reachable.
2. Generate pointer assignment edges for all methods.
3. Iteratively propagate points-to sets, add call edges, and generate pointer assignment edges for new call edges.



Partly on-the-fly call graph construction

1. Assume all methods are reachable.
2. Generate pointer assignment edges for all methods.
3. Iteratively propagate points-to sets, add call edges, and generate pointer assignment edges for new call edges.

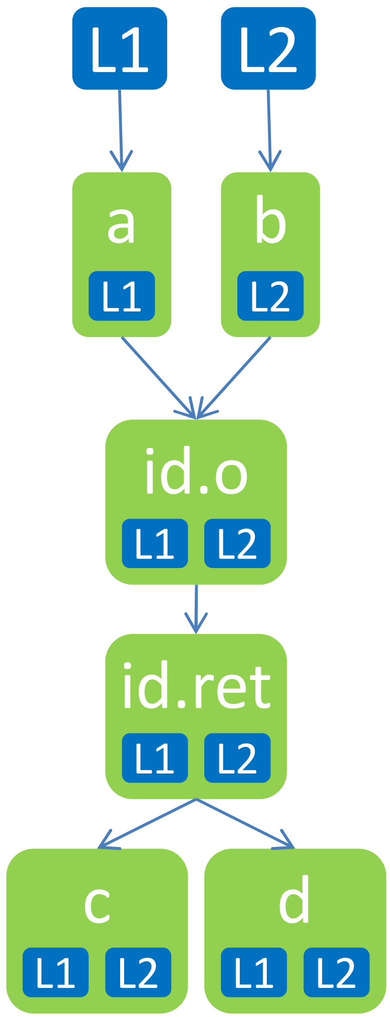
- requires iteration
- speed is in between ahead-of-time and on-the-fly
- complexity is in between...
- still imprecise due to many reachable methods

Design decisions for precision/efficiency

- The abstraction (affects precision and efficiency):
  - Type filtering
  - Field sensitivity
  - Directionality
  - Call graph construction
  - Context sensitivity
  - Flow sensitivity
- Algorithm and implementation (affects efficiency)
  - Propagation algorithm
  - Set implementation

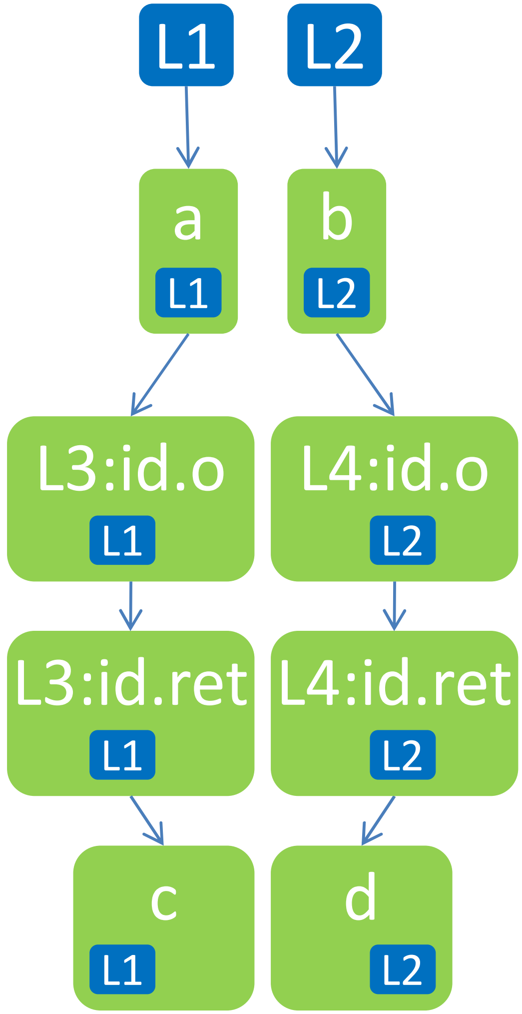
Context sensitivity motivation

```
Object id(Object o) {
   return o;
}

void f() {
L1: Object a = new Object();
L2: Object b = new Object();
    Object c = id(a);
    Object d = id(b);
}
```



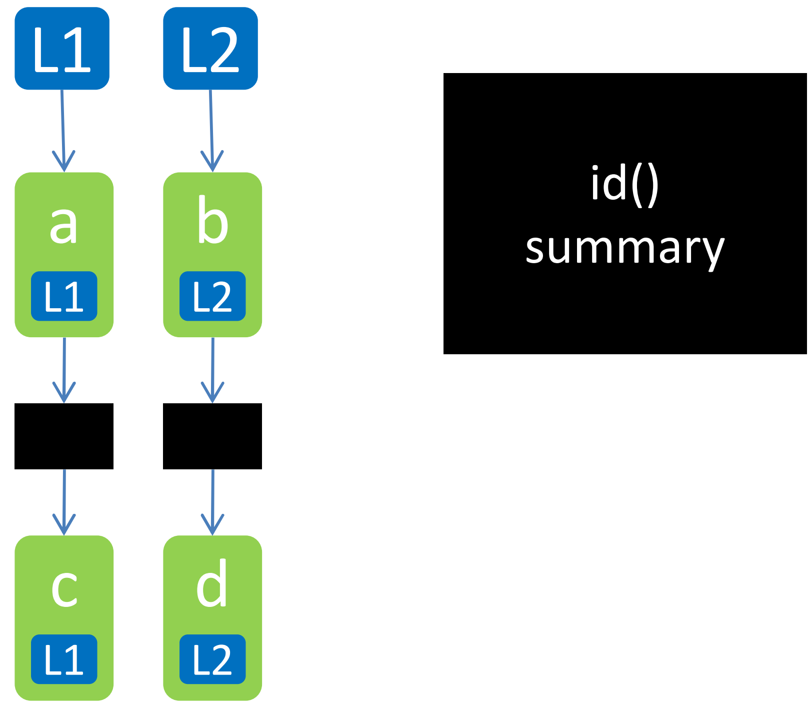
Call strings approach (aka cloning)

```
Object id(Object o) {
   return o;
}

void f() {
L1: Object a = new Object();
L2: Object b = new Object();
L3: Object c = id(a);
L4: Object d = id(b);
}
```



Summary-based approach

```
Object id(Object o) {
   return o;
}

void f() {
L1: Object a = new Object();
L2: Object b = new Object();
    Object c = id(a);
    Object d = id(b);
}
```



Challenge: how to design a summary that

- precisely models all effects of id() (and its transitive callees)
- is cheap to compute and represent
- is cheap to instantiate

Comparison with 1CFA

Field-sensitive subset-based 1-call-site-sensitive points-to analysis:

$\text{pt} : (\text{Call} \times \text{Var} \cup (\text{Obj} \times \text{Field})) \rightarrow \wp(\text{Obj})$

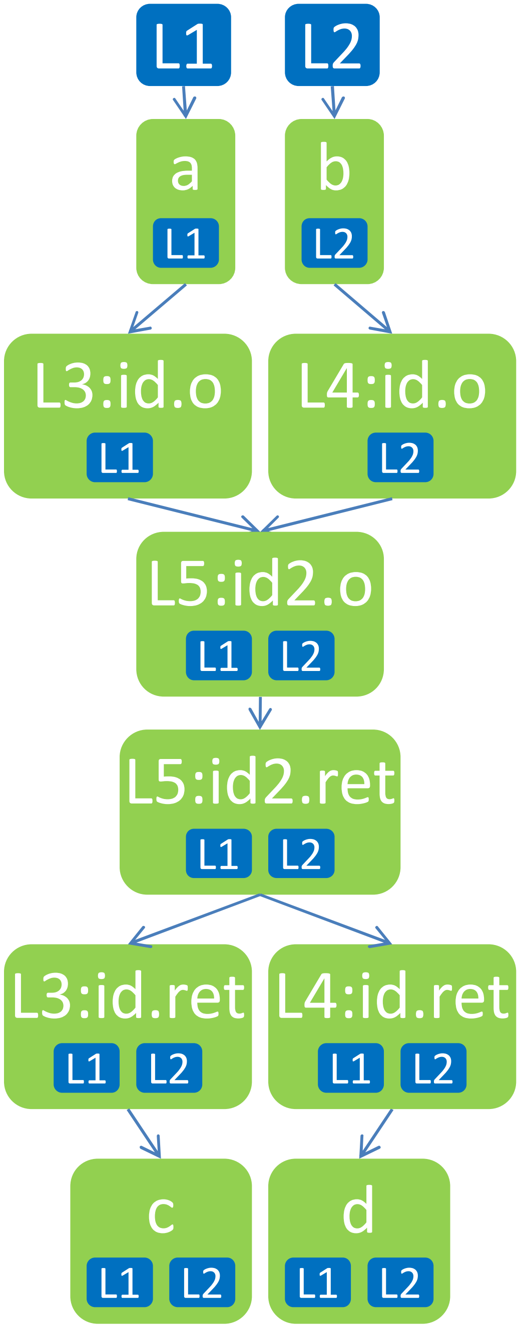
1CFA:

$\Sigma =$

$\text{Call} \times (\text{Var} \rightarrow \text{Addr}) \times (\text{Addr} \rightarrow \wp(\text{Lam} \times \text{Var} \rightarrow \text{Addr})) \times \text{Call}$

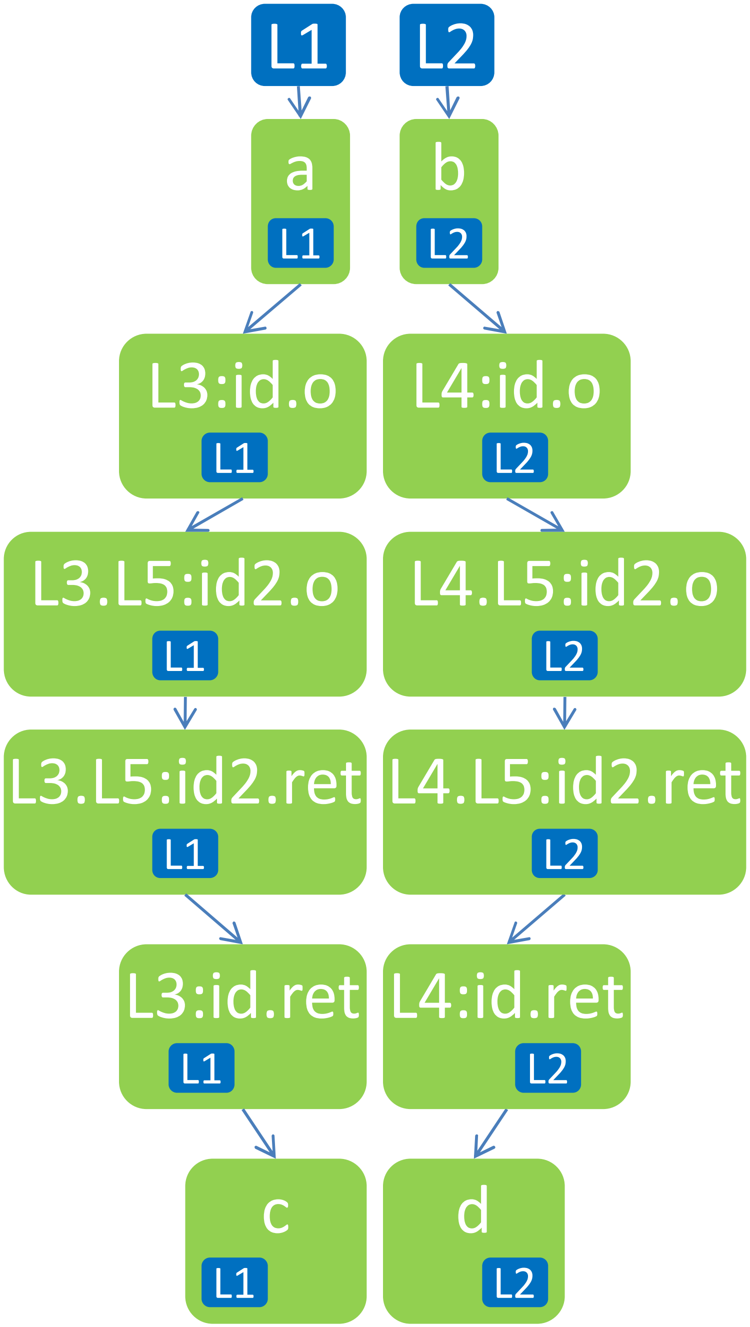
Limitation of single call site context

```
Object id(Object o) {
L5: return id2(o);
}

Object id2(Object o) {
    return o;
}

void f() {
L1: Object a = new Object();
L2: Object b = new Object();
L3: Object c = id(a);
L4: Object d = id(b);
}
```



2-call-site context sensitivity

```
Object id(Object o) {
L5: return id2(o);
}

Object id2(Object o) {
    return o;
}

void f() {
L1: Object a = new Object();
L2: Object b = new Object();
L3: Object c = id(a);
L4: Object d = id(b);
}
```

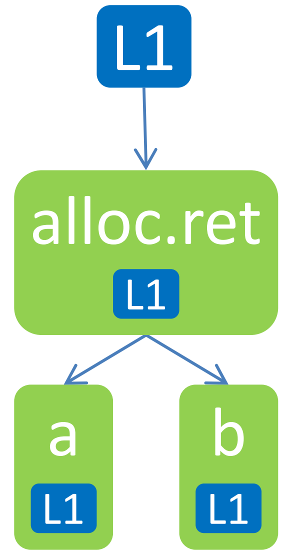


k-call-site context sensitivity

- k-Call-Site: keep last k call sites
  - space/time complexity exponential in k
- Full Call String: keep full string of call sites
  - exponential number of call strings
  - recursion: infinite number of call strings
    - exclude any call site that is in a recursive cycle from string
    - still exponential
    - what if many call sites are in recursive cycle ?
    - C: call graph is almost DAG => works well
    - Java: half of call graph is one big SCC => no precision

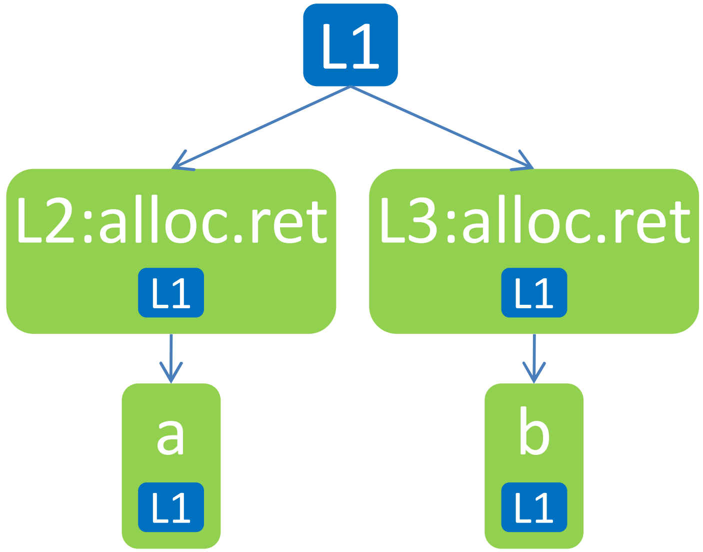
Context-sensitive heap abstraction

```
Object alloc() {
L1: return new Object();
}

void f() {
    Object a = alloc();
    Object b = alloc();
}
```



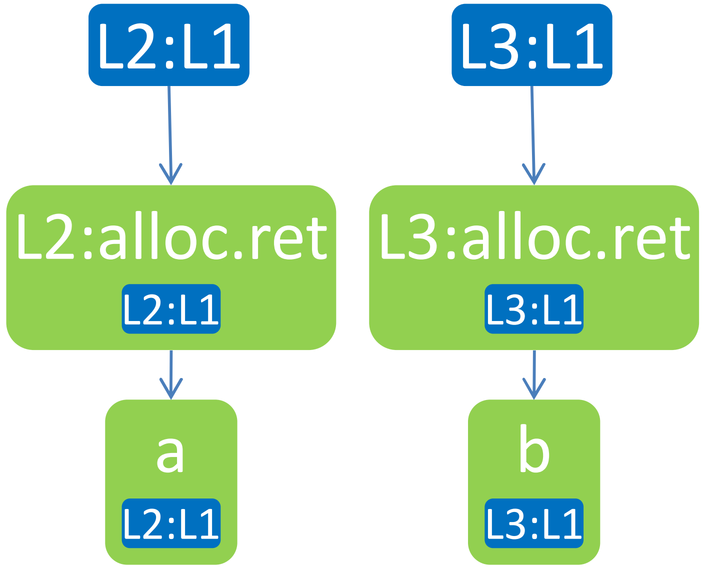
Context-sensitive heap abstraction

```
Object alloc() {
L1: return new Object();
}

void f() {
L2: Object a = alloc();
L3: Object b = alloc();
}
```



Context-sensitive heap abstraction

```
Object alloc() {
L1: return new Object();
}

void f() {
L2: Object a = alloc();
L3: Object b = alloc();
}
```



Comparison with 1CFA

Field-sensitive subset-based 1-call-site-sensitive points-to analysis with context-sensitive heap abstraction:

pt :

$$(\mathrm{Call} \times \mathrm{Var} \cup (\mathrm{Call} \times \mathrm{Obj} \times \mathrm{Field})) \rightarrow \wp(\mathrm{Call} \times \mathrm{Obj})$$

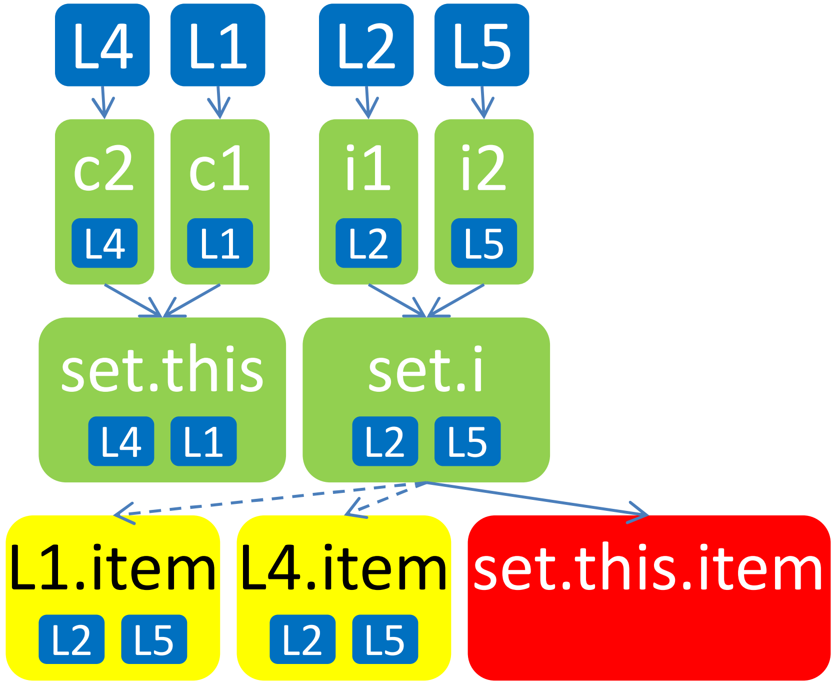
1CFA:

Σ =

$$\mathrm{Call} \times (\mathrm{Var} \rightarrow \mathrm{Addr}) \times (\mathrm{Addr} \rightarrow \wp(\mathrm{Lam} \times \mathrm{Var} \rightarrow \mathrm{Addr})) \times \mathrm{Call}$$

Object-sensitive analysis

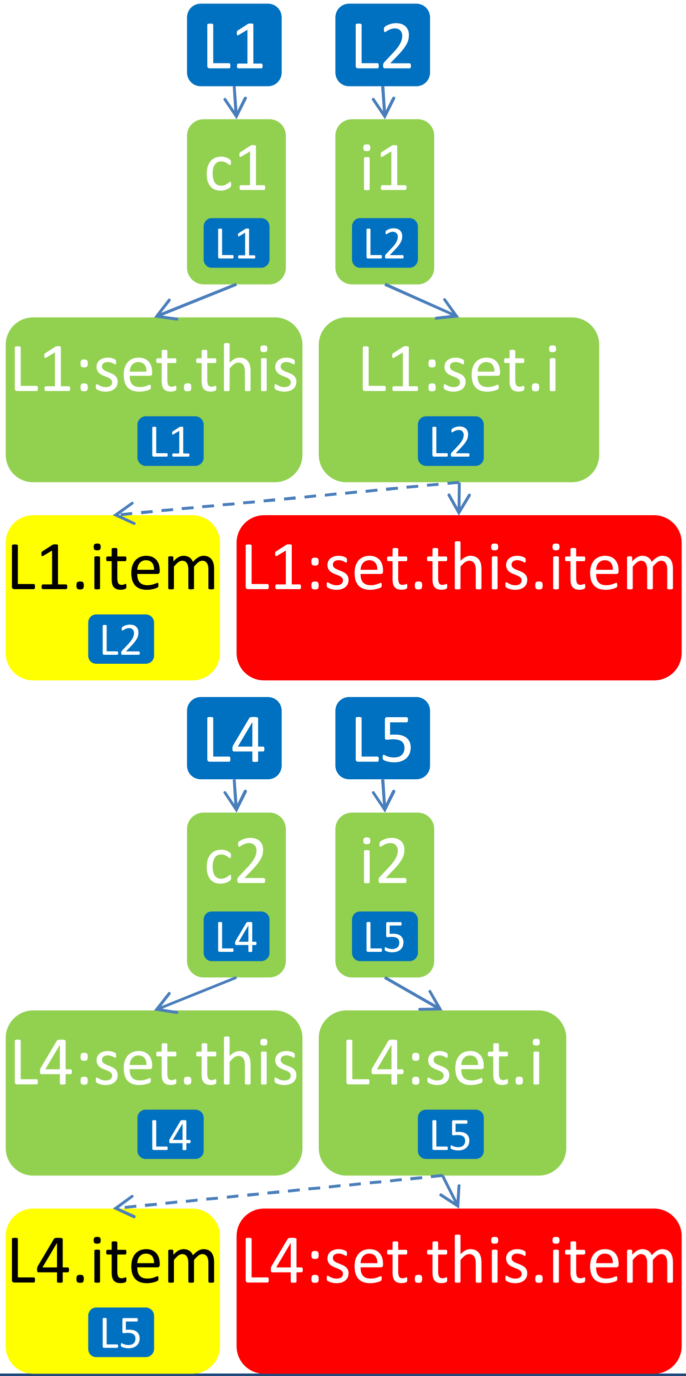
```
class Container {
   private Item item;
   public void set(Item i) {
      this.item = i;
   }
}

L1: Container c1 = new Container();
L2: Item i1 = new Item();
L3: c1.set(i1);

L4: Container c2 = new Container();
L5: Item i2 = new Item();
L6: c2.set(i2);
```



[Milanova&Ryder]

Object-sensitive analysis

```
class Container {
   private Item item;
   public void set(Item i) {
      this.item = i;
   }
}

L1: Container c1 = new Container();
L2: Item i1 = new Item();
L3: c1.set(i1);

L4: Container c2 = new Container();
L5: Item i2 = new Item();
L6: c2.set(i2);
```



[Milanova&Ryder]

Object-sensitive analysis

- Like call-site context-sensitive analysis:
  - can use strings of abstract objects
  - can make heap abstraction (object-)context-sensitive
- Call-site CS and object-sensitive CS have incomparable precision (neither is theoretically more precise)
- In practice, for OO programs, object sensitivity more precise than call-site sensitivity for the same context string length

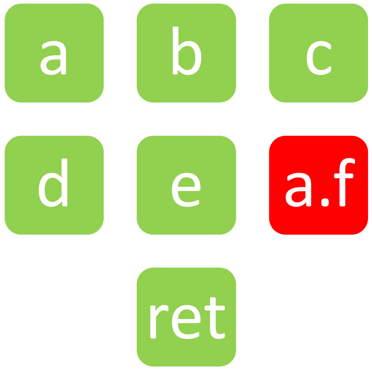
[Milanova&Ryder]

Effect of context-sensitivity in Java

- For call graph construction:
  - context sensitivity has some effect
- For cast safety analysis:
  - context sensitivity substantially improves precision
  - object sensitivity more precise than call sites
  - context sensitive heap abstraction further improves precision
  - context strings longer than 1 add little precision
  - ∞-call-site ignoring cycles less precise than 1-call-site

Context-sensitive equality-based analysis

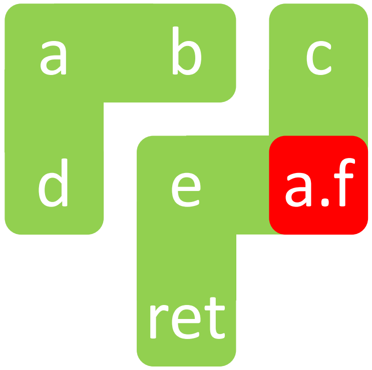
```
f(a, b, c) {
   d = a;
   d = b;
   e = c;
   e = a.f;
   return e;
}

L1: x = new A();
L2: y = new A();
L3: z = new A();
L4: w = new A();
    x.f = w;
    v = f(x, y, z);
```



[Lattner&Adve]

Context-sensitive equality-based analysis

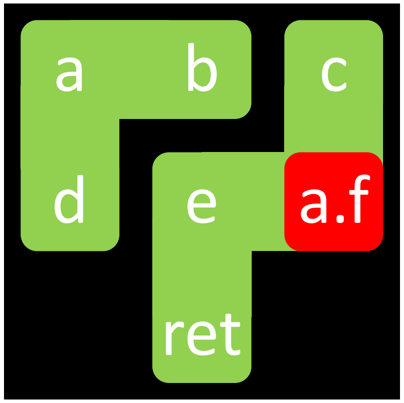
```
f(a, b, c) {
   d = a;
   d = b;
   e = c;
   e = a.f;
   return e;
}

L1: x = new A();
L2: y = new A();
L3: z = new A();
L4: w = new A();
    x.f = w;
    v = f(x, y, z);
```



[Lattner&Adve]

Context-sensitive equality-based analysis

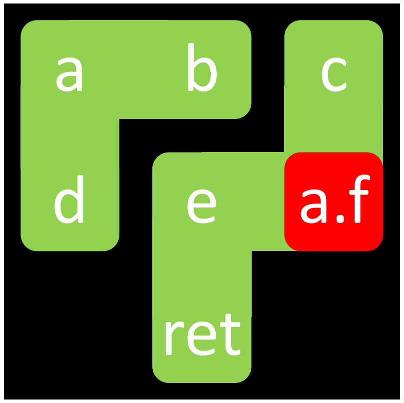
```
f(a, b, c) {
   d = a;
   d = b;
   e = c;
   e = a.f;
   return e;
}

L1: x = new A();
L2: y = new A();
L3: z = new A();
L4: w = new A();
    x.f = w;
    v = f(x, y, z);
```

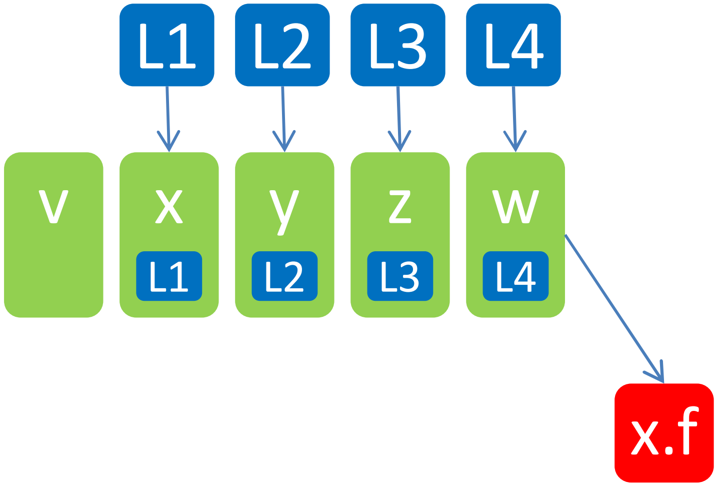


[Lattner&Adve]

Context-sensitive equality-based analysis

```
f(a, b, c) {
   d = a;
   d = b;
   e = c;
   e = a.f;
   return e;
}

L1: x = new A();
L2: y = new A();
L3: z = new A();
L4: w = new A();
    x.f = w;
    v = f(x, y, z);
```

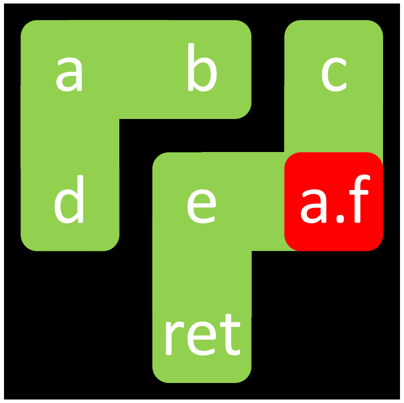


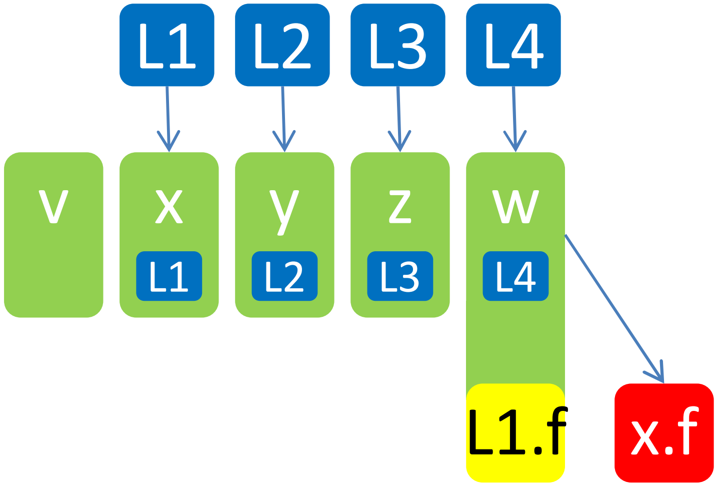


[Lattner&Adve]

Context-sensitive equality-based analysis

```
f(a, b, c) {
   d = a;
   d = b;
   e = c;
   e = a.f;
   return e;
}

L1: x = new A();
L2: y = new A();
L3: z = new A();
L4: w = new A();
    x.f = w;
    v = f(x, y, z);
```

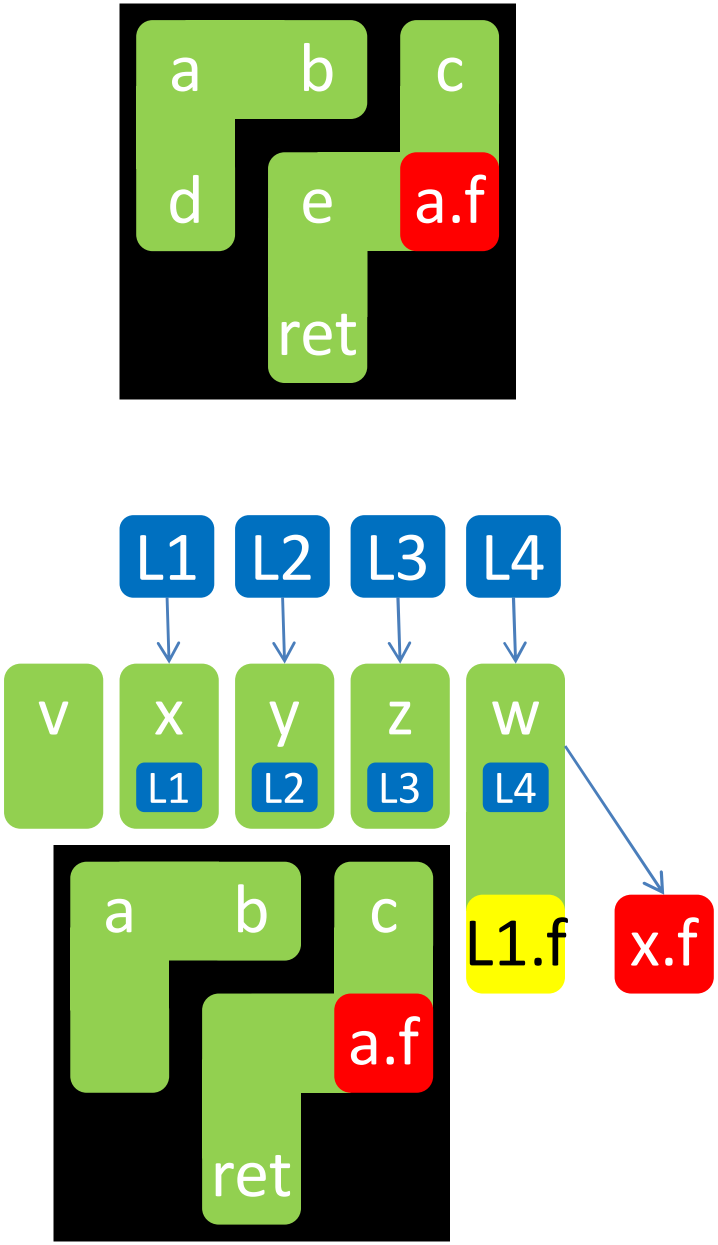




[Lattner&Adve]

Context-sensitive equality-based analysis

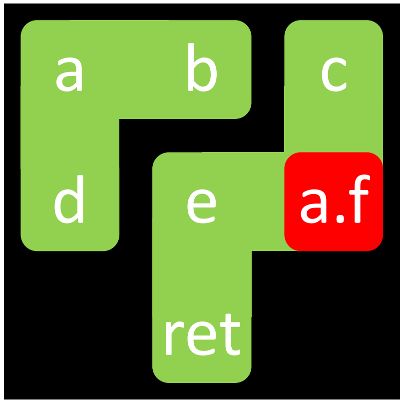
```
f(a, b, c) {
   d = a;
   d = b;
   e = c;
   e = a.f;
   return e;
}

L1: x = new A();
L2: y = new A();
L3: z = new A();
L4: w = new A();
    x.f = w;
    v = f(x, y, z);
```

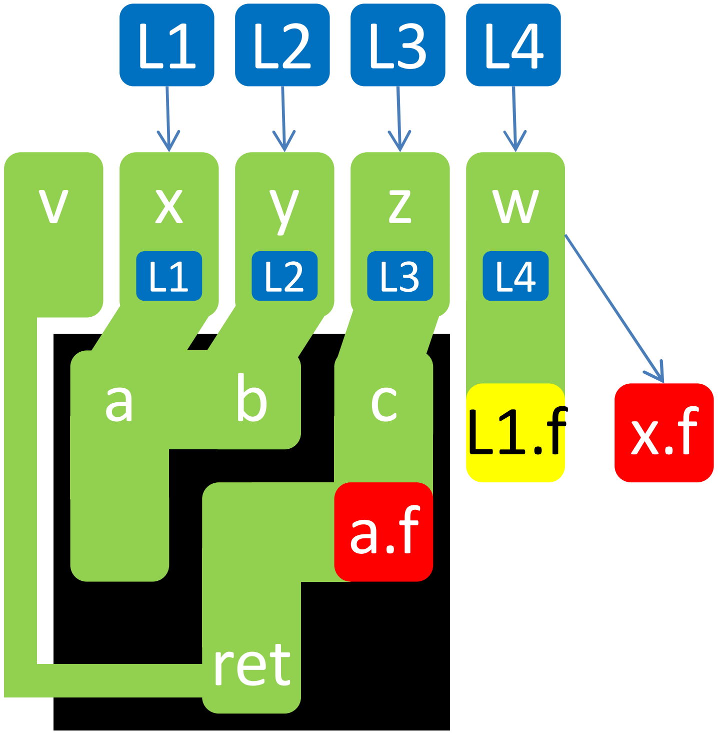


[Lattner&Adve]

Context-sensitive equality-based analysis

```
f(a, b, c) {
   d = a;
   d = b;
   e = c;
   e = a.f;
   return e;
}

L1: x = new A();
L2: y = new A();
L3: z = new A();
L4: w = new A();
    x.f = w;
    v = f(x, y, z);
```

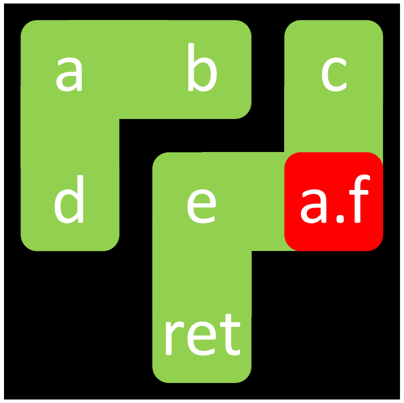


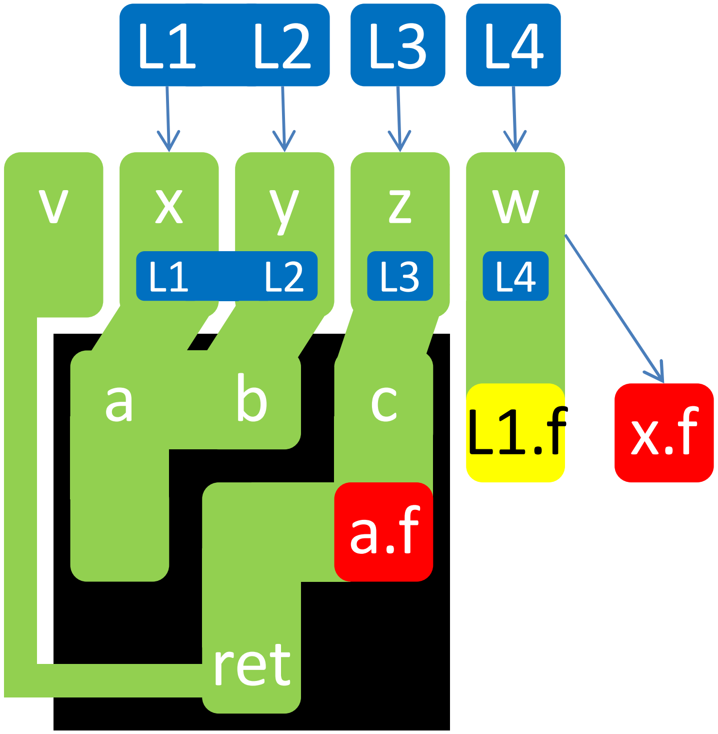


[Lattner&Adve]

Context-sensitive equality-based analysis

```
f(a, b, c) {
    d = a;
    d = b;
    e = c;
    e = a.f;
    return e;
}

L1: x = new A();
L2: y = new A();
L3: z = new A();
L4: w = new A();
    x.f = w;
    v = f(x, y, z);
```

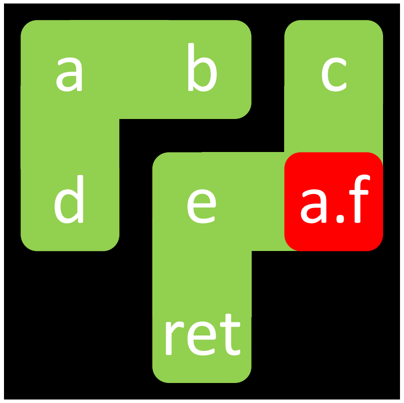


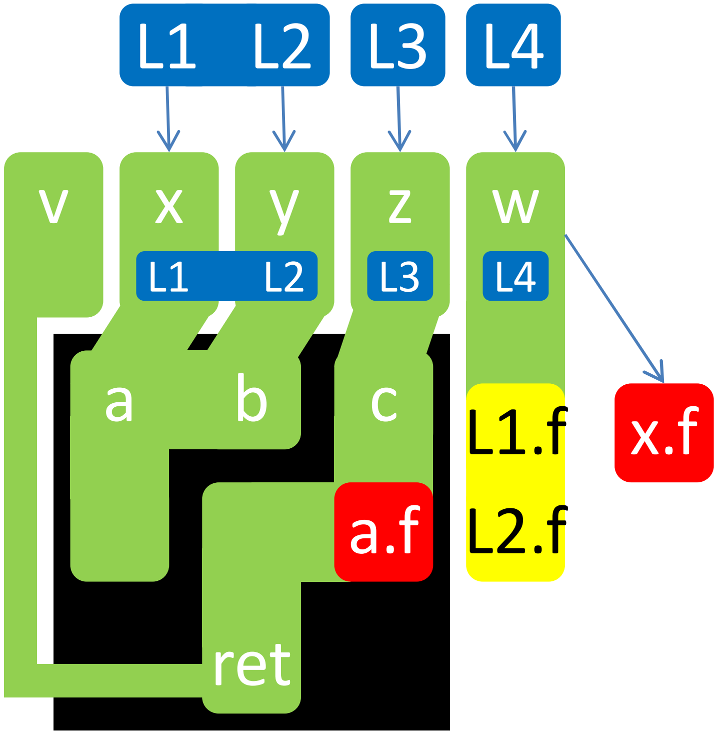


[Lattner&Adve]

Context-sensitive equality-based analysis

```
f(a, b, c) {
   d = a;
   d = b;
   e = c;
   e = a.f;
   return e;
}

L1: x = new A();
L2: y = new A();
L3: z = new A();
L4: w = new A();
    x.f = w;
    v = f(x, y, z);
```

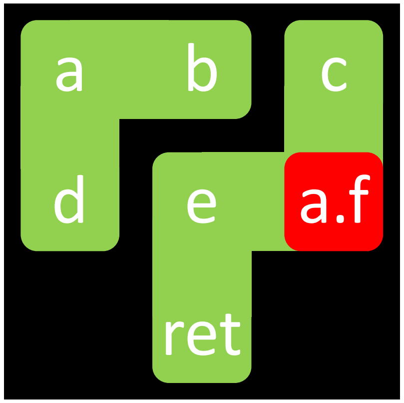


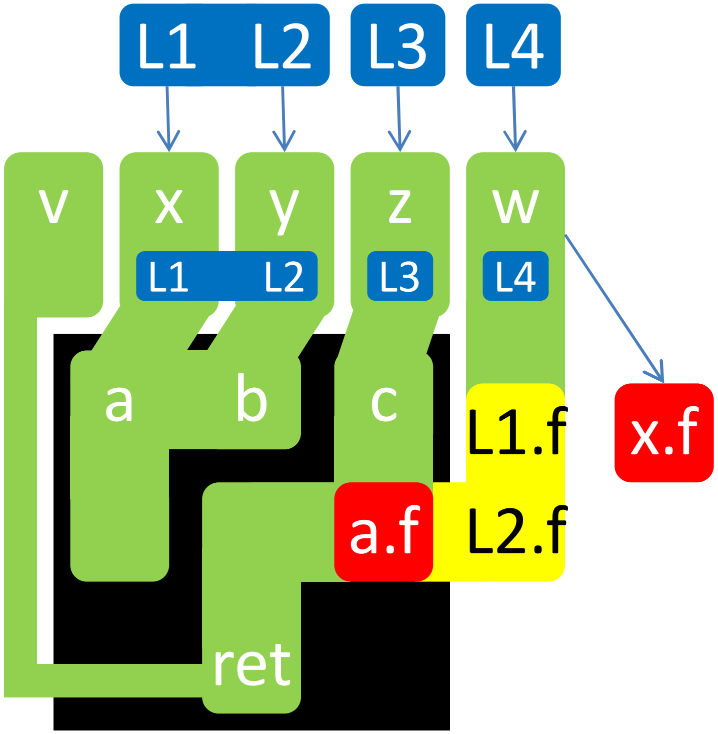


[Lattner&Adve]

Context-sensitive equality-based analysis

```
f(a, b, c) {
   d = a;
   d = b;
   e = c;
   e = a.f;
   return e;
}

L1: x = new A();
L2: y = new A();
L3: z = new A();
L4: w = new A();
    x.f = w;
    v = f(x, y, z);
```

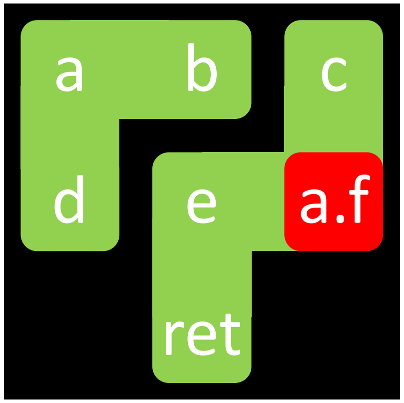


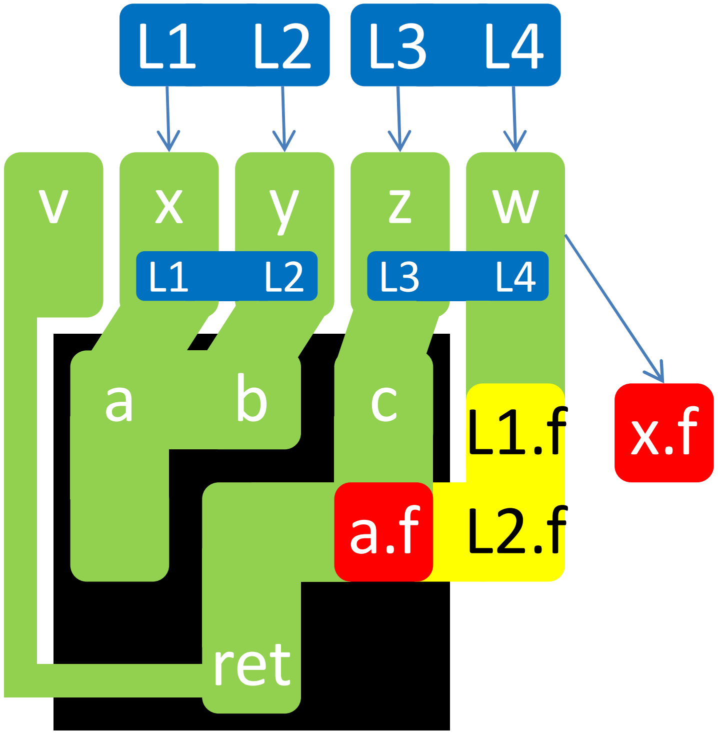


[Lattner&Adve]

Context-sensitive equality-based analysis

```
f(a, b, c) {
   d = a;
   d = b;
   e = c;
   e = a.f;
   return e;
}

L1: x = new A();
L2: y = new A();
L3: z = new A();
L4: w = new A();
    x.f = w;
    v = f(x, y, z);
```

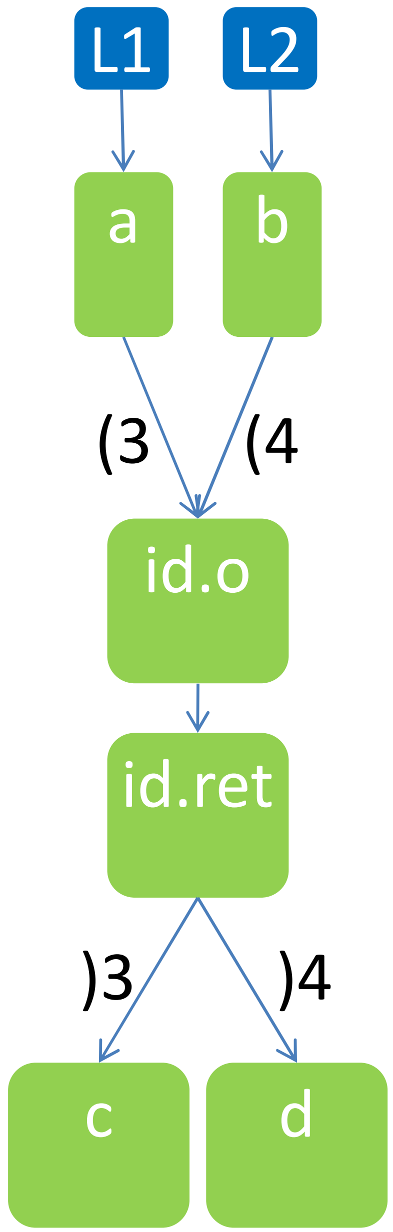




[Lattner&Adve]

Refinement demand-driven analysis

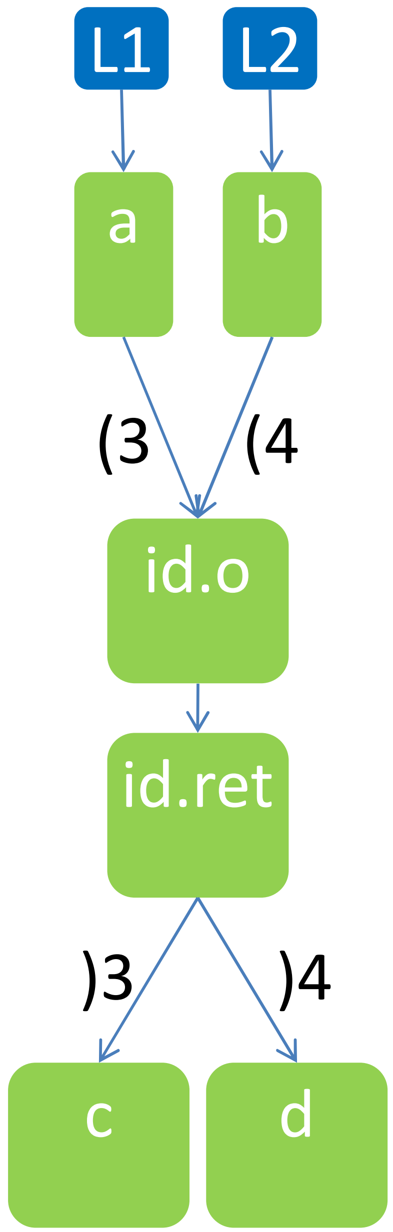
```
Object id(Object o) {
   return o;
}

void f() {
L1: Object a = new Object();
L2: Object b = new Object();
L3: Object c = id(a);
L4: Object d = id(b);
}
```



[Sridharan&Bodík]

Refinement demand-driven analysis

```
Object id(Object o) {
   return o;
}

void f() {
L1: Object a = new Object();
L2: Object b = new Object();
L3: Object c = id(a);
L4: Object d = id(b);
}
```

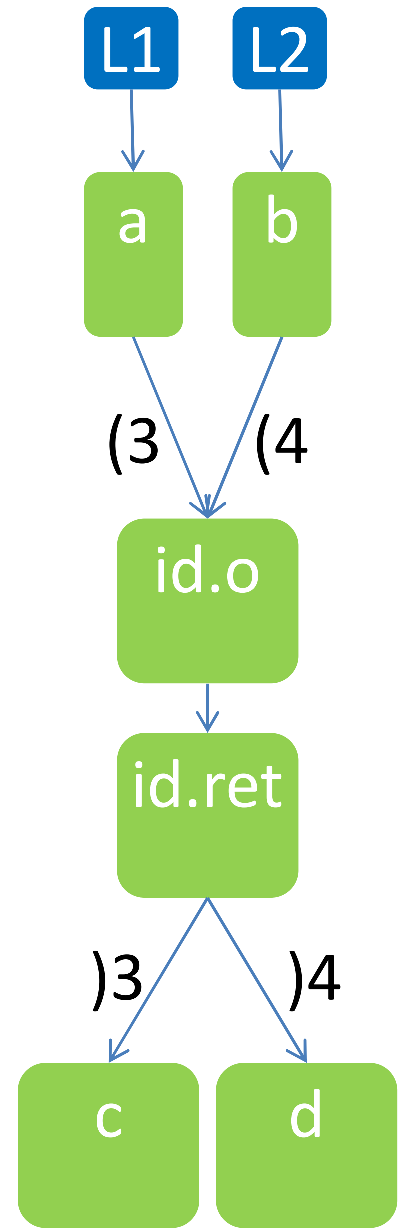


L1 ∈ pt(d) ⇔ ∃ balanced-parens path from L1 to d

[Sridharan&Bodík]

Refinement demand-driven analysis

```
Object id(Object o) {
   return o;
}

void f() {
L1: Object a = new Object();
L2: Object b = new Object();
L3: Object c = id(a);
L4: Object d = id(b);
}
```



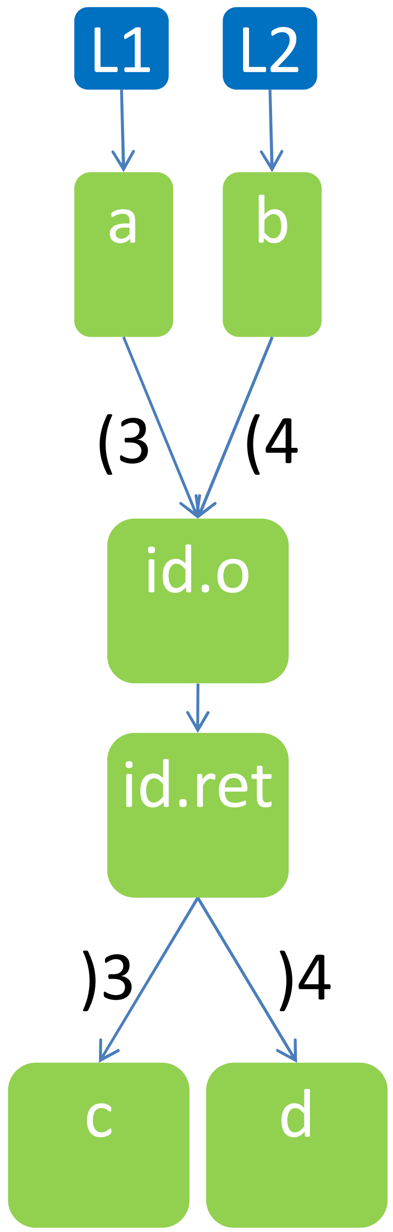
L1 ∈ pt(d) ⇔ ∃ balanced-parens path from L1 to d

But there are lots of paths to search.

[Sridharan&Bodík]

Refinement demand-driven analysis

```
Object id(Object o) {
   return o;
}

void f() {
L1: Object a = new Object();
L2: Object b = new Object();
L3: Object c = id(a);
L4: Object d = id(b);
}
```



L1 $\in$ pt(d) $\Leftrightarrow$ $\exists$ balanced-parens path from L1 to d

But there are lots of paths to search.

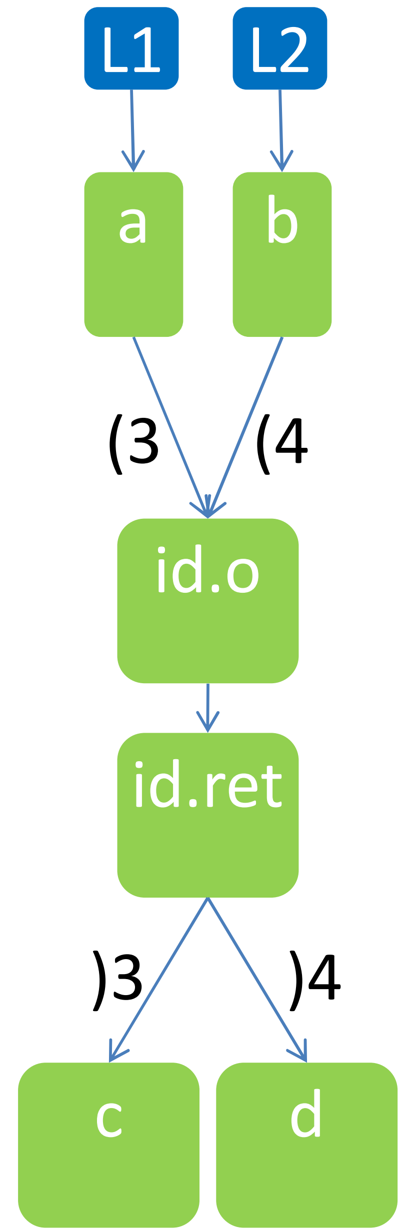
Add shortcut edges to graph.

No path even with shortcuts $\Rightarrow$ no path at all.

[Sridharan&Bodík]

Refinement demand-driven analysis

```
Object id(Object o) {
   return o;
}

void f() {
L1: Object a = new Object();
L2: Object b = new Object();
L3: Object c = id(a);
L4: Object d = id(b);
}
```



L1 $\in$ pt(d) $\Leftrightarrow$ $\exists$ balanced-parens path from L1 to d

But there are lots of paths to search.

Add shortcut edges to graph.

No path even with shortcuts $\Rightarrow$ no path at all.

On balanced paths found, gradually remove shortcuts.

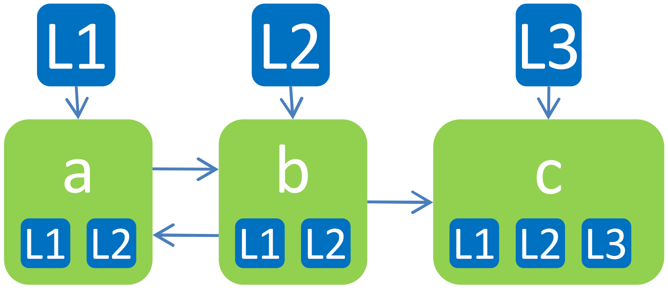
[Sridharan&Bodík]

Design decisions for precision/efficiency

- The abstraction (affects precision and efficiency):
  - Type filtering
  - Field sensitivity
  - Directionality
  - Call graph construction
  - Context sensitivity
  - Flow sensitivity
- Algorithm and implementation (affects efficiency)
  - Propagation algorithm
  - Set implementation

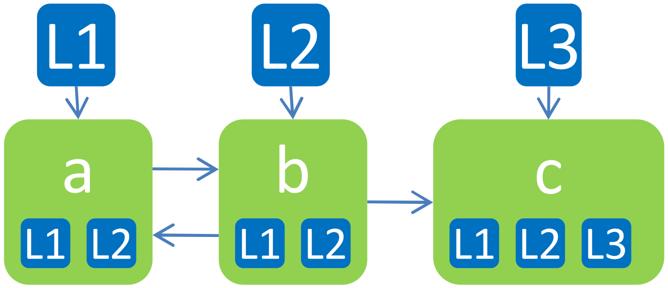
Flow sensitivity

```
L1: a = new Object();
L2: b = new Object();
L3: c = new Object();
L4: a = b;
L5: b = a;
L6: c = b;
```



Flow sensitivity

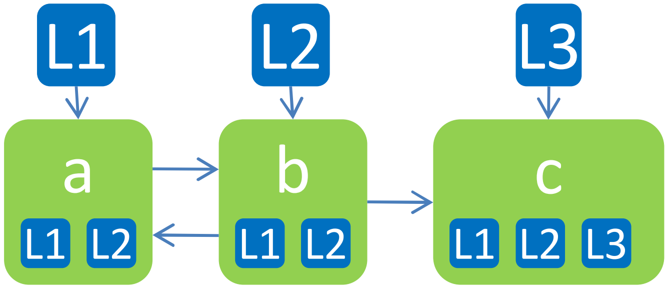
```
L1: a = new Object();  a -> {L1}
L2: b = new Object();  a -> {L1}, b -> {L2}
L3: c = new Object();  a -> {L1}, b -> {L2}, c -> {L3}
L4: a = b;             a -> {L2}, b -> {L2}, c -> {L3}
L5: b = a;             a -> {L2}, b -> {L2}, c -> {L3}
L6: c = b;             a -> {L2}, b -> {L2}, c -> {L2}
```



Strong updates: overwrite existing pt-set contents

Flow sensitivity using SSA form

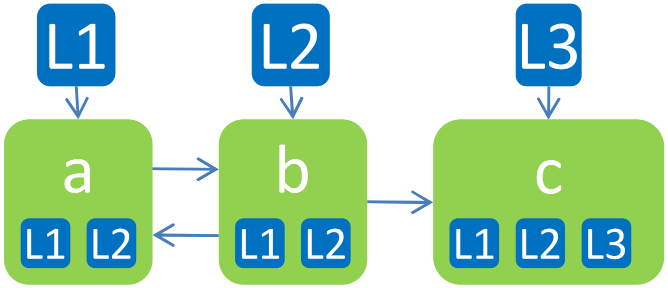
```
                                                                  a b c
L1: a = new Object();  a -> {L1}                                  1
L2: b = new Object();  a -> {L1}, b -> {L2}                       1 2
L3: c = new Object();  a -> {L1}, b -> {L2}, c -> {L3}            1 2 3
L4: a = b;             a -> {L2}, b -> {L2}, c -> {L3}            4 2 3
L5: b = a;             a -> {L2}, b -> {L2}, c -> {L3}            4 5 3
L6: c = b;             a -> {L2}, b -> {L2}, c -> {L2}            4 5 6
```

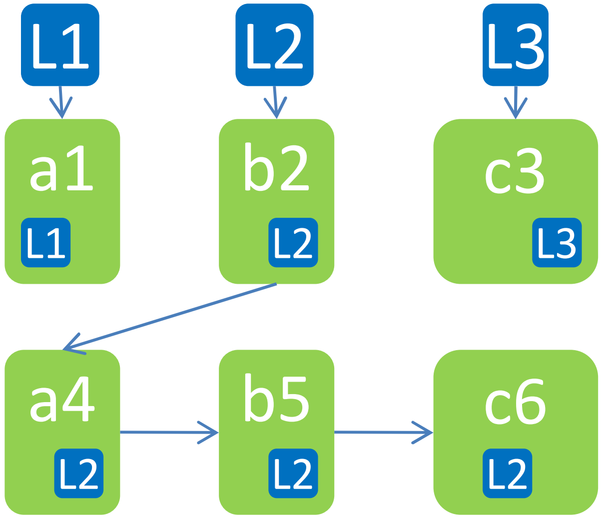


Currently live reaching definition of each variable.

Flow sensitivity using SSA form

```
                                                                          a b c
L1: a1= new Object();  a1-> {L1}                                          1
L2: b2= new Object();  a1-> {L1}, b2-> {L2}                               1 2
L3: c3= new Object();  a1-> {L1}, b2-> {L2}, c3-> {L3}                    1 2 3
L4: a4= b2;            a4-> {L2}, b2-> {L2}, c3-> {L3}                    4 2 3
L5: b5= a4;            a4-> {L2}, b5-> {L2}, c3-> {L3}                    4 5 3
L6: c6= b5;            a4-> {L2}, b5-> {L2}, c6-> {L2}                    4 5 6
```





For local variables, FI analysis on SSA form gives same result as FS analysis on original program.

Flow sensitivity using SSA form

```
                  a
L1: if(*) {
L2:    a = b;     2
L3: } else {
L4:    a = c;     4
L5: }             ?
L6:               ?
L7: d = a;        ?
```

Which definition of a is current at L7?

Flow sensitivity using SSA form

```
                      a
L1: if(*) {
L2:    a = b;         2
L3: } else {
L4:    a = c;         4
L5: }                 ?
L6: a = ϕ(a2,a4)      6
L7: d = a;            6
```

Which definition of a is current at L7?

Use ϕ to create new definition of a.

pt(a6) = pt(a2) ∪ pt(a4)

When does flow sensitivity matter?

	Java	C/C++
local variables	no, use SSA form	
address-taken local variables	no, don't exist	possibly
global variables	unlikely, values usually long-lived	
fields of heap objects	no strong updates on heap objects unless analysis extended with single-concrete-object abstraction	

Design decisions for precision/efficiency

- The abstraction (affects precision and efficiency):
  - –Type filtering
  - –Field sensitivity
  - –Directionality
  - –Call graph construction
  - –Context sensitivity
  - –Flow sensitivity
- Algorithm and implementation (affects efficiency)
  - –Propagation algorithm
  - –Set implementation