Practical Virtual Method Call Resolution for Java

Vijay Sundaresan, Laurie Hendren, Chrislain Razafimahefa, Raja Vall´ee-Rai, Patrick Lam, Etienne Gagnon and Charles Godin

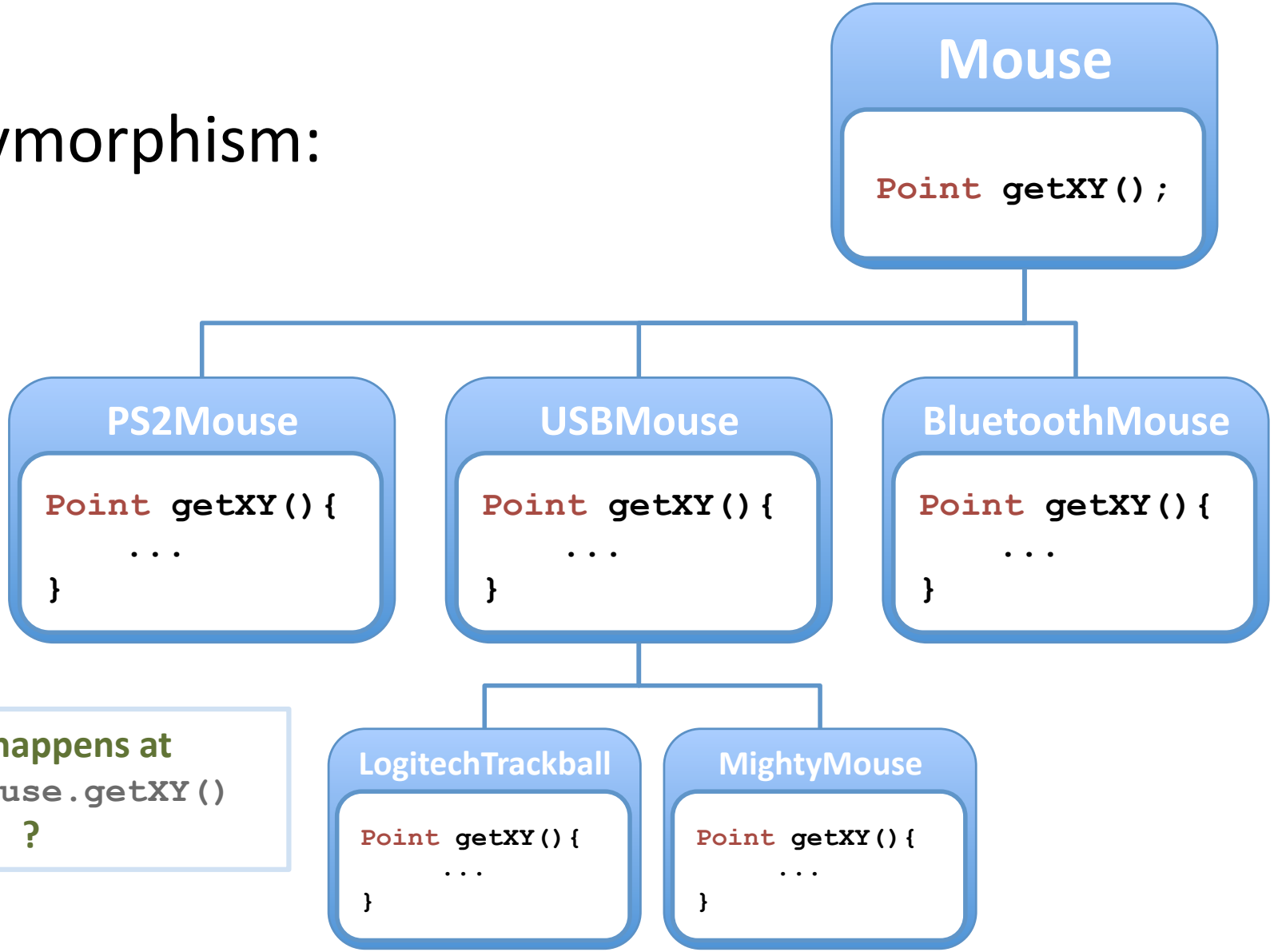
Sable Research Group

School of Computer Science

McGill University Montreal

The Problem

- Polymorphism:



What happens at

`pos = mouse.getXY()`

?

Motivation

1. **Compact Executable**
   Remove functions which are never called
2. **Faster Method Calls**
   Identify monomorphic call sites
3. **Predictable Flow**
   Reduce number of flows, for other analyses

Class Hierarchy Analysis

For every class or instance d, we have

$$hierarchy - types(d) \subseteq types$$

Defined by the following recursion:

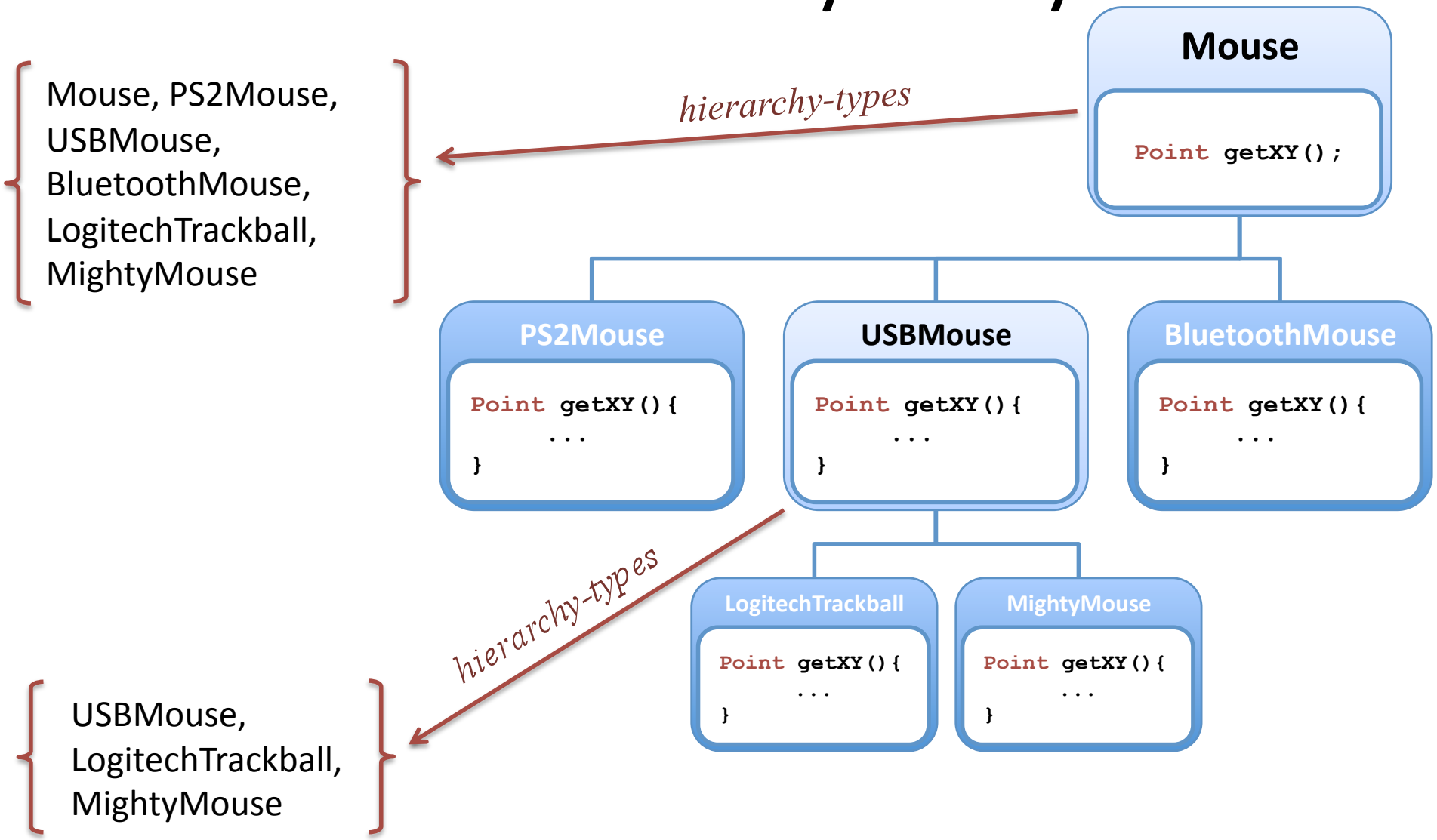
For every class or interface d

$$d \in hierarchy - types(d)$$

If d_2 **extends** d_1 or if d_2 **implements** d_1

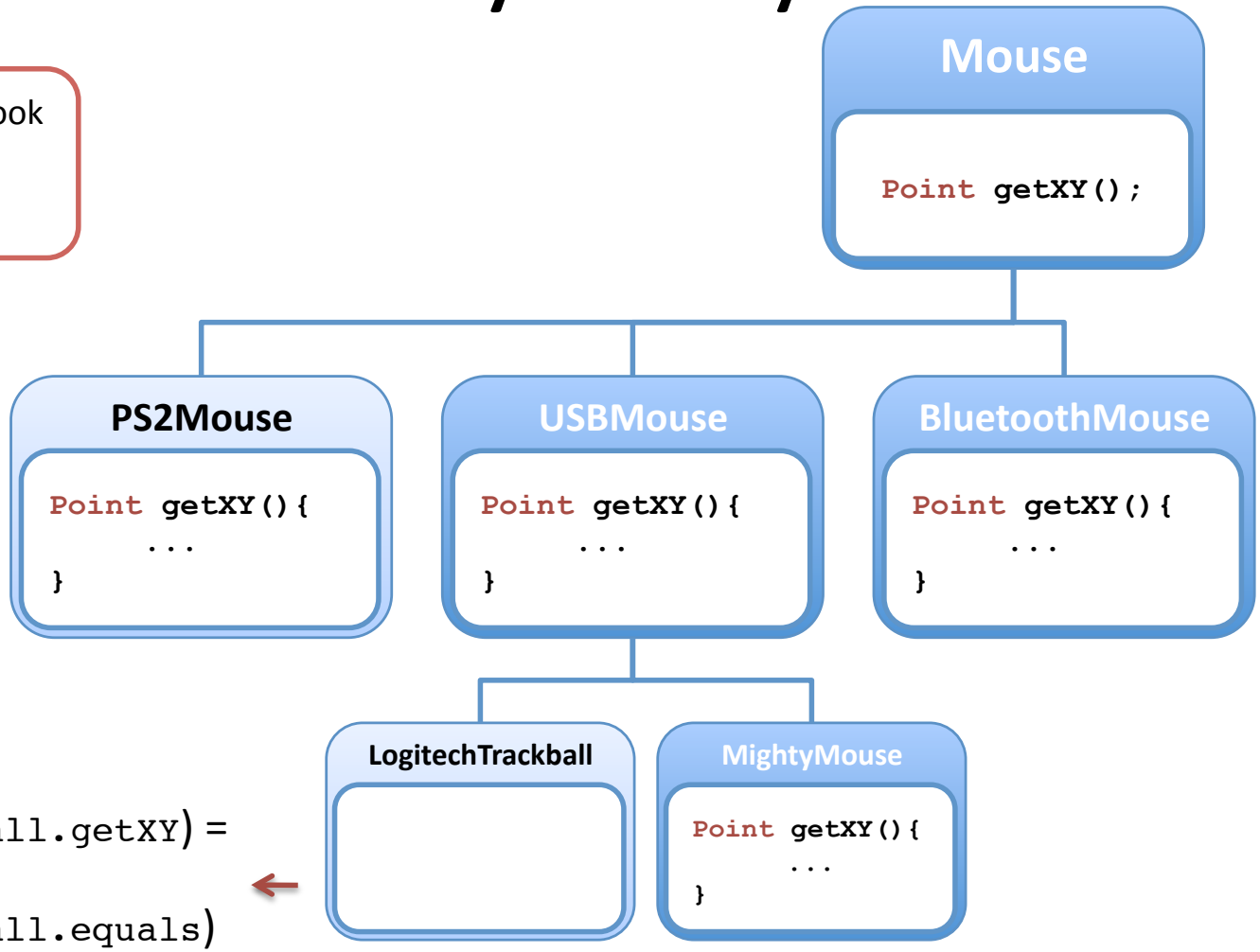
$$hierarchy - types(d_2) \subseteq hierarchy - types(d_1)$$

Class Hierarchy Analysis



Class Hierarchy Analysis

Sometimes you may need to look up the hierarchy



look-up(`PS2Mouse.getXY`) =
`PS2Mouse.getXY`

look-up(`LogitechTrackball.getXY`) =
`USBMouse.getXY`

look-up(`LogitechTrackball.equals`)
= `object.equals`

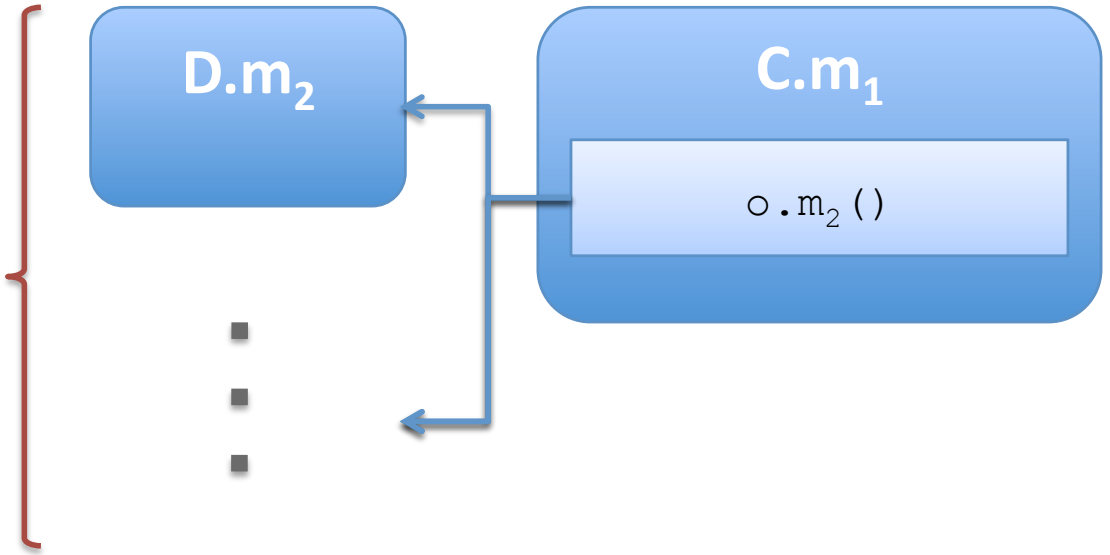
Call Graph

(pessimistic)

For a **receiver** o of type d,

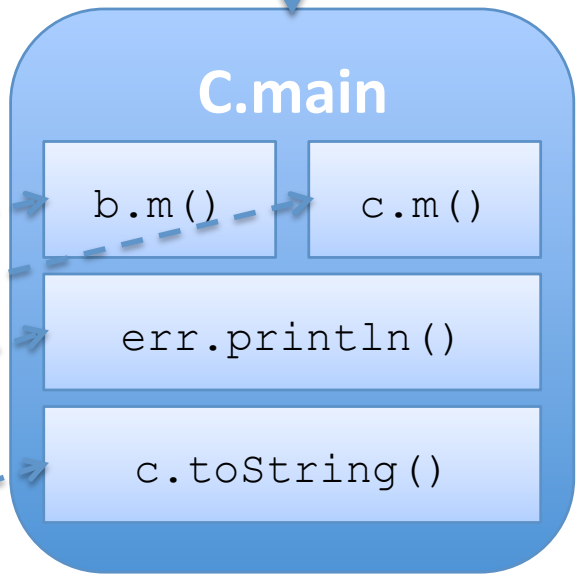
where

$D.m_2 = look\text{-}up(d'.m_2)$
for all d'
in $hierarchy\text{-}types(d)$

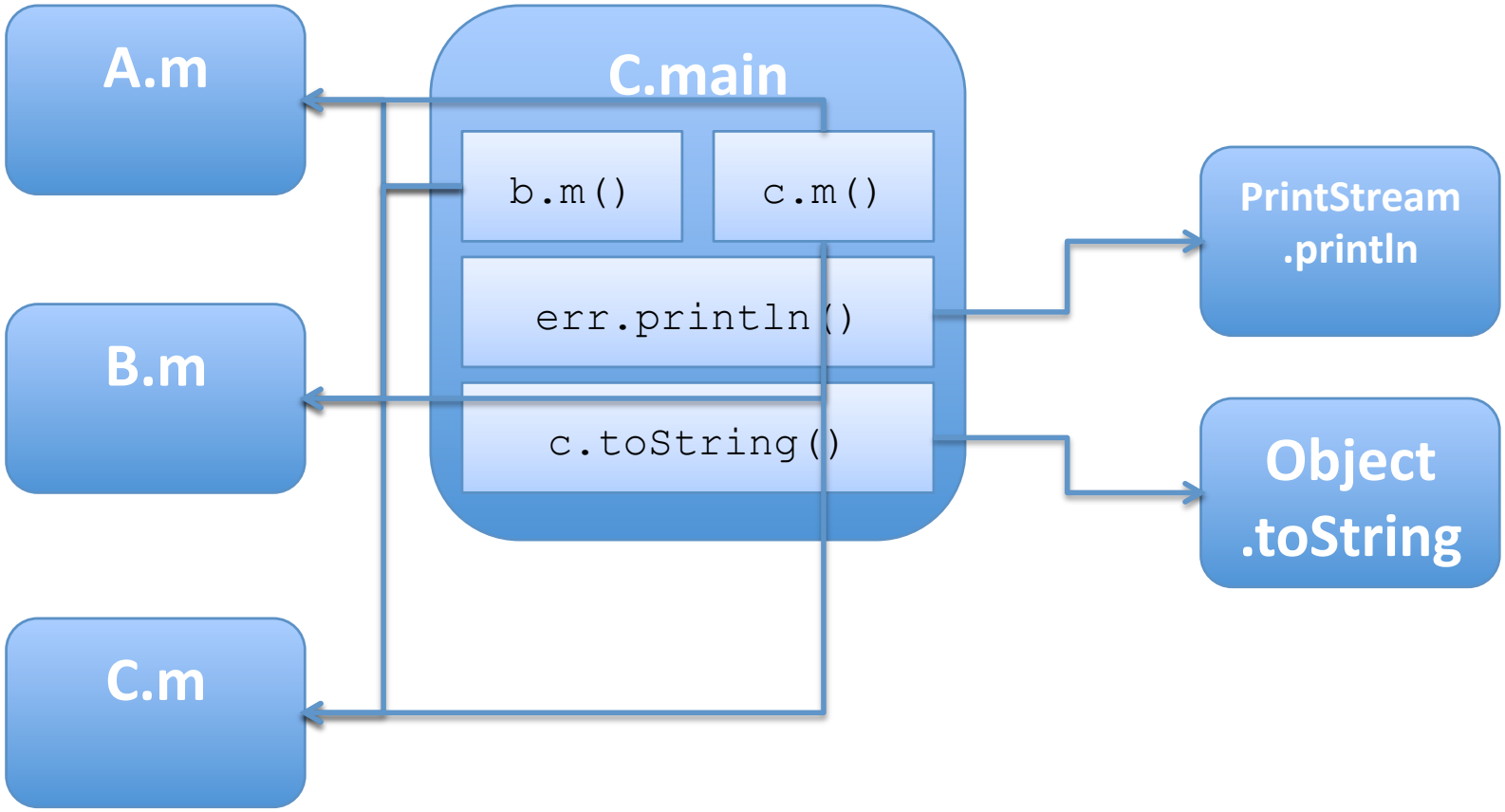


Call Graph

```
public class C extends A
{
    public static void main(String[] p)
    {
        A b = new B();
        A c = new C();
        b.m();
        c.m();
        System.err.println(c.toString());
    }
}
```



Call Graph


A.m
B.m
C.m
C.main
b.m()
c.m()
err.println()
c.toString()
PrintStream
.println
Object
.toString

Rapid Type Analysis

Improvement over Class Hierarchy

For a program P –

$$instantiated - types(P) \stackrel{\Delta}{=} \{d \mid \exists a\ stmt\ \text{"new } d\text{"}\ in\ P\}$$

- Get a more accurate result by only considering

$$hierarchy - types(d) \cap instantiated - types(P)$$

- Build call graph (and P) iteratively

Variable-Type Analysis

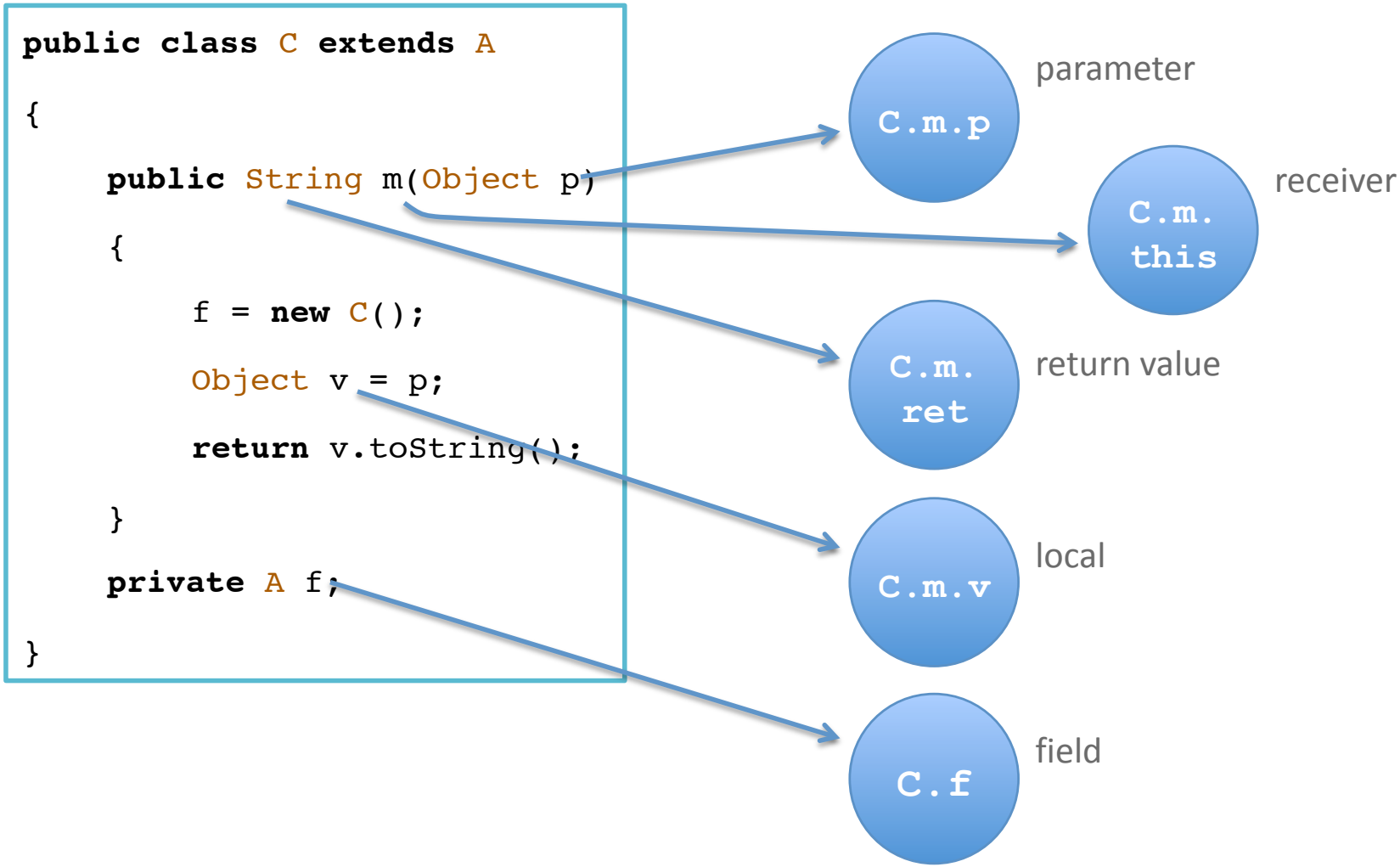
If

a receiver o may be of type d at run-time,
(where d is a subclass of the declared type of o)

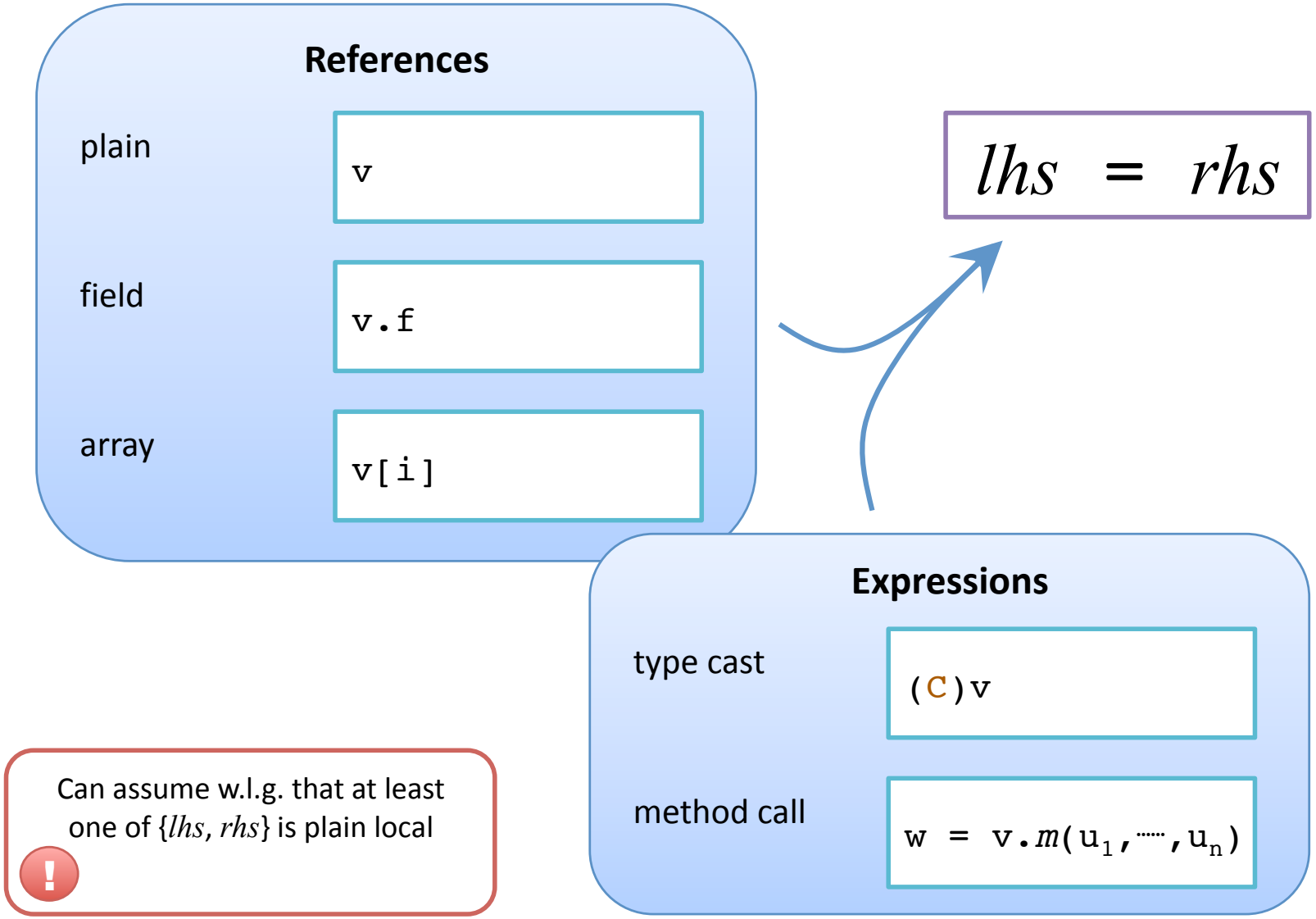
then

there must be a chain of assignments of the form
$o = \text{var}_n = \text{var}_{n-1} = \ldots = \text{var}_1 = \texttt{new}\ d()$

Representative Nodes

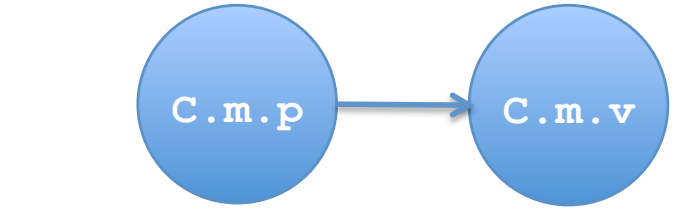


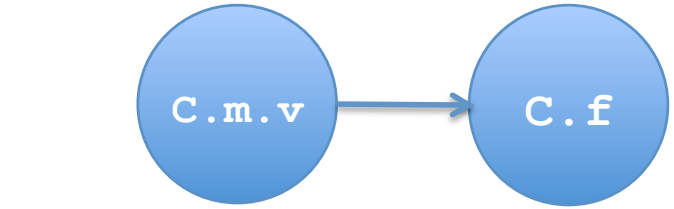
Anatomy of Assignments

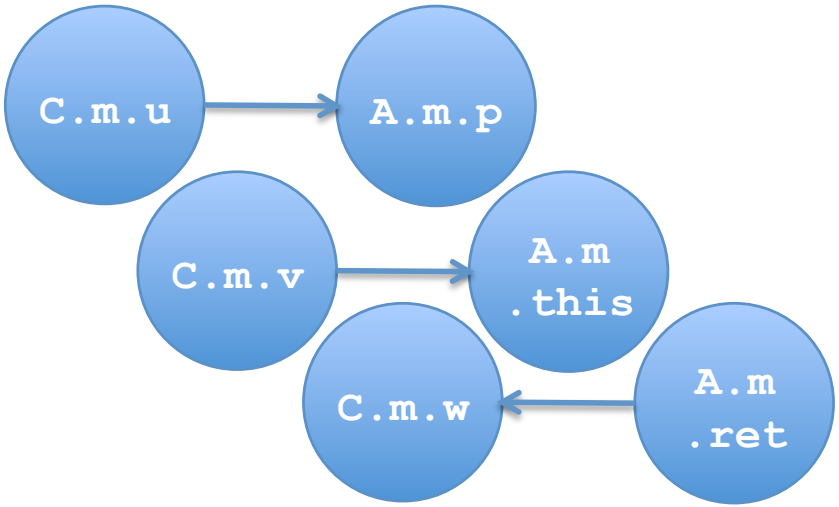


Representative Edges

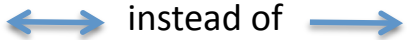
```
public class C extends A
{
      public String m(Object p)
      {
            A  u, v, w;
            v = p;
            this.f = v;
            w = v.m(u);
      }
}
```





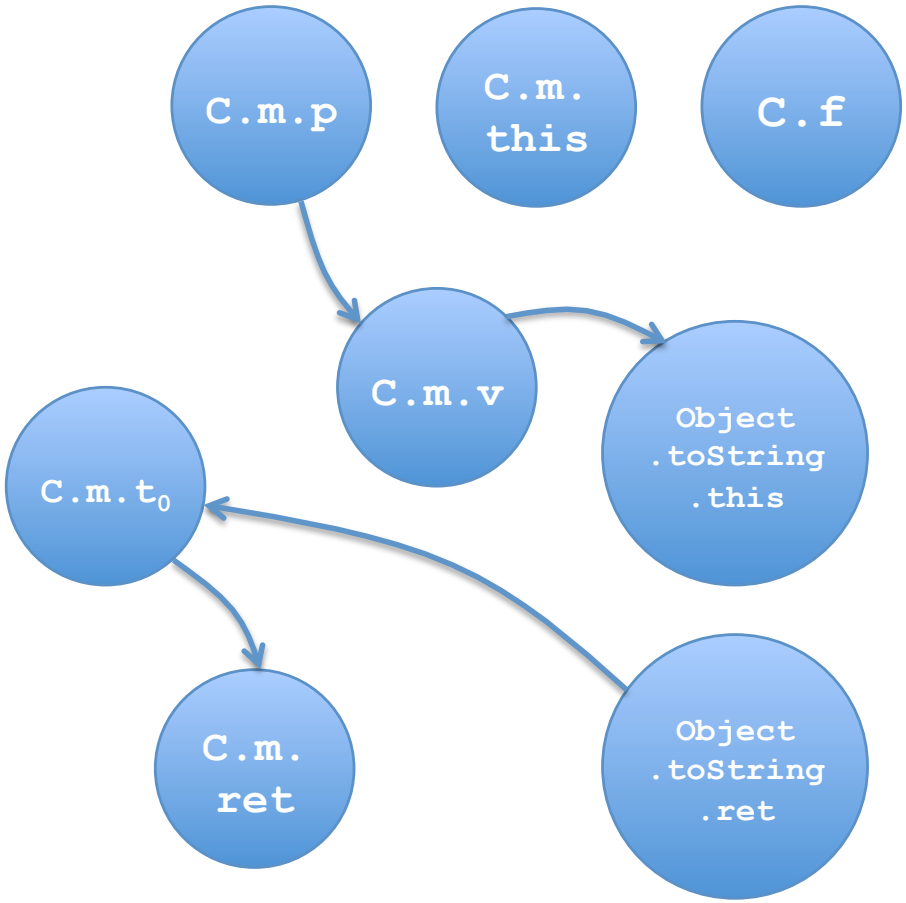


For array or `Object` references:



Representative Graph

```
public class C extends A
{
    public String m(Object p)
    {
        f = new C();
        Object v = p;
        return v.toString();
    }
    private A f;
}
```



Instances

$$instances(n) \stackrel{\Delta}{=} \{d \mid \exists assignment\ n = \text{new}\ d\}$$

Variable-Type Analysis

If

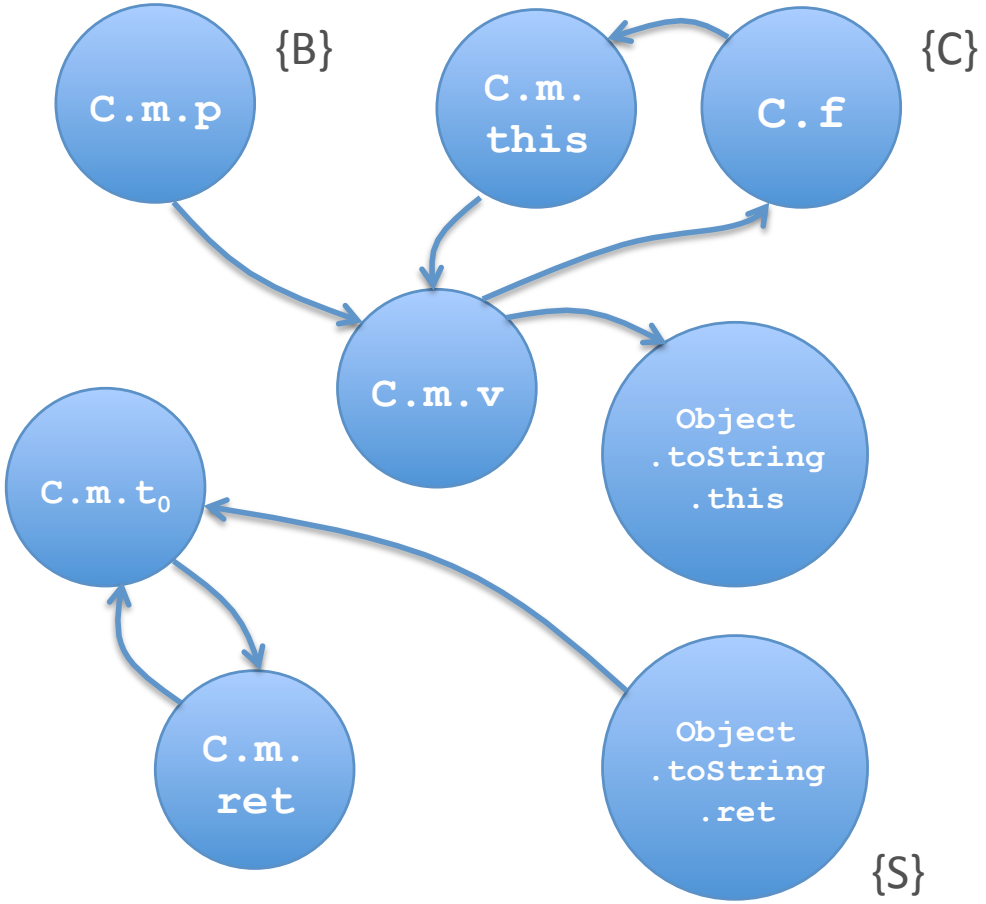
a receiver o may be of type d at run-time,
(where d is a subclass of the declared type of o)

then

there must be a path in the representative graph
from a node n to $representative(o)$ s.t. d in $instances$(n)

Variable Type Analysis

Improving Performance

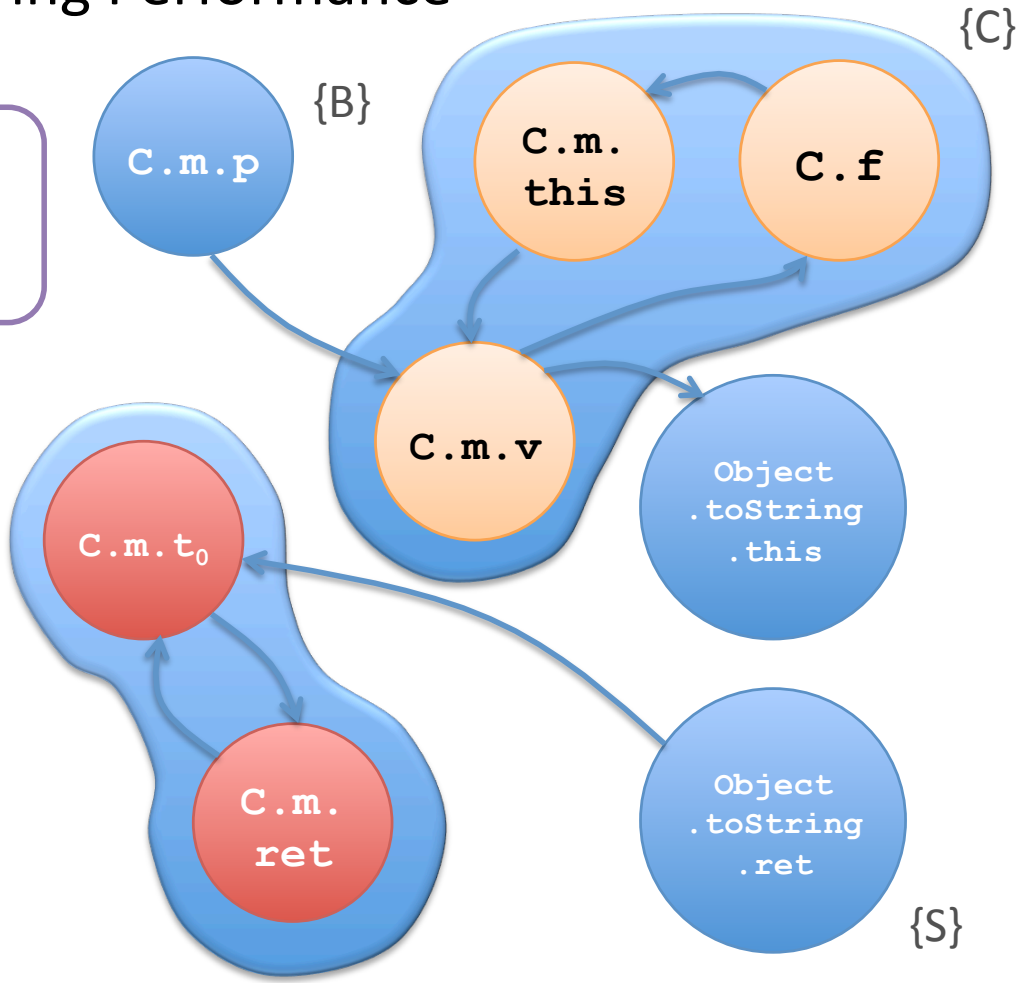


Variable Type Analysis

Improving Performance

1

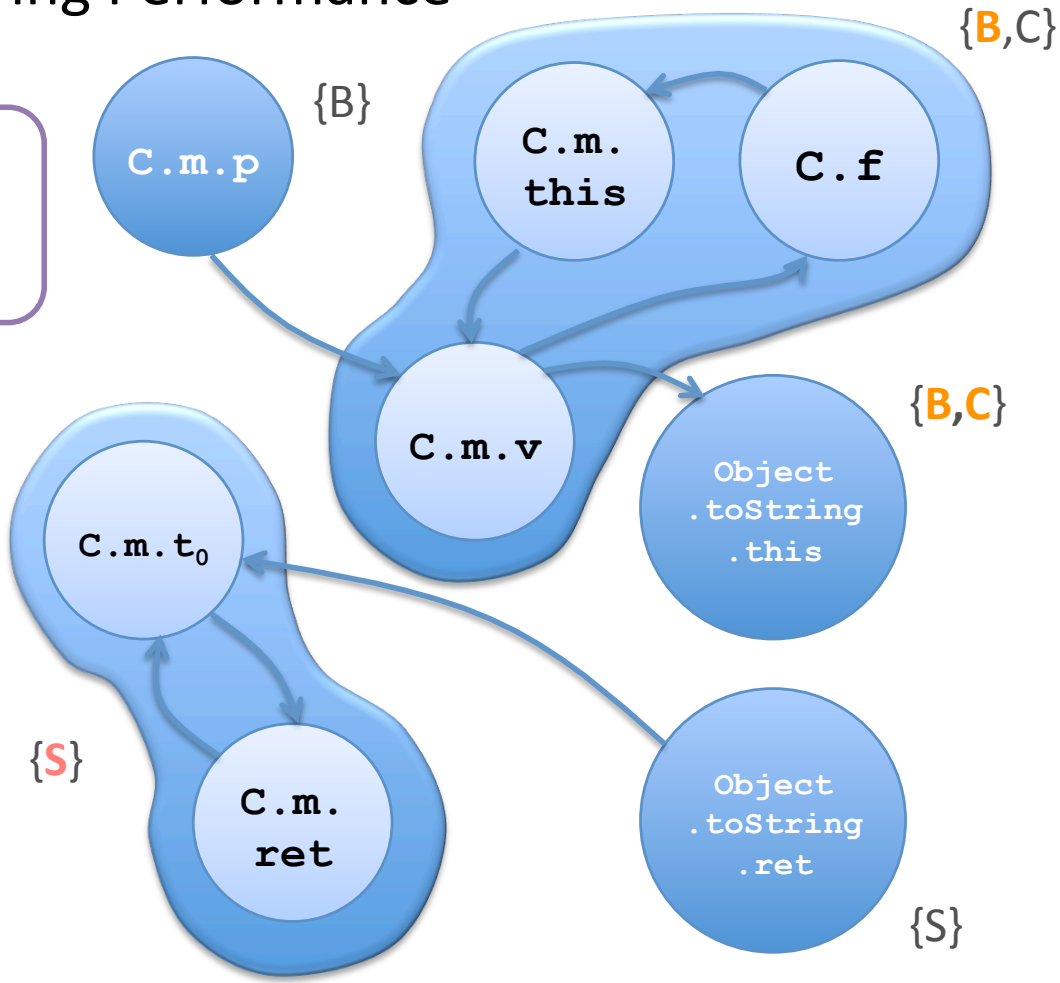
Find Strongly Connected Components
using Tarjan's algorithm



Variable Type Analysis

Improving Performance

2 **Propagate Types Along Edges**
(graph is now a DAG)



Declared-Type Analysis

- All variables of the same declared type are summarized as a single node
- Coarser-grain
  - Not as accurate as variable-type, but still more accurate than class hierarchy and rapid type
  - Faster and requires much less space

Experimental Results

- Detection of monomorphic call sites