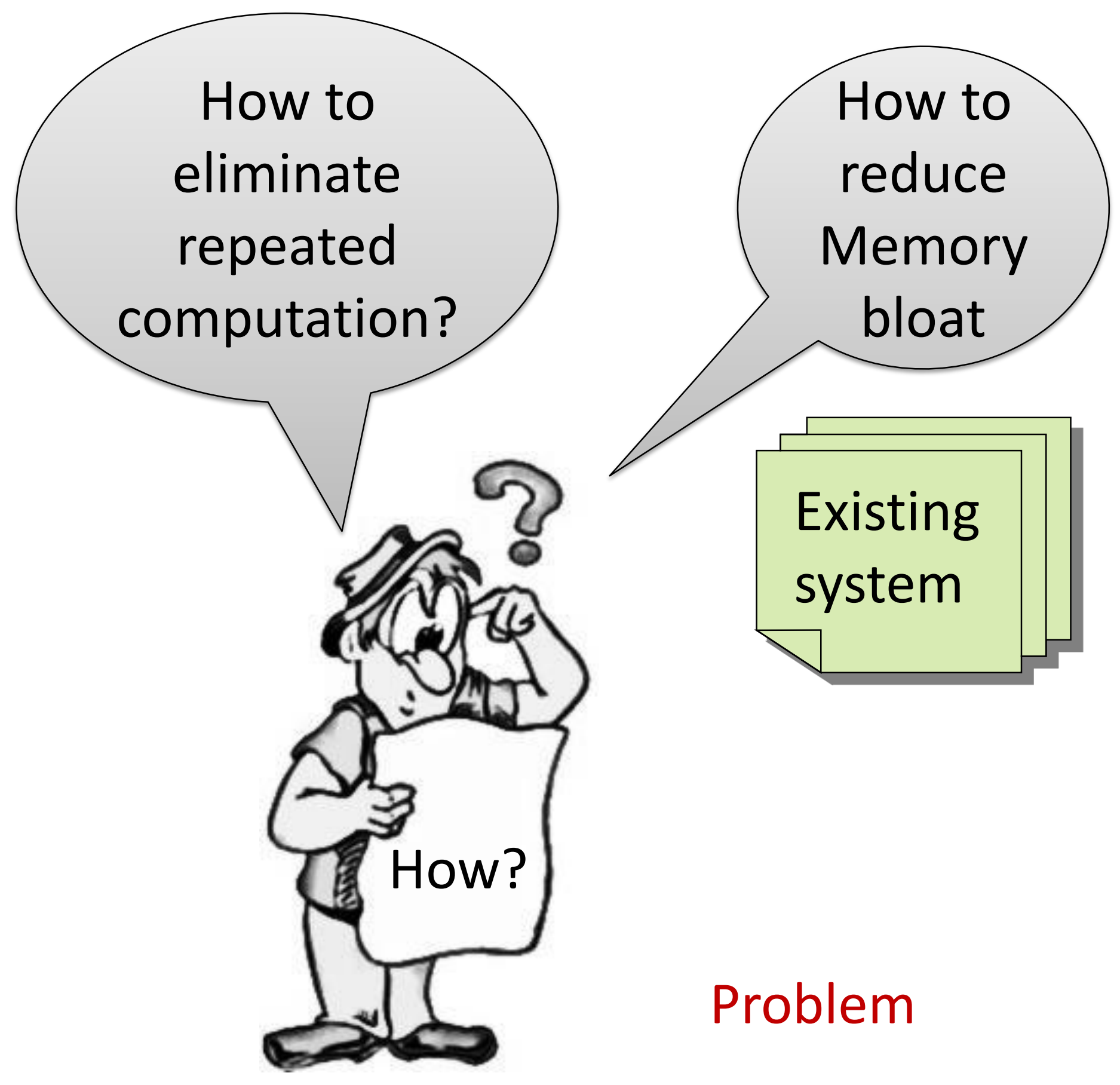
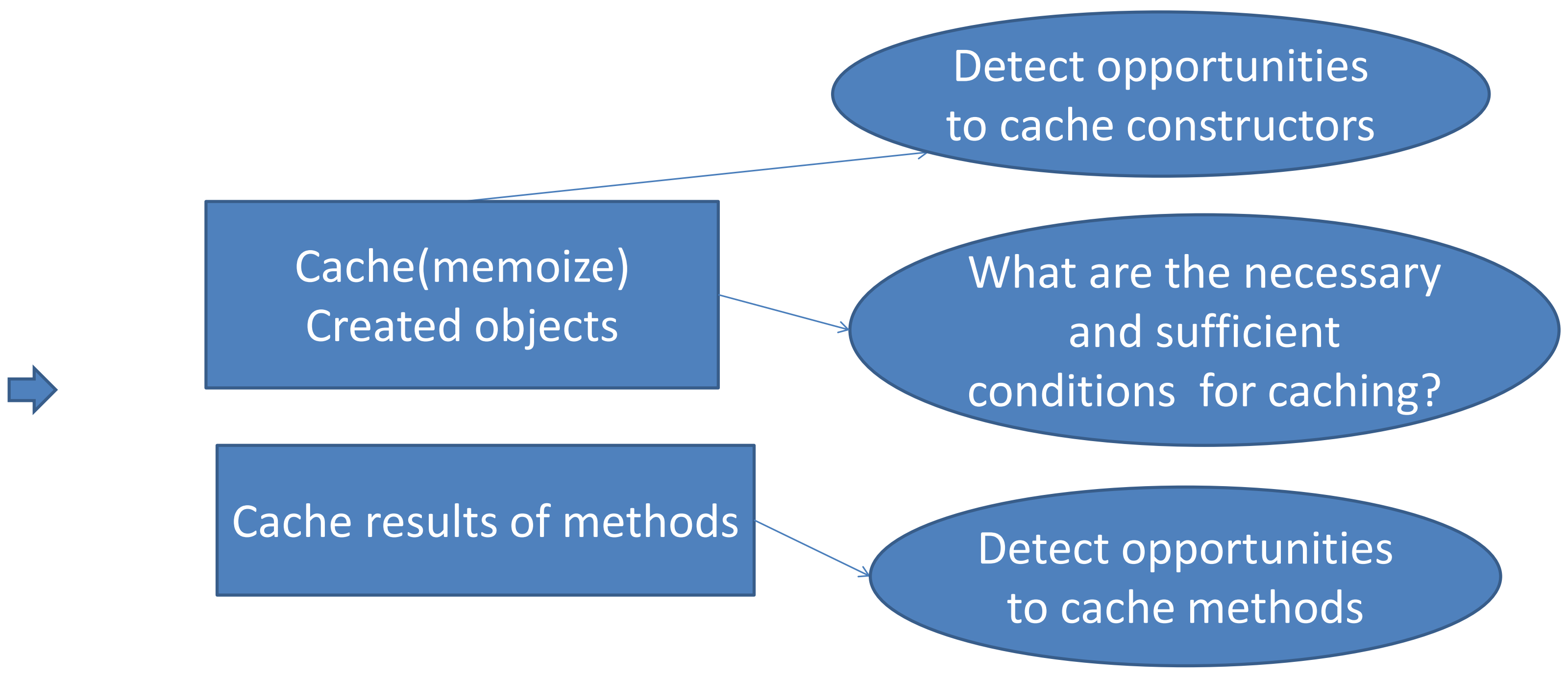
Program Analysis to Support Allocation Site based Refactorings

Girish Maskeri Rama

Advisor: Dr. K. V. Raghavan



Problem



Approach

Contribution 1: Detecting Opportunities for Object-sharing Refactoring

Problem Statement: Detect ***profitable*** allocation sites among thousands of allocation sites efficiently.

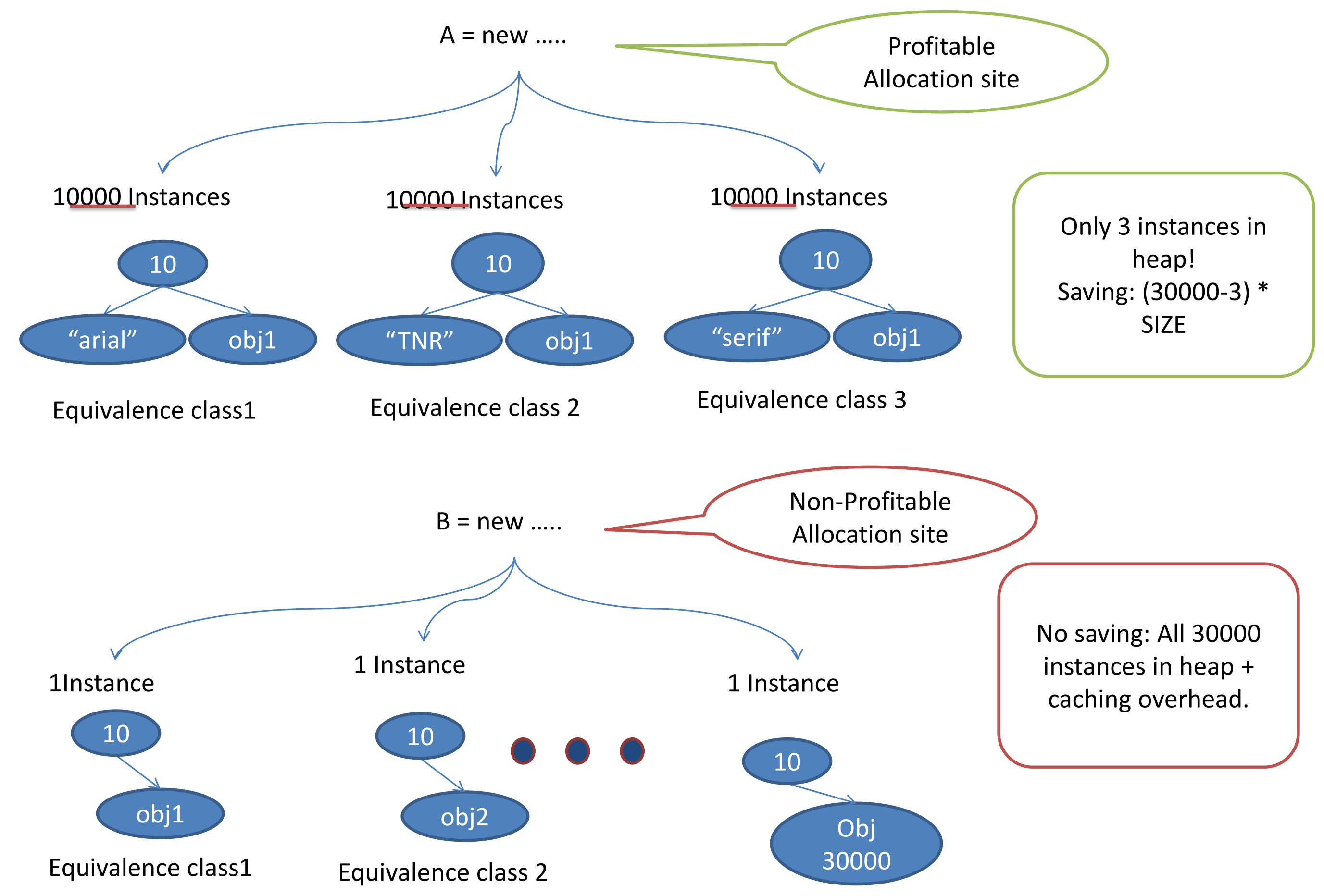
Our contribution: A dynamic analysis to detect opportunities for object-sharing refactoring.

- Computes Isomorphism equivalence classes of objects created at the selected sites.
- Reports estimated memory savings due to object-sharing refactoring

Publication: "A dynamic analysis to support object-sharing code refactorings ", Girish Maskeri Rama and Raghavan Komondoor, In Proceedings of the 29th ACM/IEEE international conference on Automated software engineering (ASE '14),2014.

Summary of results:

- 8 Dacapo systems + 2 systems (Apache FOP and PDFBox) with user introduced object sharing.
- Estimated savings (cumulative) varies 6% to 37% of tenured heap.
- Detected 10 out of 14 sites cached by the developer
- Identified a user introduced unsafe caching as actually unsafe.



Contribution 2: Static analysis to support allocation site refactorings

Problem Statement: identify program points where created object is fully initialized but has not yet escaped (If such a point exists).

Our contribution: A Static Analysis to Identify Location to Introduce Caching

For a given allocation site:

Identifies the last statement that mutates the created object across all paths from the allocation site to the end of the method.

If there exists a IVP from the last mutating statement to the statement where object escapes then no location exists.

If the site is in a loop and there exists a loop carried value flow

Soundness Guarantee : Across all runs of the program, all objects allocated at the allocation site are fully initialized at the caching location.

Summary of results: Memory reduction obtained in 5, 21, 6 and 5 sites for antlr, pmd, Apache fop and PDFbox respectively.

```
Class A{
  Bar foo(Object arg){
    for(int i; …..){
     Bar b =  new Bar(arg);
     b.attr1 = 5;
     global[i] = b
    }
    return b;
  }}
```

```
Class A{
HashMap<B,B> h = new HashMap<B,B>();
 Bar foo(Object arg){
   for(int i; …..){
     Bar b =  new Bar(arg);
     b.attr1 = 5;
      if(h.get(b) != null)
        b = h.get(b);
     else{
          h.put(b,b);
      }
     global[i] = b
   }
   return b;
 }}
```

Insert code *after* object is fully initialized but *before* it escapes!

Contribution 3: An improved scalable, iterative pointer analysis

Problem Statement: A static analysis approach to conservatively over-approximate flows of objects into object references, and hence identify object references that can potentially suffer runtime exceptions of certain kinds.

Our contribution: A scalable iterative form of points-to analysis that scales to large programs.

Initial iterations use inexpensive abstractions.

Later iterations use expensive and precise abstractions, but are targeted only at object references that did not get verified in the initial iterations.

Summary of results: For Sunflow, PMD and Chart, compared to the standard obj-sensitivity approach the reduction in running time is 75.5%, 39.6% and 8% respectively. Precision improvement is 0%, 2% and 9.4% respectively.

```
public class Main {
public A field1; //Let classes D and E be subclasses of A
public static void main(String[] args) {
  Main m1 = new Main(); Main m2 = new Main();
  C c1 = m1.meth(); C c2 = m2.meth();
  c1.field1 = new D(); c2.field1 = new  E();
  A v2 = (D) c1.field1;
  A v3 = (D) c2.field1;
}
private C meth(){ return new C(); }}
```

Safe downcast

Unsafe downcast

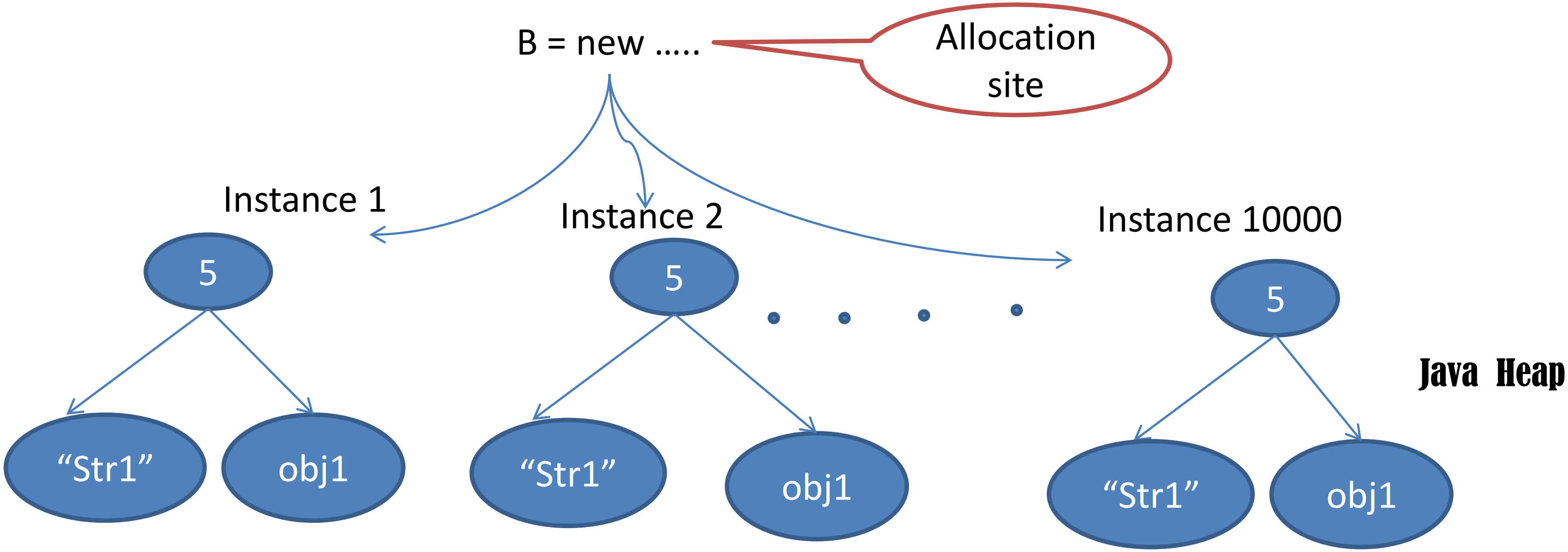
How to identify potentially unsafe downcast expressions?

Program Analysis to Support Allocation Site based Refactorings

Girish Maskeri Rama

Advisor: Dr. K V Raghavan

Problem: Runtime Memory Bloat due to Isomorphic Object Graphs



Object-sharing refactoring : Cache and share long-lived isomorphic objects
(instead of creating new objects every time)

Object-sharing refactoring : “Introduce Cache”

Pre

```
Class A{



    Bar foo(Object arg){




            return new Bar(arg);



    }
}
```

Creates isomorphic objects

Post

```
Class A{

   HashMap h = new HashMap();

   Bar  foo(Object arg){
     if(h.get(arg)  != null)
         return h.get(arg);
     else{
       bar b = new Bar(arg);
       h.put(arg,b);
       return b;
     }
   }
}
```

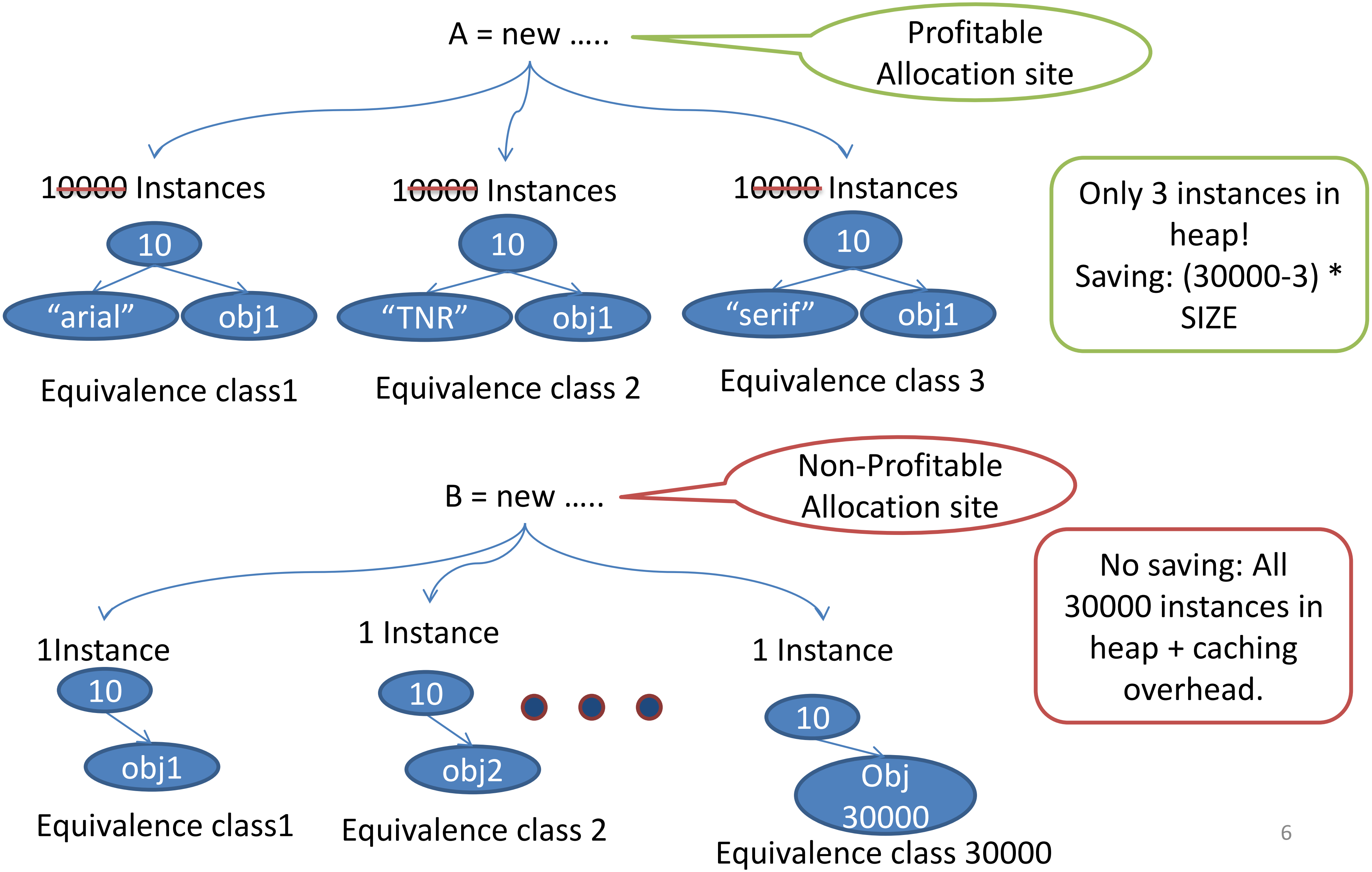
Thesis Overview

- An approach to identify, check safety and automate Object-sharing refactoring
- *Contribution 1*: Detecting Opportunities for Object-sharing Refactoring
- *Contribution 2*: Static analysis to support allocation site refactorings
- *Contribution 3*: An improved pointer analysis for checking ValueObject sites

Contributions 2 & 3 have applications other then object-sharing refactoring.

Contribution 1: Detecting Opportunities for Object-sharing Refactoring

Detecting Opportunities for Object-sharing Refactoring



A = new …..

Profitable Allocation site

~~10000~~ Instances

Equivalence class1

~~10000~~ Instances

Equivalence class 2

~~10000~~ Instances

Equivalence class 3

Only 3 instances in heap!
Saving: (30000-3) * SIZE

B = new …..

Non-Profitable Allocation site

1Instance

Equivalence class1

1 Instance

Equivalence class 2

1 Instance

Equivalence class 30000

No saving: All 30000 instances in heap + caching overhead.

Detecting Opportunities for Object-sharing Refactoring

Problem: Detect ***profitable*** allocation sites among thousands of allocation sites efficiently.

Our Contribution

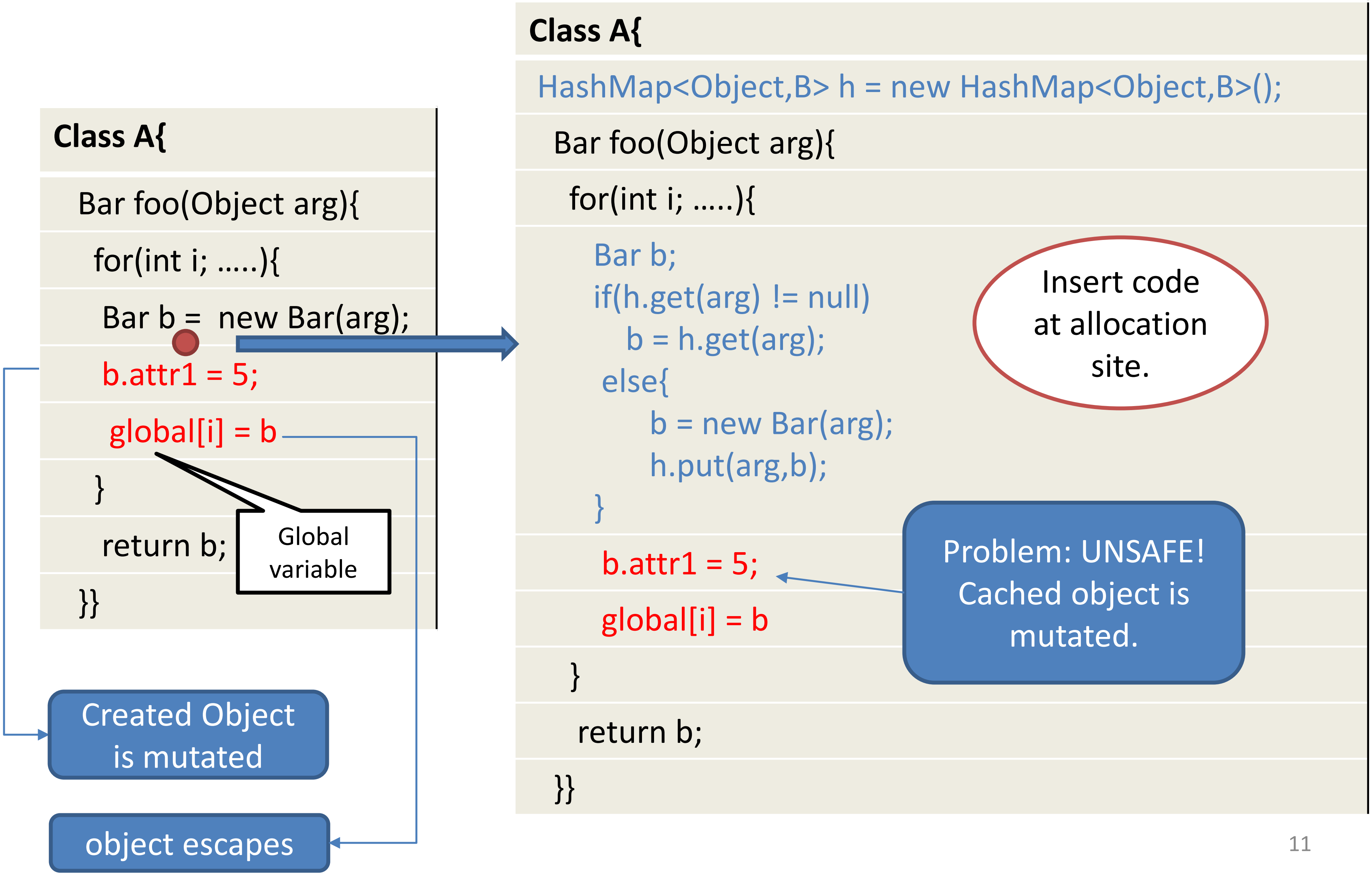
- A dynamic analysis to detect opportunities for object-sharing refactoring.
  - Computes Isomorphism equivalence classes of objects created at the selected sites.
  - Reports estimated memory savings due to object-sharing refactoring
- **Publication**: "A dynamic analysis to support object-sharing code refactorings ", Girish Maskeri Rama and Raghavan Komondoor, In Proceedings of the 29th ACM/IEEE international conference on Automated software engineering (ASE '14),2014.

Results

- 8 Dacapo systems + 2 systems (Apache FOP and PDFBox) with user introduced object sharing.
- Estimated savings (cumulative) varies
  - 6% to 37% of tenured heap.
- Detected 10 out of 14 sites cached by the developer
- Identified a user introduced unsafe caching as actually unsafe.

Contribution 2: Static analysis to support allocation site refactorings

Where to Insert Code for Object Sharing?



```
Class A{
  Bar foo(Object arg){
    for(int i; …..){
      Bar b =  new Bar(arg);
      b.attr1 = 5;
      global[i] = b
    }
    return b;
}}
```

Global variable

Created Object is mutated

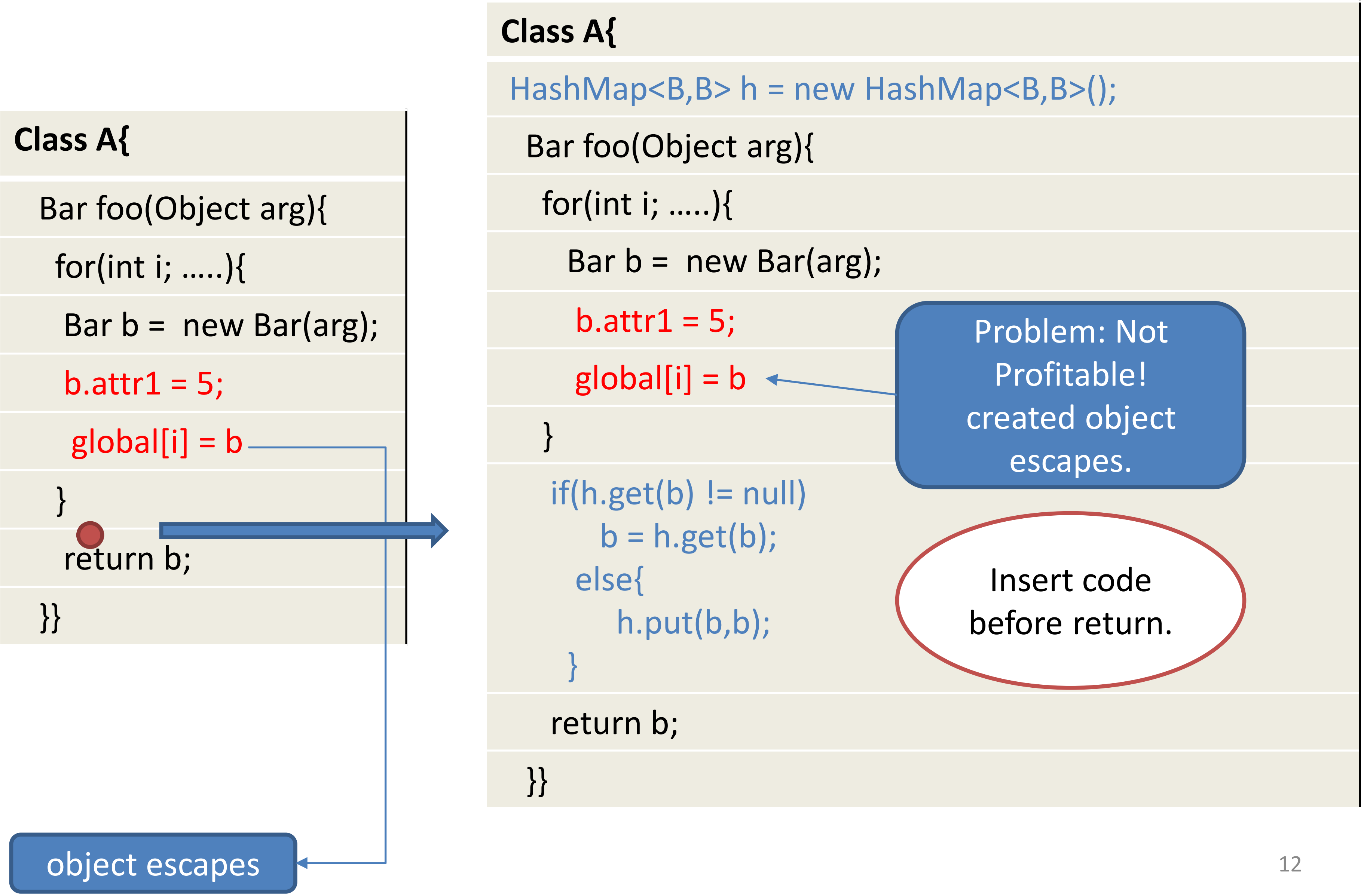
object escapes

```
Class A{
HashMap<Object,B> h = new HashMap<Object,B>();
  Bar foo(Object arg){
    for(int i; …..){
      Bar b;
      if(h.get(arg) != null)
        b = h.get(arg);
      else{
          b = new Bar(arg);
          h.put(arg,b);
      }
      b.attr1 = 5;
      global[i] = b
    }
    return b;
}}
```

Insert code at allocation site.

Problem: UNSAFE! Cached object is mutated.

Where to Insert Code for Object Sharing?



```
Class A{
  Bar foo(Object arg){
    for(int i; …..){
      Bar b =  new Bar(arg);
      b.attr1 = 5;
       global[i] = b
    }
      return b;
  }}
```

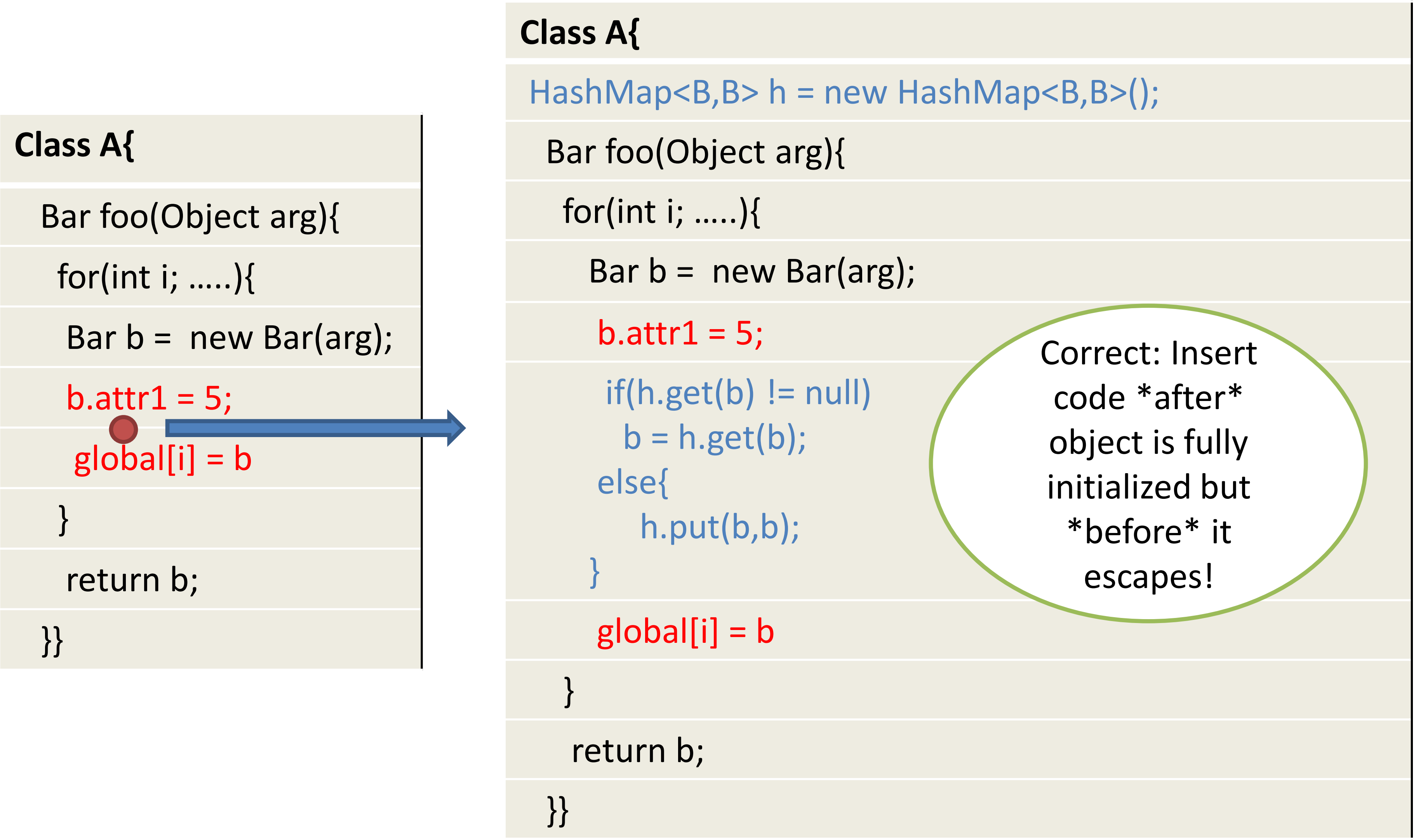
object escapes

```
Class A{
 HashMap<B,B> h = new HashMap<B,B>();
  Bar foo(Object arg){
    for(int i; …..){
      Bar b =  new Bar(arg);
       b.attr1 = 5;
       global[i] = b
    }
     if(h.get(b)  != null)
          b = h.get(b);
       else{
           h.put(b,b);
       }
     return b;
  }}
```

Problem: Not Profitable! created object escapes.

Insert code before return.

Where to Insert Code for Object Sharing?

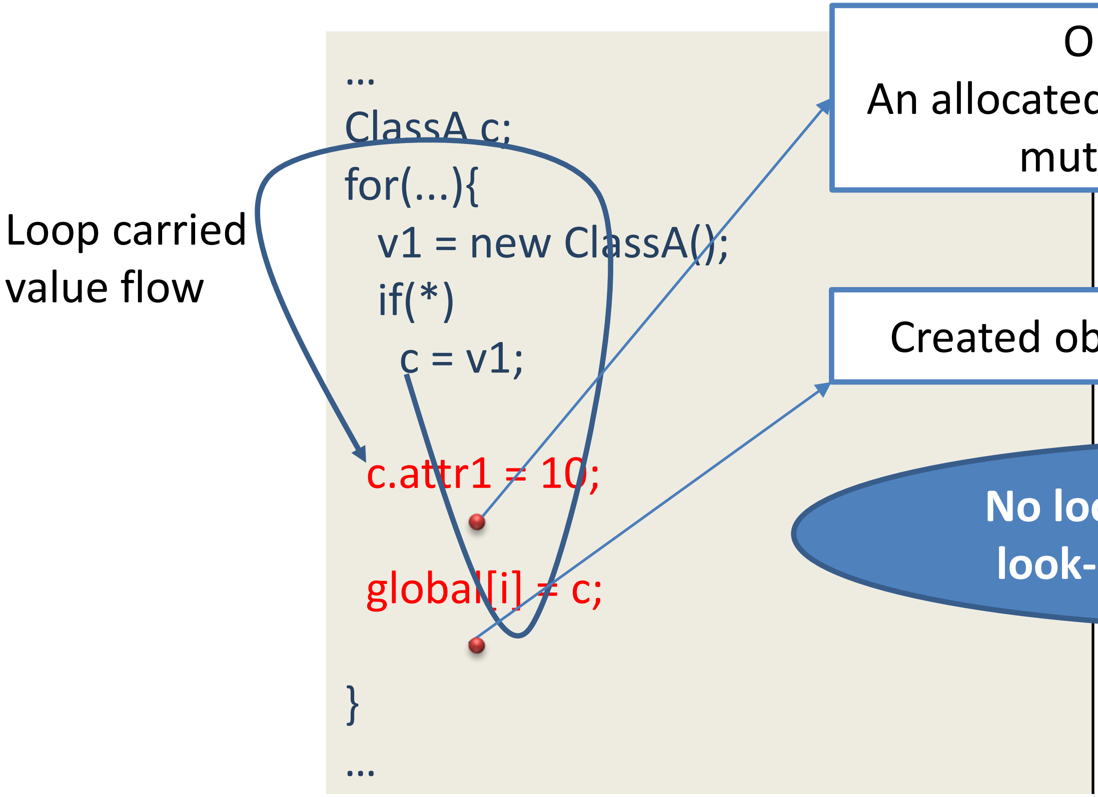


```
Class A{
  Bar foo(Object arg){
    for(int i; …..){
      Bar b =  new Bar(arg);
      b.attr1 = 5;
       global[i] = b
    }
      return b;
  }}
```

```
Class A{
 HashMap<B,B> h = new HashMap<B,B>();
   Bar foo(Object arg){
     for(int i; …..){
        Bar b =  new Bar(arg);
         b.attr1 = 5;
          if(h.get(b) != null)
            b = h.get(b);
         else{
              h.put(b,b);
         }
         global[i] = b
     }
      return b;
   }}
```

Correct: Insert code *after* object is fully initialized but *before* it escapes!

Where to Insert Code for Object Sharing?



Loop carried value flow

```
...
ClassA c;
for(...){
  v1 = new ClassA();
  if(*)
    c = v1;

  c.attr1 = 10;

  global[i] = c;

}
...
```

Object is not fully initialized.
An allocated object in current iteration could get mutated in subsequent iteration

Created object has escaped. So not profitable.

No location exists where cache look-up code can be inserted!

<u>Problem Statement</u>: identify program points where created object is fully initialized but has not yet escaped (If such a point exists).

Our Contribution: A Static Analysis to Identify Location to Introduce Caching

For a given allocation site:

- Identifies the last statement that mutates the created object across all paths from the allocation site to the end of the method.
- If there exists a IVP from the last mutating statement to the statement where object escapes then no location exists.
- If the site is in a loop and there exists a loop carried value flow

Soundness Guarantee : Across all runs of the program, all objects allocated at the allocation site are fully initialized at the caching location.

Other Applications of Finding Location where Object is Fully Initialized.

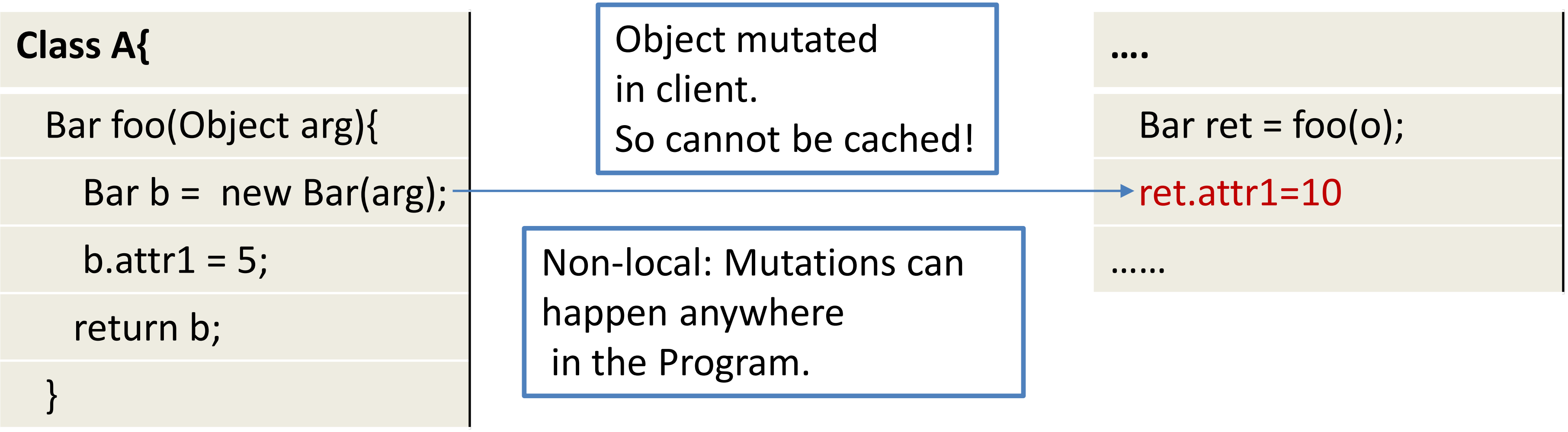
- Refactoring for immutability
- Refactoring to introduce factory method

Results

- Refactoring to introduce caching
  - Memory reduction obtained in 5, 21, 6 and 5 sites for antlr, pmd, Apache fop and PDFbox respectively.

Contribution 3: An Improved Pointer Analysis for checking ValueObject Sites

Object Mutations In Rest of Program can make Caching Unsafe



```
Class A{
  Bar foo(Object arg){
     Bar b =  new Bar(arg);
     b.attr1 = 5;
    return b;
  }
```

Object mutated
in client.
So cannot be cached!

```
....
  Bar ret = foo(o);
  ret.attr1=10
......
```

Non-local: Mutations can
happen anywhere
in the Program.

Objective: To identify whether in *any run* an object allocated at a site where caching is to be inserted is modified later in the program.

The Problem of Points-to Analysis

- Points-to analysis
  - The goal of **pointer analysis** is to compute an approximation of the set of symbolic objects that a **pointer** variable can refer to.
  - Well studied problem with rich literature.
  - In our application we use points-to analysis to determine if an object is mutated anywhere in the program.
  - Numerous other applications: Call graph, construction, Dependence analysis and optimization, Cast check elimination, Side effect analysis, Escape analysis, Slicing, Parallelization etc.
- Problem of Points-to Analysis
  - Precision depends on the extent of context-sensitivity employed (i.e. value of ‘K’)
  - Default object-sensitivity pointer analysis does not scale for ‘K’ > 2.

Our Contribution

- A scalable, client-driven, iterative form of points-to analysis that scales to large programs.
- Key idea:
  - Initial iterations use inexpensive abstractions (smaller values of ‘k’ but at larger number of sites).
  - Later iterations use expensive and precise abstractions. (i.e. Higher values of ‘k’)
  - But are targeted only at object references that did not get verified in the initial iterations.

Results

For Sunflow, Chart and PMD, compared to the existing standard objective-sensitivity approach with k=3, the reduction in running time is 40%, 25% and 18% respectively. precision of our approach is same as obj-sensitivity approach on each benchmark.

Thank You. Questions?