MICHIGAN ENGINEERING
UNIVERSITY OF MICHIGAN

Purity and Side Effect Analysis for Java Programs

Alexandru D. S˘alcianu and Martin C. Rinard

Presentation by

Group 19: Sathvika, Nivedhitha & Anthony

MICHIGAN ENGINEERING
UNIVERSITY OF MICHIGAN

Introduction / Motivation

how to analyze Java programs to figure out when methods are pure, when they have side effects, and how we can reason about them safely

Why does purity matter ?

- Purity enables formal verification techniques (JML specifications)
- Simplifies reasoning about method calls → referential transparency
- Critical for:
  - **Program verification** (proofs rely on stable state)
  - **Model checking** (reduces state explosion)
  - **Compiler optimizations** (LICM, CSE, parallelization)
  - **Understanding program behavior**, summarizing effects

The Challenge ?

- Real Java methods frequently allocate and mutate temporary objects
- Traditional purity: **no writes at all**, overly restrictive
- Need a definition that captures useful practical purity, not syntactic purity

Problem: Earlier Purity Analyses Are Too Strict

Limitations of Prior Work

- Based on syntactic restrictions
- Any heap write ⇒ method considered impure
- Calling a possibly impure method ⇒ entire method impure
- Cannot distinguish:
  - Writes to pre-existing objects
  - Writes to newly allocated helper objects

Real Java Examples That Break Old Purity Analysis

- Iterators mutate internal cursor/state
- Visitor patterns allocate new objects and write internal fields
- Methods that allocate temporary data structures (builder, accumulator)

MICHIGAN ENGINEERING
UNIVERSITY OF MICHIGAN

Key Contributions

MICHIGAN ENGINEERING
UNIVERSITY OF MICHIGAN

What this paper introduces

- First implemented purity analysis for Java that:
  - * allows modifying newly allocated objects,
  - * as long as they do not escape,
  - * and still treats the method as pure
- A unified static analysis combining:
  - * context-insensitive points-to
  - * escape analysis
  - * side-effect analysis
- Interprocedural purity inference using method summaries
- Scalable implementation + empirical results

Purity Definition & Categories

Pure

- Reads from existing objects
- Writes only to new, non-escaping objects
- Does not change any part of the pre-existing reachable state

Read-only

- Reads from existing objects
- Does NOT write to any object
- All effects are observational reads

Impure

- Writes to any object reachable before the method call
- Writes to any escaping object

java

```java
int square(int x) {
    return x * x;
}
```

java

```java
int size() {
    return this.length;
}
```

java

```java
void add(int x) {
    this.size = x;          // modifies object
}
```

How purity is computed ?

Points-to Analysis — “What objects can x point to?”

↓

Escape Analysis — Which of these objects escape the method?

↓

Side-Effect Analysis — Computes ReadSet, WriteSet & EscapeSet

↓

Method Effects Summary — (R, W, E)

↓

Purity Classification

Analysis

MICHIGAN ENGINEERING
UNIVERSITY OF MICHIGAN

Points-to Analysis

Definition: Determines which heap objects each variable or field may refer to.

- Flow-insensitive, context-insensitive
- Builds the program's abstract heap graph
- Maps variables/fields → allocation sites
- Identifies objects that reads/writes may target
- Basis for constructing R(m) and W(m)

```java
x = new Node();
y = x;
```

Escape Analysis

Definition: Determines whether a newly allocated object becomes visible outside the method.

- Determines whether objects created inside a method become visible outside
- If an object does not escape, writing to it does not break purity

Escape happens if object is:

- returned
- stored into a field of a reachable object
- passed as argument to a method where it may escape

Non-escaping objects can be safely mutated

Enables the paper’s relaxed purity definition

```
Node foo() {
    Node x = new Node();   // fresh
    return x;              // escapes!
}
```

Side-Effect Analysis

Definition: Computes all heap locations a method may read or write, using points-to + escape results.

Produces:

- R(m): objects/fields read
- W(m): objects/fields written
- E(m): allocated objects that escape

```java
void inc() {
    this.x++;
}
```

Interprocedurally propagated via method summaries

Final purity classification based on (R, W, E)

Method Summary

Method Summary = (R, W, E)

The analysis computes three sets:

- **ReadSet:** R(m) — set of object abstractions that method m may read
- **WriteSet:** W(m) — set of object abstractions method m may write
- **EscapeSet:** E(m) — set of allocated objects that escape m

Purity rule:

- Pure if: $W(m) \subseteq$ fresh objects AND E(m) is disjoint from W(m)
- Read-only if: $W(m) = \varnothing$
- Impure otherwise

```
Method foo():
  ReadSet = {obj.a}
  WriteSet = {}
  EscapeSet = {}
  => Read-only
```

How Do These Analyses Come Together Into One View?

MICHIGAN ENGINEERING
UNIVERSITY OF MICHIGAN

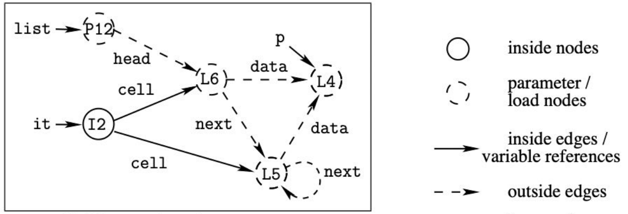
Points-To Graph (PTG)

A Points-To Graph (PTG) is an abstract heap model that summarizes all objects a method may interact with and how they are connected.

- Nodes represent abstract objects
- Edges represent abstract references
- Annotated with:
    - Allocation origin (inside vs. load)
    - Possible field targets
    - Escape status

MICHIGAN ENGINEERING
UNIVERSITY OF MICHIGAN

Points-To Graph (PTG)



a. Cell constructor

Node Types in Points-To Graph

- **Parameter Nodes**
  Represent object abstractions passed to the method
- **Load Nodes (L-nodes)**
  Represent objects reached via field loads from prestate objects → These are still part of the prestate
- **Inside Nodes (I-nodes)**
  Represent objects allocated by new inside the method

Edge Types

- **Outside Edges**: read from prestate (load nodes, parameter nodes)
- **Inside Edges**: created inside the method

Purity Determination via Prestate Reachability

Prestate Reachability

- Prestate = nodes reachable from parameter nodes via outside edges
- Compute reachable set Pre(m)
- Compute write set W(m)

Purity Rules

A method **m** is pure iff:

1. For all writes $w \in W(m)$:
   $target(w) \notin Pre(m)$
2. No modified inside node escapes:
   $W(m) \cap EscapingInsideNodes = \varnothing$

Example:

```
class List {
  Cell head = null;
  void add(Object e) {
    head = new Cell(e, head);
  }
  Iterator iterator() {
    return new ListItr(head);
  }
}

class Cell {
  Cell(Object d, Cell n) {
    data = d; next = n;
  }
  Object data;
  Cell   next;
}

interface Iterator {
  boolean hasNext();
  Object next();
}

class ListItr implements Iterator {
  ListItr(Cell head) {
    cell = head;
  }
  Cell cell;
  public boolean hasNext() {
    return cell != null;
  }
  public Object next() {
    Object result = cell.data;
    cell = cell.next;
    return result;
  }
}

class Point {
  Point(float x, float y) {
    this.x = x; this.y = y;
  }
  float x, y;
  void flip() {
    float t = x; x = y; y = t;
  }
}

class Main {
  static float sumX(List list) {
    float s = 0;
    Iterator it = list.iterator();
    while(it.hasNext()) {
      Point p = (Point) it.next();
      s += p.x;
    }
    return s;
  }

  static void flipAll(List list) {
    Iterator it = list.iterator();
    while(it.hasNext()) {
      Point p = (Point) it.next();
      p.flip();
    }
  }

  public static void main(String args[]) {
    List list = new List();
    list.add(new Point(1,2));
    list.add(new Point(2,3));
    sumX(list);
    flipAll(list);
  }
}
```

Example 1: sumX Is Pure

```
static float sumX(List list) {
  float s = 0;
  Iterator it = list.iterator();
  while(it.hasNext()) {
    Point p = (Point) it.next();
    s += p.x;
  }
  return s;
}
```

Purity Result:

✔ All writes target **inside node**
✔ No writes to **prestate**
→ **sumX = Pure Method**

Iterator creation:

- list.iterator() → allocates **ListItr** → **Inside Node (I-node)**
- Writes to iterator fields → **writes to I-node only**

Iterator next():

- cell = cell.next → moves internal pointer
- **Mutation stays inside I-node** (fresh object)
- No effect on list, cells, or points

Reads from list:

- Point p = (Point) it.next() → loads Point
- Point objects → **Prestate Nodes** (reachable from parameter)
- Only **read**, not written

No prestate writes:

- No modification to:
  • list (P-node)
  • Cell objects (L-nodes)
  • Point objects (prestate L-nodes)

MICHIGAN ENGINEERING
UNIVERSITY OF MICHIGAN

Example 1: sumX Is Pure

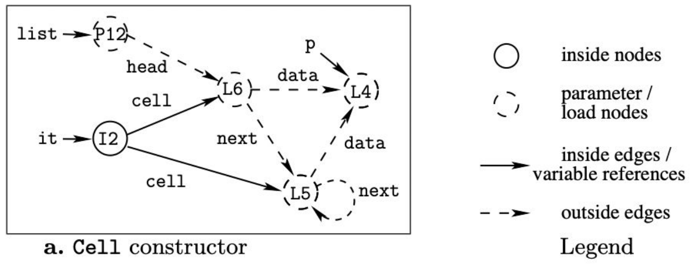


Fig. 2. Points-To Graph for the end of Main.sumX(List)

Purity Result:

✔ All writes target **inside node**
✔ No writes to **prestate**
→ **sumX = Pure Method**

Iterator creation:

- list.iterator() → allocates **ListItr** → **Inside Node (I-node)**
- Writes to iterator fields → **writes to I-node only**

Iterator next():

- cell = cell.next → moves internal pointer
- **Mutation stays inside I-node** (fresh object)
- No effect on list, cells, or points

Reads from list:

- Point p = (Point) it.next() → loads Point
- Point objects → **Prestate Nodes** (reachable from parameter)
- Only **read**, not written

No prestate writes:

- No modification to:
  • list (P-node)
  • Cell objects (L-nodes)
  • Point objects (prestate L-nodes)

MICHIGAN ENGINEERING
UNIVERSITY OF MICHIGAN

Example 2: `flipAll` Is Impure

```
static void flipAll(List list) {
  Iterator it = list.iterator();
  while(it.hasNext()) {
    Point p = (Point) it.next();
    p.flip();
  }
}
```

Purity Result:

✘ Writes target **prestate nodes**
✘ Mutates objects reachable from parameter

→ **flipAll = Impure Method**

Iterator creation:

- Same as sumX: new ListItr → Inside Node (I-node)

Loading points:

- Point p = (Point) it.next()
- Point objects → Prestate Nodes (from list.head)
- These existed before the call

flip():

- p.flip() writes: x = y; y = t;
- Direct write to Point.x and Point.y
- These fields belong to prestate nodes

Write path (from analysis):

- list.head.next*.data.(x | y)
- This path ends at prestate Point fields → outside/parameter nodes

Applications / Optimizations

Common Subexpression Elimination

```
a = pure_func(x, y);
b = pure_func(x, y);
```

→

```
a = pure_func(x, y);
b = a;
```

Loop-invariant Code Motion

```
for(...){
    y += pure_func(x)
}
```

→

```
temp = pure_func(x)
for(...){
    y += temp
}
```

Parallelization / Vectorization

```
int[100] arr;
for(int i = 0; i < 100; i++){
      Arr[i] = pure_func(i);
}
```

Results

MICHIGAN ENGINEERING
UNIVERSITY OF MICHIGAN

Benchmarks

Application	Description
BH	Barnes-Hut N-body solver
BiSort	Bitonic Sort
Em3d	Simulation of electromagnetic waves
Health	Health-care system simulation
MST	Bentley's algorithm for minimum spanning tree in a graph
Perimeter	Computes region perimeters in an image represented as a quad-tree
Power	Maximizes the economic efficiency of a community of power consumers
TSP	Randomized algorithm for the traveling salesman problem
TreeAdd	Recursive depth-first traversal of a tree to sum the node values
Voronoi	Voronoi diagram for random set of points

Table 1. Java Olden benchmark applications.

Results - 3.4 to 7.2 seconds

Application	All Methods		User Methods	
	count	% pure	count	% pure
BH	264	55%	59	47%
BiSort	214	57%	13	38%
Em3d	228	55%	20	40%
Health	231	57%	27	48%
MST	230	58%	31	54%
Perimeter	236	63%	37	89%
Power	224	53%	29	31%
TSP	220	56%	14	35%
TreeAdd	203	58%	5	40%
Voronoi	308	62%	70	71%

Table 2. Percentage of Pure Methods in the Java Olden benchmarks.

Strengths and weaknesses

Strengths

- The authors formalized a more relaxed version of purity and created a system to identify this definition of pure
- This relaxed definition of purity allows for more aggressive optimizations

Weaknesses

- The authors do not compare their results to other pureness analysis programs.
  - Number of functions identified as pure
  - Performance/speed of the analysis
- Their analysis requires analyzing all code paths which can be impossible for programs that utilize dynamic class loading or when the entire program is available

MICHIGAN ENGINEERING
UNIVERSITY OF MICHIGAN

Conclusion / Takeaways

Conclusion

- The authors create a more relaxed definition of what is pure
- It requires further analysis to identify this relaxed definition of a pure function, but allows more functions to be defined as pure
- This allows for more optimizations than the stricter definition of pure of no memory reads or writes.

Thank you
Q&A