

Programming Assignment: Pointer Analysis

Program Analysis Course

Professor: Minseok Jeon

Course: IC637

August 12, 2025

1 Overview

In this assignment, you will implement a flow-insensitive, context-insensitive pointer analysis for Java programs. The analysis is based on Andersen's algorithm and operates on intermediate facts extracted from Java bytecode using the Soot framework.

Your task is to implement nine key functions in the provided `analysis.py` file:

1. `process_alloc` : Handle object allocations
2. `process_move` : Handle variable assignments
3. `process_load` : Handle field loads
4. `process_store` : Handle field stores
5. `process_static_call` : Handle static method calls
6. `process_special_call` : Handle special method calls (constructors, super calls)
7. `process_virtual_call` : Handle virtual method calls
8. `process_param` : Handle interprocedural parameter assignments
9. `process_return` : Handle interprocedural return assignments

You can use auxiliary functions or libraries as needed, but the main logic should be implemented in the above functions.

2 Repository Structure

The pointer analysis framework is organized as follows:

```
HW1/
|-- analysis.py          # Main analysis implementation (YOUR TASK)
|-- main.py               # Pipeline orchestrator
|-- results.py            # Results storage and export
|-- requirements.txt       # Dependencies documentation
|-- Dockerfile             # Container setup
|-- bin/                   # Java bytecode processing tools
|-- frontend/              # Fact extraction pipeline
|-- util/                  # Utility modules for statistics and printing
|-- analysis_results/      # Final pointer analysis results
```

2.1 Key Files

- **analysis.py**: Your main implementation file. Contains the `PointerAnalysisAnalyzer` class with functions to implement.
- **frontend/read_facts.py**: Data structures for all fact types. Study this to understand the input data format.
- **benchmarks/**: Test programs to validate your implementation. Each subdirectory contains a specific test case.
- **main.py**: Pipeline script to run complete analysis. Use this to test your implementation.

2.2 Workflow

1. **Input**: Use provided .jar files in `benchmarks/` (e.g., `benchmarks/alloc/Alloc.jar`) or your own .jar file as input.
2. **Statement Extraction**: `JarStmtCollector.java` processes JAR → Jimple statements
Generated statements will be stored in `results/analysis_{pgm}/inputs/`
3. **Fact Extraction**: `frontend/extract_facts.py` converts statements → facts for pointer analysis
The generated facts will be stored in `results/analysis_{pgm}/facts/`
4. **Pointer Analysis**: Your `analysis.py` performs the actual pointer analysis
5. **Results**: Output written to `results/analysis_*/` with JSON, facts, and text reports

2.3 Workflow

Running analysis. You can run the analysis by running the following command:

```
1 python main.py <input jar> <options>
```

For instance, you can run the analysis on the `benchmarks/alloc/Alloc.jar` as follows:

```
1 python main.py benchmarks/alloc/Alloc.jar --verbose
```

The generated facts and analysis results will be stored in `results/analysis_Alloc/`.

3 Program Representation

Notations. We use the following notations.

- \mathbb{V} : set of variables
- \mathbb{H} : set of heap allocation sites
- \mathbb{M} : set of method signatures
- \mathbb{F} : set of field signatures
- \mathbb{I} : set of parameter indices in method calls
- \mathbb{S} : set of method names
- \mathbb{N} : natural numbers (indices)
- \mathbb{T} : class types

Program Representation. A Java program is represented as a set of facts, each of which is a tuple of the form:

$(var, heap, inMeth) \in \text{HEAPALLOCATION} \subseteq \mathbb{V} \times \mathbb{H} \times \mathbb{M}$	Heap allocation
$(to, from, inMeth) \in \text{MOVE} \subseteq \mathbb{V} \times \mathbb{V} \times \mathbb{M}$	Move instruction
$(to, from, fld, inMeth) \in \text{LOAD} \subseteq \mathbb{V} \times \mathbb{V} \times \mathbb{F} \times \mathbb{M}$	Load instruction
$(to, fld, from, inMeth) \in \text{STORE} \subseteq \mathbb{V} \times \mathbb{F} \times \mathbb{V} \times \mathbb{M}$	Store instruction
$(invo, toMeth, inMeth) \in \text{STATICCALL} \subseteq \mathbb{I} \times \mathbb{M} \times \mathbb{M}$	Static method call
$(invo, base, toMeth, inMeth) \in \text{SPECIALCALL} \subseteq \mathbb{I} \times \mathbb{V} \times \mathbb{M} \times \mathbb{M}$	Special method call
$(invo, base, name, inMeth) \in \text{VIRTUALCALL} \subseteq \mathbb{I} \times \mathbb{V} \times \mathbb{S} \times \mathbb{M}$	Virtual method call
$(heap, type) \in \text{HEAPALLOCATE} \subseteq \mathbb{H} \times \mathbb{T}$	Type of allocated heap
$(idx, invo, var) \in \text{ACTUALPARAM} \subseteq \mathbb{N} \times \mathbb{I} \times \mathbb{V}$	Parameter variable of call site
$(idx, meth, var) \in \text{FORMALPARAM} \subseteq \mathbb{N} \times \mathbb{M} \times \mathbb{V}$	Parameter variable of method
$(invo, var) \in \text{ASSIGNRETURNVALUE} \subseteq \mathbb{I} \times \mathbb{V}$	Variable taking return of method call
$(var, meth) \in \text{RETURNVAR} \subseteq \mathbb{V} \times \mathbb{M}$	Return variable of method
$(this, meth) \in \text{THISVAR} \subseteq \mathbb{V} \times \mathbb{M}$	This variable of method
$(meth, name, type) \in \text{METHNAMETYPE} \subseteq \mathbb{M} \times \mathbb{N} \times \mathbb{T}$	Method name and type
$(meth) \in \text{METHOD} \subseteq \mathbb{M}$	Method in the program

The relations will be stored as `*.fact` files in `results/analysis_{pgm}/facts/` directory. The generated facts are loaded through `frontend/read_facts.py`.

4 Pointer Analysis

The goal of pointer analysis is to compute the following three results (set of relations):

$varptsto \subseteq 2^{(\mathbb{V} \times \mathbb{H})}$	Variable points-to relations
$fldptsto \subseteq 2^{(\mathbb{H} \times \mathbb{F} \times \mathbb{H})}$	Field points-to relations
$callgraph \subseteq 2^{(\mathbb{I} \times \mathbb{M})}$	Call graph relations

The three relations start as follows:

$$varptsto \leftarrow \emptyset, \quad fldptsto \leftarrow \emptyset, \quad callgraph \leftarrow \{(_, <\text{mainMethod}>)\}$$

Variable points-to and field points-to relations starts as empty sets. The call graph starts with the main method (e.g., `<Alloc: void main(java.lang.String[])>`).

Analysis Rules. Pointer analyzer computes the analysis results by using the following nine analysis rules:

$$\frac{(var, heap, inMeth) \in \text{HEAPALLOCATION} \quad (_, inMeth) \in callgraph}{(var, heap) \in varptsto}$$

$$\frac{(to, from, inMeth) \in \text{MOVE} \quad (_, inMeth) \in callgraph \quad (from, heap) \in varptsto}{(to, heap) \in varptsto}$$

$$\frac{(to, from, fld, inMeth) \in \text{LOAD} \quad (_, inMeth) \in callgraph \quad (from, heap) \in varptsto \quad (heap, fld, heap') \in fldptsto}{(to, heap') \in varptsto}$$

$$\frac{(to, fld, from, inMeth) \in \text{STORE} \quad (_, inMeth) \in callgraph \quad (to, heap) \in varptsto \quad (from, heap') \in varptsto}{(heap, fld, heap') \in fldptsto}$$

$$\frac{(invo, toMeth, inMeth) \in \text{STATICCALL} \quad (_, inMeth) \in callgraph}{(invo, toMeth) \in callgraph}$$

$$\frac{(invo, base, toMeth, inMeth) \in \text{SPECIALCALL} \quad (_, inMeth) \in callgraph \quad (this, toMeth) \in \text{THISVAR} \quad (base, heap) \in varptsto}{(invo, toMeth) \in callgraph \quad (this, heap) \in varptsto}$$

$$\frac{(invo, base, name, inMeth) \in \text{VIRTUALCALL} \quad (_, inMeth) \in callgraph \quad (base, heap) \in varptsto \quad (heap, type) \in \text{HEAPALLOCATE} \quad (toMeth, name, type) \in \text{METHNAMETYPE} \quad (toMeth, this) \in \text{THISVAR}}{(invo, toMeth) \in callgraph \quad (this, heap) \in varptsto}$$

$$\frac{(invo, meth) \in callgraph \quad (idx, param, invo) \in \text{ACTUALPARAM} \quad (idx, param', meth) \in \text{FORMALPARAM} \quad (param, heap) \in varptsto}{(param', heap) \in varptsto}$$

$$\frac{(invo, meth) \in callgraph \quad (return, invo) \in \text{ASSIGNRETURNVALUE} \quad (return', meth) \in \text{RETURNVAR} \quad (return', heap) \in varptsto}{(return, heap) \in varptsto}$$