Andersen's Points-to Analysis
Constraint Graph, Analysis Rules and Solving

COMP6131 UNSW

1. Overview of Andersen's Points-to Analysis

Andersen's inclusion-based points-to analysis is a static analysis technique used to determine which memory objects a pointer variable may point to at runtime. It generates and solves set inclusion constraints on a constraint graph to determine the points-to sets of variables.

The analysis is flow-insensitive, meaning it ignores the control-flow order of program statements, and context-insensitive, meaning it does not distinguish between different function invocation contexts. It is inclusion-based, which allows more precise modelling than unification-based approaches by preserving subset relationships among points-to sets. For more details, see the wiki link here: Pointer Analysis.

2. Motivations and Use Cases

Points-to analysis serves as a foundational building block in many advanced program analysis and compiler optimisation tasks. Andersen's analysis, in particular, offers a good balance between precision and scalability. One key motivation is to provide reasonable accuracy in alias analysis compared to simpler approaches. By tracking inclusion relationships, the analysis can disambiguate pointers that other analyses might conservatively treat as aliases.

Andersen's analysis can support a wide range of compiler optimisations, such as enabling more aggressive constant propagation, dead code elimination, and safe instruction reordering. It is also essential in security analysis, where understanding aliasing helps detect memory safety violations, such as taint analysis and buffer overflows, which are introduced in this course. In software verification and program analysis tools, Andersen's analysis provides the underlying pointer information needed to reason about memory safety properties and data structure invariants. Moreover, refactoring tools can use points-to information to safely transform pointer-heavy legacy code.

3. Constraint Graph and Analysis Rules

We explain the four key rules of Andersen's analysis using constraint graphs. In these graphs, nodes represent pointer variables or memory objects/variables, and directed edges represent constraints between program variables.

Table 1 summarizes the four key rules in Andersen's inclusion-based points-to analysis. Each rule is presented with its C-like syntax, corresponding constraint form, the solving rule applied to update points-to sets, and a brief explanation of its effect on the constraint graph. These rules define how pointer assignments and dereferences are translated into graph edges and how they propagate points-to information. Specifically, the rules cover address-of assignments (ADDR), direct pointer

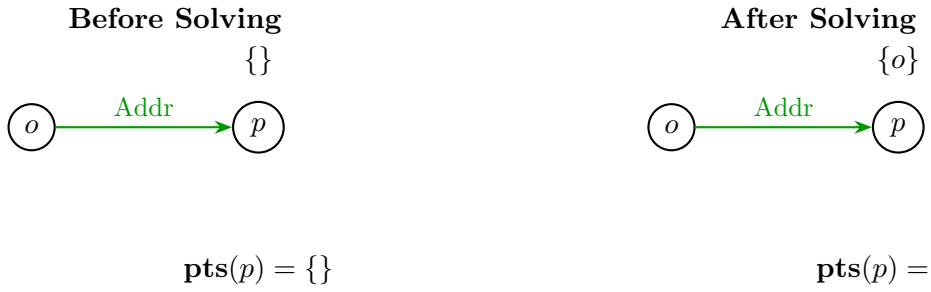
Table 1: Summary of Andersen's Constraint Rules

C-like	Constraint Edge	Solving rule	Explanation
`p = &o`	$o \xrightarrow{\text{Addr}} p$	$\text{pts}(p) = \text{pts}(p) \cup \{o\}$	add o into p's points-to set
`q = p`	$p \xrightarrow{\text{Copy}} q$	$\text{pts}(q) = \text{pts}(q) \cup \text{pts}(p)$	union p's points-to set into q's one
`q = *p`	$p \xrightarrow{\text{Load}} q$	for each $o \in \text{pts}(p)$: add $o \xrightarrow{\text{Copy}} q$	for each o in p's points-to set, add a COPY edge from o to q (if it is not on the graph)
`*p = q`	$q \xrightarrow{\text{Store}} p$	for each $o \in \text{pts}(p)$: add $q \xrightarrow{\text{Copy}} o$	for each o in p's points-to set, add a COPY edge from q to o (if it is not on the graph)

copies (COPY), memory loads via dereferencing (LOAD), and memory stores (STORE), forming the foundation of Andersen's constraint-solving process.

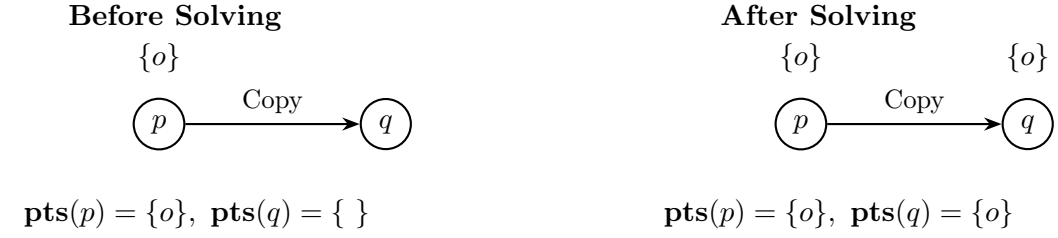
Rule 1 (ADDR): `p = &o`

This statement introduces an address-of operation (memory allocation). The constraint form is $o \xrightarrow{\text{Addr}} p$, and the solving rule is to update $\text{pts}(p) = \text{pts}(p) \cup \{o\}$. In the constraint graph, this corresponds to adding o into p's points-to set using an Addr edge from o to p.



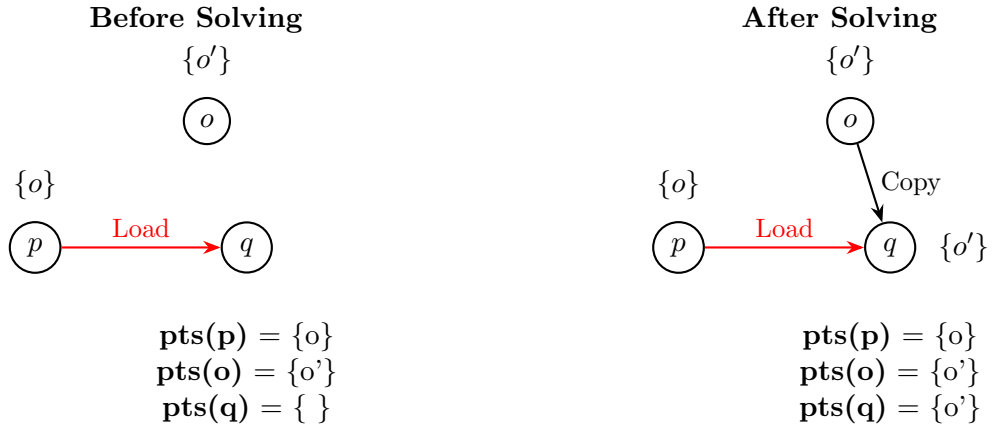
Rule 2 (COPY): `q = p`

This rule represents a direct copy of one pointer to another. The constraint form is $p \xrightarrow{\text{Copy}} q$, and the solving rule is to propagate points-to set of p to that of q, i.e., merge $\text{pts}(p)$ into $\text{pts}(q)$.



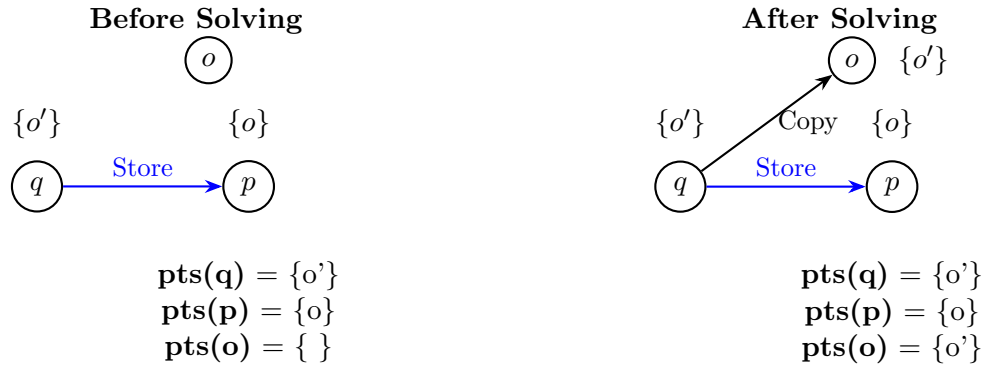
Rule 3 (LOAD): `q = *p`

This dereference operation loads a value from memory. The constraint form is $p \xrightarrow{\text{Load}} q$. For each $o \in \text{pts}(p)$, we add $o \xrightarrow{\text{Copy}} q$ to the graph, and propagate the points-to set of o to that of q.



Rule 4 (STORE): `*p = q`

This store operation writes a pointer value to a location. The constraint form is $q \xrightarrow{\text{Store}} p$. For each $o \in \text{pts}(p)$, we add $q \xrightarrow{\text{Copy}} o$ to the graph, and propagate the points-to set of q to that of o.



4. Constraint Solving Algorithm

The solving phase of Andersen's analysis involves propagating points-to information across the constraint graph using a fixed-point algorithm (i.e., involves iteratively applying the analysis rules to an initial graph until the points-to set of each variable remains unchanged). Each of the four constraint rules contributes a specific edge type or points-to relation that must be resolved iteratively until no more changes occur. **A formal algorithm for constraint solving can be found in Figure 1 of this paper**: The Ant and the Grasshopper: Fast and Accurate Pointer Analysis for Millions of Lines of Code (PLDI'07)

One algorithm (can be different from the above paper) proceeds as follows:

1. Initialize the constraint graph with all edges corresponding to the program's pointer assignments based on the four rules: ADDR, COPY, LOAD, and STORE.
2. Handle all ADDR edges by initialising the corresponding pointers and adding them to the worklist; then maintain the worklist with variables whose points-to sets have changed.
3. While the worklist is not empty:
   - Remove a variable x from the worklist.
   - For each outgoing edge from x:

– If it is a COPY edge to y, update $\text{pts}(y) = \text{pts}(y) \cup \text{pts}(x)$. If $\text{pts}(y)$ changes, add y to the worklist.
– If it is a LOAD edge to y, for each $o \in \text{pts}(x)$, update $\text{pts}(y) = \text{pts}(y) \cup \text{pts}(o)$. If $\text{pts}(y)$ changes, add y to the worklist.
– If it is a STORE edge from y, for each $o \in \text{pts}(x)$, update $\text{pts}(o) = \text{pts}(o) \cup \text{pts}(y)$. If $\text{pts}(o)$ changes, add o to the worklist.

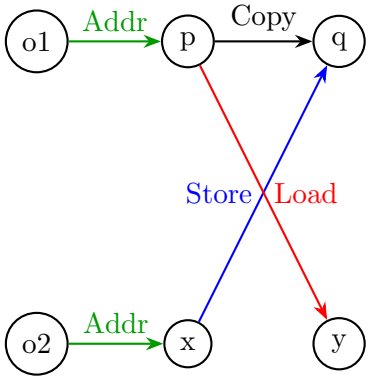
5. Illustrative Example: Applying All Four Rules

To consolidate understanding, we walk through a pseudo-code example below using all four rules.

```
void main() {
    int o1, o2; int *p, *q, *x, *y;
    p = &o1;      // Rule 1: ADDR
    x = &o2;      // Rule 1: ADDR
    q = p;        // Rule 2: COPY
    *q = x;       // Rule 4: STORE
    y = *p;       // Rule 3: LOAD
}
```

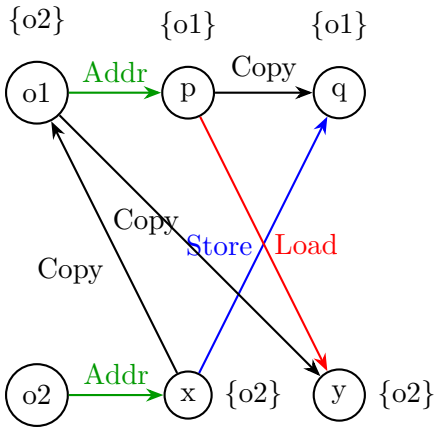
Constraint Graphs Before and After Solving

Before Solving



Points-to Set Before
$\text{pts}(p) = \{\}$
$\text{pts}(q) = \{\}$
$\text{pts}(x) = \{\}$
$\text{pts}(y) = \{\}$
$\text{pts}(o1) = \{\}$
$\text{pts}(o2) = \{\}$

After Solving



Points-to Set After
$\text{pts}(p) = \{o1\}$
$\text{pts}(q) = \{o1\}$
$\text{pts}(x) = \{o2\}$
$\text{pts}(y) = \{o2\}$
$\text{pts}(o1) = \{o2\}$
$\text{pts}(o2) = \{\}$

5. Final Remarks

Andersen's analysis is a fundamental inclusion-based points-to analysis with strong practical relevance in compiler optimisations, program verification, and security analysis. Using constraint

graphs allows us to visualise how memory locations propagate through pointer assignments, making it easier to understand and implement aliasing information.