Pointer Analysis

Mayur Naik
CIS 573 – Fall 2019
Introducing Pointers

Example without pointers

```
x = 1;
y = x;
assert(y == 1)
```

Example with pointers

```
x = new Circle();
x.radius = 1;
y = x.radius;
assert(y == 1)
```
Introducing Pointers

Example without pointers

[x == 1]  
\[x = 1;\]
\[y = x;\]
[y == 1]  
\[\text{assert}(y == 1)\]

Same example with pointers

x = new Circle();
x.radius = 1;
[x.radius == 1]  
\[y = x.radius;\]
[y == 1]  
\[\text{assert}(y == 1)\]
• Situation in which same address referred to in different ways

```java
Circle x = new Circle();
x.radius = 1;
y = x.radius;
assert(y == 1)
```

```java
Circle z = ?
x.radius = 1;
z.radius = 2;
y = x.radius;
assert(y == 1)
```
May-Alias Analysis

Circle x = new Circle();
Circle z = new Circle();
x.radius = 1;
z.radius = 2;
y = x.radius;
assert(y == 1)
Must-Alias Analysis

- May-Alias and Must-Alias are dual problems
- Must-Alias more advanced, less useful in practice
- Focus of this Lesson: May-Alias Analysis

```java
Circle x = new Circle();
Circle z = x;
x.radius = 1;
z.radius = 2;
y = x.radius;
assert(y == 1)  y == 2
```
Why Is Pointer Analysis Hard?

```java
class Node {
    int data;
    Node next, prev;
}

Node h = null;
for (...) {
    Node v = new Node();
    if (h != null) {
        v.next = h;
        h.prev = v;
    }
    h = v;
}
```

And many more ...

h.data
h.next.prev.data
h.next.next.prev.prev.prev.data
h.next.prev.next.prev.data
Approximation to the Rescue

- Pointer analysis problem is undecidable

=> We must sacrifice some combination of: Soundness, Completeness, Termination

- We are going to sacrifice completeness

=> False positives but no false negatives
What False Positives Mean

Pointer analysis answers questions of form: MayAlias(x, z)?

No  => x and z are not aliased in any run

Yes => Can’t tell if x and z are aliased in some run

Circle x = new Circle();
Circle z = new Circle();

x.radius = 1;
z.radius = 2;

y = x.radius;
assert(y == 1)

False Positive!

[x != z]
[x.radius == 1, x != z]
[x.radius == 1]
[y == 1]
Approximation to the Rescue

• Many sound approximate algorithms for pointer analysis

• Varying levels of precision

• Differ in two key aspects:
  – How to abstract the heap (i.e. dynamically allocated data)
  – How to abstract control-flow
Example Java Program

class Elevator {
    Object[] floors;
    Object[] events;
}

void doit(int M, int N) {
    Elevator v = new Elevator();
    v.floors = new Object[M];
    v.events = new Object[N];

    for (int i = 0; i < M; i++) {
        Floor f = new Floor();
        v.floors[i] = f;
    }

    for (int i = 0; i < N; i++) {
        Event e = new Event();
        v.events[i] = e;
    }
}
void doit(int M, int N) {
    Elevator v = new Elevator();
    v.floors = new Object[M];
    v.events = new Object[N];
    for (int i = 0; i < M; i++) {
        Floor f = new Floor();
        v.floors[i] = f;
    }
    for (int i = 0; i < N; i++) {
        Event e = new Event();
        v.events[i] = e;
    }
}
Abstracting the Heap

```java
void doit(int M, int N) {
    Elevator v = new Elevator();
    v.floors = new Object[M];
    v.events = new Object[N];

    for (int i = 0; i < M; i++) {
        Floor f = new Floor();
        v.floors[i] = f;
    }

    for (int i = 0; i < N; i++) {
        Event e = new Event();
        v.events[i] = e;
    }
}
```
Result of Heap Abstraction: Points-to Graph
void doit(int M, int N) {
    Elevator v = new Elevator();
    v.floors = new Object[M];
    v.events = new Object[N];
    for (int i = 0; i < M; i++) {
        Floor f = new Floor();
        v.floors[i] = f;
    }
    for (int i = 0; i < N; i++) {
        Event e = new Event();
        v.events[i] = e;
    }
}
Flow Insensitivity

```java
void doit(int M, int N) {
    Elevator v = new Elevator();

    v.floors = new Object[M];
    v.events = new Object[N];

    for (int i = 0; i < M; i++) {
        Floor f = new Floor();
        v.floors[i] = f;
    }

    for (int i = 0; i < N; i++) {
        Event e = new Event();
        v.events[i] = e;
    }
}
```

```java
void doit(int M, int N) {
    v = new Elevator

    v.floors = new Object[]
    v.events = new Object[]

    f = new Floor
    v.floors[*] = f

    e = new Event
    v.events[*] = e
}
```
Chaotic Iteration Algorithm

graph = empty
repeat:
    for (each statement $s$ in set)
        apply rule corresponding to $s$ on graph
until graph stops changing
Kinds of Statements

(statement) $s ::= v = \text{new} \ldots \mid v = v2 \mid v2 = v.f \mid v.f = v2 \mid v2 = v[*] \mid v[*] = v2$

(pointer-type variable) $v$

(pointer-type field) $f$
Is This Grammar Enough?

\[
v = \text{new } \ldots \quad | \quad v = v2 \quad | \quad v2 = v.f \quad | \\
v.f = v2 \quad | \quad v2 = v[*] \quad | \quad v[*] = v2
\]

\[
v\text{.events} = \text{new } \text{Object}[] \quad \rightarrow \quad \text{tmp} = \text{new } \text{Object}[] \\
v\text{.events}[*] = e
\]

\[
tmp = v\text{.events} \\
tmp[*] = e
\]
Example Program in Normal Form

void doit(int M, int N) {
    v = new Elevator
    v.floors = new Object[]
    v.events = new Object[]

    f = new Floor
    v.floors[*] = f

    e = new Event
    v.events[*] = e
}

void doit(int M, int N) {
    v = new Elevator
    tmp1 = new Object[]
    v.floors = tmp1
    tmp2 = new Object[]
    v.events = tmp2

    f = new Floor
    tmp3 = v.floors
    tmp3[*] = f

    e = new Event
    tmp4 = v.events
    tmp4[*] = e
}
QUIZ: Normal Form of Programs

\[ v = \text{new ...} \quad | \quad v = v2 \quad | \quad v2 = v.f \quad | \quad v.f = v2 \quad | \quad v2 = v[\ast] \quad | \quad v[\ast] = v2 \]

Convert each of these two expressions to normal form:

\[ v1.f = v2.f \quad \quad \rightarrow \quad \quad \]

\[ v1.f.g = v2.h \quad \quad \rightarrow \quad \quad \]
QUIZ: Normal Form of Programs

\[
\begin{align*}
v &= \text{new} \ldots & v &= v2 & v2 &= v.f \\
v.f &= v2 & v2 &= v[*] & v[*] &= v2
\end{align*}
\]

Convert each of these two expressions to normal form:

\[
\begin{align*}
v1.f &= v2.f & \text{tmp} &= v2.f \\
v1.f.g &= v2.h & v1.f &= \text{tmp} \\
\end{align*}
\]
Rule for Object Allocation Sites

Before: \( v = \text{new } B \)
Rule for Object Allocation Sites: Example

```java
void doit(int M, int N) {
    Elevator v = new Elevator
    Object[] tmp1 = new Object[M]
    v.floors = tmp1
    Object[] tmp2 = new Object[N]
    v.events = tmp2

    Floor f = new Floor
    v.floors[tmp3[*]] = f

    Event e = new Event
    v.events[tmp4[*]] = e
}
```
Rule for Object Copy

Before:

\[ v_1 = v_2 \]

After:

\[ v_1 = v_2 \]
Rule for Field Writes

Before:  \[v_1 \rightarrow A \quad v_2 \rightarrow B\]  
\[v_1 \cdot f = v_2\]
\[\text{or}\]
\[v_1[\ast] = v_2\]

After:  \[v_1 \rightarrow A \quad v_2 \rightarrow B\]  
\[v_1 \rightarrow A \quad f \text{ or } [\ast] \rightarrow C\]  
\[f \text{ or } [\ast] \rightarrow C\]
Rule for Field Writes: Example

```java
void doit(int M, int N) {
    Elevator v = new Elevator
    Object[] tmp1 = new Object[]
    v.floors = tmp1
    Object[] tmp2 = new Object[]
    v.events = tmp2
    Floor f = new Floor
    tmp3 = v.floors
    tmp3[*] = f
    Event e = new Event
    tmp4 = v.events
    tmp4[*] = e
}
```
Rule for Field Reads

Before: 

\[ v1 = v2.f \] 
or 
\[ v1 = v2[\ast] \]

After: 

\[ v1 \]
void doit(int M, int N) {
    v = new Elevator
    tmp1 = new Object[]
    v.floors = tmp1
    tmp2 = new Object[]
    v.events = tmp2
    f = new Floor
    tmp3 = v.floors
    tmp3[*] = f
    e = new Event
    tmp4 = v.events
    tmp4[*] = e
}
void doit(int M, int N) {
    v = new Elevator
    tmp1 = new Object[]
    v.floors = tmp1
    tmp2 = new Object[]
    v.events = tmp2
    f = new Floor
    tmp3 = v.floors
    tmp3[*] = f
    e = new Event
    tmp4 = v.events
    tmp4[*] = e
}
class Node {
    int data;
    Node next, prev;
}

Node h = null;
for (...) {
    Node v = new Node();
    if (h != null) {
        v.next = h;
        h.prev = v;
    }
    h = v;
}
QUIZ: Pointer Analysis Example

class Node {
    int data;
    Node next, prev;
}

Node h = null;
for (...) {
    Node v = new Node();
    if (h != null) {
        v.next = h;
        v.prev = h.prev = v;
    }
    h = v;
}
Classifying Pointer Analysis Algorithms

• Is it flow-sensitive?

• Is it context-sensitive?

• What heap abstraction scheme is used?

• How are aggregate data types modeled?
Flow Sensitivity

- How to model control-flow **within** a procedure

- Two kinds: flow-insensitive vs. flow-sensitive

  - Flow-insensitive == **weak updates**
    - Suffices for may-alias analysis

  - Flow-sensitive == **strong updates**
    - Required for must-alias analysis
Context Sensitivity

• How to model control-flow **across** procedures

• Two kinds: context-insensitive vs. context-sensitive

• Context-insensitive: analyze each procedure once

• Context-sensitive: analyze each procedure possibly multiple times, once per abstract calling context
Heap Abstraction

• Scheme to partition unbounded set of concrete objects into finitely many abstract objects (oval nodes in points-to graph)

• Ensures that pointer analysis terminates

• Many sound schemes, varying in precision & efficiency
  – Too few abstract objects => efficient but imprecise
  – Too many abstract objects => expensive but precise
Scheme #1: Allocation-Site Based

One abstract object per allocation site

Allocation site identified by:
- **new** keyword in Java/C++
- **malloc()** call in C

Finitely many allocation sites in a program
=> finitely many abstract objects
Scheme #2: Type Based

- Allocation-site based scheme can be costly
  - Large programs
  - Clients needing quick turnaround time
  - Overly fine granularity of sites

- One abstract object per type

- Finitely many types in a program
  => finitely many abstract objects
Scheme #3: Heap-Insensitive

**Single** abstract object representing entire heap

Popular for languages with primarily stack-directed pointers (e.g. C)

Unsuitable for languages with only heap-directed pointers (e.g. Java)
Tradeoffs in Heap Abstraction Schemes

More Precise

Allocation-site based

Type based

Heap-insensitive

More Efficient
## QUIZ: May-Alias Analysis

Do the expression pairs may-alias under these two pointer analyses?

<table>
<thead>
<tr>
<th>May-Alias?</th>
<th>Allocation-Site Based</th>
<th>Type Based</th>
</tr>
</thead>
<tbody>
<tr>
<td>e, f</td>
<td>No</td>
<td></td>
</tr>
<tr>
<td>v.floors, v.events</td>
<td></td>
<td></td>
</tr>
<tr>
<td>v.floors[0], v.events[0]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>v.events[0], v.events[2]</td>
<td>Yes</td>
<td></td>
</tr>
</tbody>
</table>

![Diagram](https://via.placeholder.com/150)
**QUIZ: May-Alias Analysis**

Do the expression pairs may-alias under these two pointer analyses?

<table>
<thead>
<tr>
<th>May-Alias?</th>
<th>Allocation-Site Based</th>
<th>Type Based</th>
</tr>
</thead>
<tbody>
<tr>
<td>e, f</td>
<td>No</td>
<td></td>
</tr>
<tr>
<td>v.floors, v.events</td>
<td>No</td>
<td></td>
</tr>
<tr>
<td>v.floors[0], v.events[0]</td>
<td>No</td>
<td></td>
</tr>
<tr>
<td>v.events[0], v.events[2]</td>
<td>Yes</td>
<td></td>
</tr>
</tbody>
</table>

![Diagram showing elevator, object, floor, event, and v, f, e relationships](image)
QUIZ: May-Alias Analysis

Do the expression pairs may-alias under these two pointer analyses?

<table>
<thead>
<tr>
<th>May-Alias?</th>
<th>Allocation-Site Based</th>
<th>Type Based</th>
</tr>
</thead>
<tbody>
<tr>
<td>e, f</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>v.floors, v.events</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>v.floors[0], v.events[0]</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>v.events[0], v.events[2]</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>

Diagram:

- v
  - floors
  - events
- Object[]
  - [f]
  - [e]
- Elevator
- Floor
- Event
Modeling Aggregate Data Types: Arrays

• Common choice: single field [*] to represent all array elements
  – Cannot distinguish different elements of same array

• More sophisticated representations that make such distinctions are employed by array dependence analyses
  – Used to parallelize sequential loops by parallelizing compilers
Modeling Aggregate Data Types: Records

Three choices:

1. **Field-insensitive**: merge all fields of each record object

2. **Field-based**: merge each field of all record objects

3. **Field-sensitive**: keep each field of each (abstract) record object separate
### QUIZ: Pointer Analysis Classification

Classify the pointer analysis algorithm we learned in this lesson.

<table>
<thead>
<tr>
<th></th>
<th>A. Yes</th>
<th>B. No</th>
</tr>
</thead>
<tbody>
<tr>
<td>Flow-sensitive?</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Context-sensitive?</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Distinguishes fields of object?</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Distinguishes elements of array?</td>
<td></td>
<td></td>
</tr>
<tr>
<td>What kind of heap abstraction?</td>
<td>A. Allocation-site based</td>
<td>B. Type based</td>
</tr>
</tbody>
</table>
Classify the pointer analysis algorithm we learned in this lesson.

<table>
<thead>
<tr>
<th>Question</th>
<th>Choice 1</th>
<th>Choice 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Flow-sensitive?</td>
<td>B</td>
<td>A. Yes</td>
</tr>
<tr>
<td>Context-sensitive?</td>
<td>B</td>
<td>B. No</td>
</tr>
<tr>
<td>Distinguishes fields of object?</td>
<td>A</td>
<td>A. Yes</td>
</tr>
<tr>
<td>Distinguishes elements of array?</td>
<td>B</td>
<td>B. No</td>
</tr>
<tr>
<td>What kind of heap abstraction?</td>
<td>A</td>
<td>A. Alloc. based</td>
</tr>
<tr>
<td></td>
<td></td>
<td>B. Type based</td>
</tr>
</tbody>
</table>
What Have We Learned?

• What is pointer analysis?

• May-alias analysis vs. must-alias analysis

• Points-to graphs

• Working of a pointer analysis algorithm

• Classifying pointer analyses: flow sensitivity, context sensitivity, heap abstraction, aggregate modeling