A Tour of Pointer Analysis

Ondřej Lhoták
Pointer Analysis: Haven't We Solved This Problem Yet?

Michael Haya
Silicon Valley Research Center
2480 Moffett Blvd
Moffett Field, CA 94035
mhaya@siliconvalley.com

ABSTRACT

In the last years, new, very accurate pointer analyzers have come on the market. These analyzers are much more accurate than the ones we had in the past; they can even analyze some problems that we couldn’t analyze before. However, they are still very slow. We need to improve the efficiency of our pointer analyzers in order to make them usable in practice.

1. INTRODUCTION

A pointer analyzer is a program that decides whether two variables in a program can refer to the same object. It does this by analyzing the program and figuring out which variables are the same. The most common way to do this is to use a data flow analysis algorithm, which can be very slow. To make the analysis faster, we need to improve the efficiency of our pointer analyzers.

7. BACKGROUND

As a matter of fact, there is a more accurate way to do pointer analysis. It is called pointer analysis with constraints. It is more accurate because it can handle more constraints. However, it is also much slower.

3. GENERAL ISSUES

There are several open problems in pointer analysis. One of them is how to make the analysis faster. Another is how to make the analysis more accurate. We also need to improve the efficiency of our pointer analyzers.

54
What does pointer analysis do?

For each pointer (reference) in the program, what memory locations (objects) does it point to?
Why pointer analysis?

\[
\begin{align*}
    a &= 1 \\
    b &= 2 \\
    c &= a \times b \quad 3
\end{align*}
\]
Why pointer analysis?

\begin{align*}
a &= 1 \\
b &= 2 \\
\ast x &= 4 \\
c &= a + b \\?
\end{align*}
Why pointer analysis?

\begin{align*}
a &= 1 \\
b &= 2 \\
\ast x &= 4 \\
c &= a + b ? \\
\end{align*}

\begin{align*}
\text{If } x &= \& a, \quad \text{then } c = 6. \\
\text{If } x &= \& b, \quad \text{then } c = 5. \\
\text{If } x \neq \& a \& \& x \neq \& b, \quad \text{then } c = 3. \\
\end{align*}
Why pointer analysis?

\[
\begin{align*}
a &= 1 \\
b &= 2 \\
\text{foo}() \\
c &= a + b
\end{align*}
\]

\[
\text{void foo() \{ ... \}}
\]
Why pointer analysis?

a = 1
b = 2
\texttt{x.foo()}
c = a + b

\begin{align*}
\text{class } X & \{ \\
& \quad \text{void } \texttt{foo()} \{ \ldots \} \\
& \}
\end{align*}

\begin{align*}
\text{class } Y \text{ extends } X & \{ \\
& \quad \text{void } \texttt{foo()} \{ \ldots \} \\
& \}
\end{align*}

\begin{align*}
\text{class } Z \text{ extends } X & \{ \\
& \quad \text{void } \texttt{foo()} \{ \ldots \} \\
& \}
\end{align*}
Applications of pointer analysis

- Call graph construction
- Dependence analysis and optimization
- Cast check elimination
- Side effect analysis
- Escape analysis
- Slicing
- Parallelization
- ...

A Tour of Pointer Analysis | Ondrej Lhotak | University of Waterloo
Pointer analysis as an abstraction

For each pointer (reference) in the program, what memory locations (objects) does it point to?
For each pointer (reference) in the program, what memory locations (objects) does it point to?

\[ \alpha(p) = \alpha(q) = \begin{bmatrix} \text{Object} \end{bmatrix} \]
Pointer analysis as an abstraction

For each pointer (reference) in the program, what memory locations (objects) does it point to?

\[ \alpha(p) = p \]

\[ \alpha(q) = \]
Pointer analysis as an abstraction

Concrete program execution

Abstract analysis

\[ \alpha(p) = p \]

\[ \alpha(q) = \text{complex object} \]
Precision of points-to sets

\[
\{L_1\} \subset \{L_1, L_2\} \subset \{L_1, L_2, L_3\} \subset \{L_1, L_2, L_3, L_4\}
\]

← more precise

less precise →

unsound   uncomputable   conservative

actual behaviour on some executions

actual behaviour on all executions

possible analysis results
Precision vs. efficiency

A Tour of Pointer Analysis | Ondrej Lhotak | University of Waterloo
Precision vs. efficiency

Implementation tuning

← Efficient

Precise →
Precision for a specific application

\{L_1, L_2\} \subset \{L_1, L_2, L_3, L_4\}

This points-to set is more precise because it is smaller.

But suppose a particular application only cares whether \(L_1\) is in the set. Then for that application, both sets are equally precise.

Thus, precision/efficiency tradeoff must consider the application.
Design decisions for precision/efficiency

• The abstraction (affects precision and efficiency):
  – Type filtering
  – Field sensitivity
  – Directionality
  – Call graph construction
  – Context sensitivity
  – Flow sensitivity

• Algorithm and implementation (affects efficiency)
  – Propagation algorithm
  – Set implementation
An example abstraction and analysis

• The abstraction:
  – Type filtering
  – Field sensitivity
  – Directionality
  – Call graph construction
  – Context sensitivity
  – Flow sensitivity

First example:
  – without type filtering
  – field-sensitive
  – subset-based
  – ahead-of-time call graph
  – context-insensitive
  – flow-insensitive
Abstract object node:

Java:
L1: x = new Object()

C:
L1: x = malloc(42)

Represents some set of run-time objects (targets of pointers).
e.g. all objects allocated at a given allocation site
e.g. all objects of a given dynamic type
Address-of abstract object node:

\[ C: \quad x = \&a \]

Represents some set of run-time objects (targets of pointers). e.g. the object whose address is \&a.
Pointer variable node:

Represents some pointer-typed variable(s).
e.g. all instances of the local variable p in method m.
Pointer dereference node:

Java: 
\[ y = p.f \]

C: 
\[ y = *p \]

Represents a dereference of some pointer (where the pointer is a pointer variable node).
Heap pointer node:

$$pt(L1.f) = \{L2, L3\}$$

Represents a pointer stored in some object in the heap.
State space (the analysis result)

\[ pt(\ p) = \{ L1, L2, &q \} \]

\[ pt(\ L1.f) = \{ L1, L2, &q \} \]

\[ pt : (Var \cup (Obj \times Field)) \rightarrow \wp(Obj) \]

\[ pt : (Var \times Obj) \cup (Obj \times Field \times Obj) \]

where \ Obj = Alloc \cup AddrOf
State space (the analysis result)

\[
\begin{align*}
pt(p) &= \{ L1, L2, &q \} \\
pt(L1.f) &= \{ L1, L2, &q \}
\end{align*}
\]
Pointer Assignment Graph edges

allocation  \( L1: x = \text{new Object}() \)

assignment  \( x = y \)

store  \( y.f = x \)

load  \( x = y.f \)

\[ L1 \rightarrow x \]
\[ \{L1\} \subseteq pt(x) \]

\[ y \rightarrow x \]
\[ pt(y) \subseteq pt(x) \]

\[ x \rightarrow y.f \quad o \in pt(y) \]
\[ pt(x) \subseteq pt(o.f) \]

\[ y.f \rightarrow x \quad o \in pt(y) \]
\[ pt(o.f) \subseteq pt(x) \]

A Tour of Pointer Analysis | Ondrej Lhotak | University of Waterloo
Example

```java
static void foo() {
    L1: p = new O();
        q = p;
    L2: r = new O();
        p.f = r;
        t = bar( q );
}

static O bar( O s ) {
    return s.f;
}
```
static void foo() {
    L1: p = new O();
        q = p;
    L2: r = new O();
        p.f = r;
        t = bar( q );
}

static O bar( O s ) {
    return s.f;
}

Generate points-to assignment graph.
Example

static void foo() {
    L1: p = new O();
        q = p;
    L2: r = new O();
        p.f = r;
        t = bar( q );
}

static O bar( O s ) {
    return s.f;
}

Propagate points-to sets.
static void foo() {
    L1: p = new O();
        q = p;
    L2: r = new O();
        p.f = r;
        t = bar( q );
}

static O bar( O s ) {
    return s.f;
}
Example

static void foo() {
    L1: p = new O();
        q = p;
    L2: r = new O();
        p.f = r;
        t = bar( q );
}

static O bar( O s ) {
    return s.f;
}

Propagate points-to sets.
static void foo() {
    L1: p = new O();
        q = p;
    L2: r = new O();
        p.f = r;
        t = bar(q);
}
static O bar(O s) {
    return s.f;
}
Example

```java
static void foo() {
    L1: p = new O();
    q = p;
    L2: r = new O();
    p.f = r;
    t = bar( q );
}

static O bar( O s ) {
    return s.f;
}
```

Add load/store edges.
static void foo() {
    L1: p = new O();
        q = p;
    L2: r = new O();
        p.f = r;
        t = bar(q);
}

static O bar(O s) {
    return s.f;
}

Re-propagate points-to sets.
Overall algorithm

Simple:

```plaintext
repeat until no change {
    propagate abstract objects along edges
    for each load/store, add indirect edges to heap ptr nodes
}
```

Detailed:

```plaintext
add all allocation nodes to worklist
while worklist not empty {
    remove node v1 from worklist
    for each edge v1 -> v2, propagate pt(v1) into pt(v2)
        if v2 changed, add v2 to worklist
    for each load v1.f -> v3 {
        for each a in pt(v1) {
            add edge a.f -> v3 to assignment graph
            add node a.f to worklist
        }
    }
    for each store v3 -> v1.f {
        ... (as above)
    }
}
```
Comparison with 0CFA

Field-sensitive subset-based points-to analysis:
\[ \text{pt} : (\text{Var} \cup (\text{Obj} \times \text{Field})) \rightarrow \wp(\text{Obj}) \]

0CFA:

\[ \hat{\xi} \in \hat{\Sigma} = \text{Call} \times \hat{\text{Env}} \]
\[ \hat{\rho} \in \hat{\text{Env}} = \text{Var} \rightarrow \mathcal{P}\left(\hat{\text{Clo}}\right) \]
\[ \hat{\text{clo}} \in \hat{\text{Clo}} = \text{Lam} \]

\[ \Rightarrow \quad \Sigma = \text{Call} \times (\text{Var} \rightarrow \wp(\text{Lam})) \]
Comparison with OCF

L1: new

\[ p \]

L1

L1: \( \lambda \)

\[ p \]

L1
Set implementation

- **hash**: Using `java.util.HashSet`
- **array**: Sorted array, binary search
  
<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>d</th>
<th>g</th>
</tr>
</thead>
</table>

- **bit vector**:
  
<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>e</th>
<th>f</th>
<th>g</th>
<th>h</th>
<th>i</th>
<th>j</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

- **hybrid**:
  - array for small sets
  - bit vector for large sets

- **sparse bit vector**:

<table>
<thead>
<tr>
<th>0 (ab)</th>
<th>1</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 (cd)</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>3 (gh)</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

- **binary decision diagram**:
### Set implementation

<table>
<thead>
<tr>
<th>Method</th>
<th>Speed</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>hash</td>
<td>slow</td>
<td>large</td>
</tr>
<tr>
<td>array</td>
<td>slow</td>
<td>small</td>
</tr>
<tr>
<td>bit vector</td>
<td>fast</td>
<td>large</td>
</tr>
<tr>
<td>hybrid</td>
<td>fast</td>
<td>small</td>
</tr>
<tr>
<td>sparse bit vector</td>
<td>fast</td>
<td>small</td>
</tr>
<tr>
<td>binary decision diagram</td>
<td>depends</td>
<td>depends</td>
</tr>
</tbody>
</table>

**Slow vs. fast**: up to 100x difference  
**Large vs. small**: up to 3x difference

Set implementation is very important.
Incremental propagation

A Tour of Pointer Analysis | Ondrej Lhotak | University of Waterloo
Incremental propagation

- 1st iteration: propagate \{L1, L2, L3, L4\}
Incremental propagation

- 1st iteration: propagate \{L1, L2, L3, L4\}
- add L5 to pt(p)
Incremental propagation

1st iteration: propagate \{L1, L2, L3, L4\}
add L5 to pt(p)
2nd iteration: propagate \{L1, L2, L3, L4, L5\}
Idea: Split sets into old part and new part.
Incremental propagation

Idea: Split sets into old part and new part.

- 1st iteration: propagate \{L1, L2, L3, L4\]
Incremental propagation

Idea: Split sets into old part and new part.

- 1st iteration: propagate \{L1, L2, L3, L4\}
- flush new to old
Incremental propagation

Idea: Split sets into old part and new part.

- 1st iteration: propagate \{L1, L2, L3, L4\]
- flush new to old
- add L5 to new part of pt(p)
Incremental propagation

Idea: Split sets into old part and new part.

• 1st iteration: propagate \{L1, L2, L3, L4\}
• flush new to old
• add L5 to new part of pt(p)
• 2nd iteration: propagate \{L5\}
Incremental propagation

Idea: Split sets into old part and new part.

- 1st iteration: propagate \{L1, L2, L3, L4\}
- flush new to old
- add L5 to new part of \text{pt}(p)
- 2nd iteration: propagate \{L5\}
- flush new to old
Design decisions for precision/efficiency

• The abstraction (affects precision and efficiency):
  – Type filtering
  – Field sensitivity
  – Directionality
  – Call graph construction
  – Context sensitivity
  – Flow sensitivity

• Algorithm and implementation (affects efficiency)
  – Propagation algorithm
  – Set implementation
Type filtering

A x, z;
B y;
A: x = new A();
B: y = new B();
y = (B) x;
z = y;

Inheritance hierarchy:

A

B C
A x, z;
B y;
A: x = new A();
B: y = new B();
y = (B) x;
z = y;

Inheritance hierarchy:

A

B  C
Type filtering: after analysis

A x, z;
B y;
A: x = new A();
B: y = new B();
y = (B) x;
z = y;

Inheritance hierarchy:

A

B C
Type filtering: during analysis

```
A x, z;
B y;
A: x = new A();
B: y = new B();
y = (B) x;
z = y;
```

Inheritance hierarchy:

```
A
  B
  C
```
class A {
    Object fA;
}
class B {
    Object fB;
}
class C {
    Object fC;
}
A: a = new A();
B: b = new B();
C: c = new C();
Type filtering

- Ignoring types yields many large points-to sets.
- Filtering after propagation is almost as precise as during propagation.
- Filtering during propagation is both most precise and most efficient.

<table>
<thead>
<tr>
<th></th>
<th>Slow</th>
<th>Imprecise</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ignore</td>
<td></td>
<td></td>
</tr>
<tr>
<td>After propagation</td>
<td></td>
<td>Precise</td>
</tr>
<tr>
<td>During propagation</td>
<td>Fast</td>
<td>Precise</td>
</tr>
</tbody>
</table>
Design decisions for precision/efficiency

• The abstraction (affects precision and efficiency):
  – Type filtering
  – Field sensitivity
  – Directionality
  – Call graph construction
  – Context sensitivity
  – Flow sensitivity

• Algorithm and implementation (affects efficiency)
  – Propagation algorithm
  – Set implementation
Field reference representation

Idea: merge yellow nodes with same abstract object (resp. same field).
static void foo() {
  L1: p = new O();
      q = p;
  L2: r = new O();
      p.f = r;
      t = bar( q );
}

static O bar( O s ) { return s.f; }
Overall (field-based) algorithm

merge each SCC in assignment graph into a single node
topologically sort resulting DAG
for each node v1 in topological order {
  for each edge v1 -> v2 {
    propagate pt(v1) into pt(v2)
  }
}

Each edge is processed only once.
Worst-case $O(n^2)$.
Also, worst-case is linear in size of output.

In contrast, field-sensitive algorithm is $O(n^3)$. 
Example of precision loss

```
A x, y, z;
B u, v, w;
A1: x = new A();
y = x;
A2: z = new A();
B: u = new B();
x.f = u;
v = y.f;
w = z.f;
```

Field-sensitive

Field-based
### Field sensitivity summary

<table>
<thead>
<tr>
<th></th>
<th>Java</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>field-insensitive</td>
<td>sound</td>
<td>sound</td>
</tr>
<tr>
<td></td>
<td>slow</td>
<td>slow</td>
</tr>
<tr>
<td></td>
<td>imprecise</td>
<td>imprecise</td>
</tr>
<tr>
<td>field-based</td>
<td>sound</td>
<td>unsound</td>
</tr>
<tr>
<td></td>
<td>fast</td>
<td></td>
</tr>
<tr>
<td></td>
<td>imprecise</td>
<td></td>
</tr>
<tr>
<td>field-sensitive</td>
<td>sound</td>
<td>sound</td>
</tr>
<tr>
<td></td>
<td>slowest</td>
<td>slowest</td>
</tr>
<tr>
<td></td>
<td>precise</td>
<td>precise</td>
</tr>
</tbody>
</table>
Comparison: field-based PTA vs. 0CFA

Field-based subset-based points-to analysis:
\[ \text{pt} : \text{Var} \rightarrow \wp(\text{Obj}) \]

0CFA:
\[ \hat{\xi} \in \hat{\Sigma} = \text{Call} \times \hat{\text{Env}} \]
\[ \hat{\rho} \in \hat{\text{Env}} = \text{Var} \rightarrow \mathcal{P} \left( \hat{\text{Clo}} \right) \]
\[ \hat{\text{clo}} \in \hat{\text{Clo}} = \text{Lam} \]
\[ \Rightarrow \Sigma = \text{Call} \times (\text{Var} \rightarrow \wp(\text{Lam})) \]
Design decisions for precision/efficiency

• The abstraction (affects precision and efficiency):
  – Type filtering
  – Field sensitivity
  – Directionality
  – Call graph construction
  – Context sensitivity
  – Flow sensitivity

• Algorithm and implementation (affects efficiency)
  – Propagation algorithm
  – Set implementation
Object x, y, z;
L1: x = new Object();
L2: y = new Object();
    if(*) {
        z = x;
    } else {
        z = y;
    }
Object x, y, z;
L1: x = new Object();
L2: y = new Object();
if(*) {
    z = x;
} else {
    z = y;
}

Equality-based analysis
aka Unification-based analysis
aka Steensgaard's analysis
Implementation of unification

Object x, y, z;
L1: x = new Object();
L2: y = new Object();
if(*) {
    z = x;
} else {
    z = y;
}

Step 1: Process allocation edges

Equality-based analysis
aka Unification-based analysis
aka Steensgaard's analysis
Implementation of unification

Object x, y, z;
L1: x = new Object();
L2: y = new Object();
if(*) {
    z = x;
} else {
    z = y;
}

Equality-based analysis
aka Unification-based analysis
aka Steensgaard's analysis

Step 1: Process allocation edges
Step 2: Repeatedly unify nodes connected by assignments

Running time: almost linear
Implementation of unification

Object x, y, z;
L1: x = new Object();
L2: y = new Object();
if(*) {
    z = x;
} else {
    z = y;
}

Equality-based analysis
aka Unification-based analysis
aka Steensgaard's analysis

Step 1: Process allocation edges
Step 2: Repeatedly unify nodes connected by assignments.
Implementation of unification

```java
Object x, y, z;
L1: x = new Object();
L2: y = new Object();
if(*) {
    z = x;
} else {
    z = y;
}
```

Equality-based analysis
aka Unification-based analysis
aka Steensgaard's analysis

Step 1: Process allocation edges
Step 2: Repeatedly unify nodes connected by assignments.
Also unify nodes pointed-to by same node.

Running time: almost linear
A C example

```c
a = &b;
b = &c;
a = &d;
d = &e;
```

Subset-based analysis
A C example

a = &b;
b = &c;
a = &d;
d = &e;

Subset-based analysis
A C example

```
a = &b;
b = &c;
a = &d;
d = &e;
```

Equality-based analysis
A C example

```c
a = &b;
b = &c;
a = &d;
d = &e;
```

Equality-based analysis
A C example

a = &b;
b = &c;
a = &d;
d = &e;

Equality-based analysis
A C example

Equality-based analysis

Invariant: each node points to at most one other node.
A C example

a = &b;
b = &c;
a = &d;
d = &e;

Equality-based analysis
A problem with unification and OOP

class A extends Object {
    public A() {
        super();
    }
}
class B extends Object {
    ...
}
L1: x = new A();
L2: y = new B();

Subset-based analysis
A problem with unification and OOP

class A extends Object {
    public A() {
        super();
    }
}
class B extends Object {
    ...
}
L1: x = new A();
L2: y = new B();

Equality-based analysis
A problem with unification and OOP

class A extends Object {
    public A() {
        super();
    }
}
class B extends Object {
    ...
}
L1: x = new A();
L2: y = new B();

Equality-based analysis
A problem with unification and OOP

class A extends Object {
    public A() {
        super();
    }
}
class B extends Object {
    ...
}
L1: x = new A();
L2: y = new B();

Equality-based analysis
A problem with unification and OOP

class A extends Object {
    public A() {
        super();
    }
}
class B extends Object {
    ...
}
L1: x = new A();
L2: y = new B();

Equality-based analysis
A problem with unification and OOP

class A extends Object {
    public A() {
        super();
    }
}
class B extends Object {
    ...
}
L1: x = new A();
L2: y = new B();

Equality-based analysis
A problem with unification and OOP

class A extends Object {
    public A() {
        super();
    }
}
class B extends Object {
    ...
}
L1: x = new A();
L2: y = new B();

Equality-based analysis

Every pointer points to every object!
... but context sensitivity will fix this.
Design decisions for precision/efficiency

• The abstraction (affects precision and efficiency):
  – Type filtering
  – Field sensitivity
  – Directionality
  – Call graph construction
  – Context sensitivity
  – Flow sensitivity

• Algorithm and implementation (affects efficiency)
  – Propagation algorithm
  – Set implementation
How big is the call graph?

```java
public class Hello {
    public static final void main(String[] args) {
        System.out.println("Hello");
    }
}
```

• Number of methods actually executed: ??
• Number of methods in static call graph: ??
How big is the call graph?

```
public class Hello {
    public static final void main(String[] args) {
        System.out.println("Hello");
    }
}
```

- Number of methods actually executed: 498
- Number of methods in static call graph: ??
How big is the call graph?

```java
public class Hello {
    public static final void main(String[] args) {
        System.out.println("Hello");
    }
}
```

- Number of methods actually executed: 498
- Number of methods in static call graph: 3204
## Call graph construction

<table>
<thead>
<tr>
<th>To determine ...</th>
<th>... we need to know</th>
</tr>
</thead>
<tbody>
<tr>
<td>points-to sets</td>
<td></td>
</tr>
<tr>
<td>pointer assignment edges</td>
<td></td>
</tr>
<tr>
<td>reachable methods</td>
<td></td>
</tr>
<tr>
<td>call graph edges</td>
<td></td>
</tr>
</tbody>
</table>
### Call graph construction

<table>
<thead>
<tr>
<th>To determine ...</th>
<th>... we need to know</th>
</tr>
</thead>
<tbody>
<tr>
<td>points-to sets</td>
<td>pointer assignment edges</td>
</tr>
<tr>
<td>pointer assignment edges</td>
<td></td>
</tr>
<tr>
<td>reachable methods</td>
<td></td>
</tr>
<tr>
<td>call graph edges</td>
<td></td>
</tr>
</tbody>
</table>
## Call graph construction

<table>
<thead>
<tr>
<th>To determine ...</th>
<th>... we need to know</th>
</tr>
</thead>
<tbody>
<tr>
<td>points-to sets</td>
<td>pointer assignment edges</td>
</tr>
<tr>
<td>pointer assignment edges</td>
<td>reachable methods</td>
</tr>
<tr>
<td>reachable methods</td>
<td>call graph edges</td>
</tr>
<tr>
<td>call graph edges</td>
<td></td>
</tr>
</tbody>
</table>
## Call graph construction

<table>
<thead>
<tr>
<th>To determine ...</th>
<th>... we need to know</th>
</tr>
</thead>
<tbody>
<tr>
<td>points-to sets</td>
<td>pointer assignment edges</td>
</tr>
<tr>
<td>pointer assignment edges</td>
<td>reachable methods call graph edges</td>
</tr>
<tr>
<td>reachable methods</td>
<td>call graph edges</td>
</tr>
<tr>
<td>call graph edges</td>
<td></td>
</tr>
</tbody>
</table>
## Call graph construction

<table>
<thead>
<tr>
<th>To determine ...</th>
<th>... we need to know</th>
</tr>
</thead>
<tbody>
<tr>
<td>points-to sets</td>
<td>pointer assignment edges</td>
</tr>
<tr>
<td>pointer assignment edges</td>
<td>reachable methods call graph edges</td>
</tr>
<tr>
<td>reachable methods</td>
<td>call graph edges</td>
</tr>
<tr>
<td>call graph edges</td>
<td>points-to sets</td>
</tr>
</tbody>
</table>
## Call graph construction

<table>
<thead>
<tr>
<th>To determine ...</th>
<th>... we need to know</th>
</tr>
</thead>
<tbody>
<tr>
<td>points-to sets</td>
<td>pointer assignment edges</td>
</tr>
<tr>
<td>pointer assignment edges</td>
<td>reachable methods call graph edges</td>
</tr>
<tr>
<td>reachable methods</td>
<td>call graph edges</td>
</tr>
<tr>
<td>call graph edges</td>
<td>points-to sets</td>
</tr>
</tbody>
</table>

![Diagram showing relationships between reachability, call graph edges, pointer assignment edges, and points-to sets](diagram.png)
Ahead of time call graph construction

1. Assume every pointer can point to any object compatible with its declared type.

2. Explore call graph using this assumption, listing reachable methods (Class Hierarchy Analysis).

3. Generate pointer assignment graph using resulting call edges and reachable methods.

4. Propagate points-to sets along pointer assignment graph.
Ahead of time call graph construction

1. Assume every pointer can point to any object compatible with its declared type.

2. Explore call graph using this assumption, listing reachable methods (Class Hierarchy Analysis).

3. Generate pointer assignment graph using resulting call edges and reachable methods.

4. Propagate points-to sets along pointer assignment graph.
   - no iteration
   - very imprecise due to many reachable methods
On-the-fly call graph construction

1. Start with only initial reachable methods, no call edges, no pointer assignment edges, and no points-to sets.
2. Iteratively generate pointer assignment edges, points-to sets, call edges, and reachable methods implied by current information.
3. Stop when overall fixed point is reached.
On-the-fly call graph construction

1. Start with only initial reachable methods, no call edges, no pointer assignment edges, and no points-to sets.

2. Iteratively generate pointer assignment edges, points-to sets, call edges, and reachable methods implied by current information.

3. Stop when overall fixed point is reached.

- requires iteration
- slower
- more complicated
- much more precise due to fewer reachable methods
Partly on-the-fly call graph construction

1. Assume all methods are reachable.
2. Generate pointer assignment edges for all methods.
3. Iteratively propagate points-to sets, add call edges, and generate pointer assignment edges for new call edges.
Partly on-the-fly call graph construction

1. Assume all methods are reachable.
2. Generate pointer assignment edges for all methods.
3. Iteratively propagate points-to sets, add call edges, and generate pointer assignment edges for new call edges.

- requires iteration
- speed is in between ahead-of-time and on-the-fly
- complexity is in between...
- still imprecise due to many reachable methods
Design decisions for precision/efficiency

• The abstraction (affects precision and efficiency):
  – Type filtering
  – Field sensitivity
  – Directionality
  – Call graph construction
  – Context sensitivity
  – Flow sensitivity

• Algorithm and implementation (affects efficiency)
  – Propagation algorithm
  – Set implementation
Object id(Object o) {
  return o;
}

void f() {
  L1: Object a = new Object();
  L2: Object b = new Object();
  Object c = id(a);
  Object d = id(b);
}
Call strings approach (aka cloning)

Object id(Object o) {
    return o;
}

void f() {
L1: Object a = new Object();
L2: Object b = new Object();
L3: Object c = id(a);
L4: Object d = id(b);
}
Object id(Object o) {
  return o;
}
void f() {
  L1: Object a = new Object();
  L2: Object b = new Object();
  Object c = id(a);
  Object d = id(b);
}

Challenge: how to design a summary that
• precisely models all effects of id() (and its transitive callees)
• is cheap to compute and represent
• is cheap to instantiate
Comparison with 1CFA

Field-sensitive subset-based 1-call-site-sensitive points-to analysis:

$$\text{pt} : (\text{Call} \times \text{Var} \cup (\text{Obj} \times \text{Field})) \rightarrow \wp(\text{Obj})$$

1CFA:

$$\Sigma = \text{Call} \times (\text{Var} \rightarrow \text{Addr}) \times (\text{Addr} \rightarrow \wp(\text{Lam} \times \text{Var} \rightarrow \text{Addr})) \times \text{Call}$$
Limitation of single call site context

Object id(Object o) {
    L5: return id2(o);
}

Object id2(Object o) {
    return o;
}

void f() {
    L1: Object a = new Object();
    L2: Object b = new Object();
    L3: Object c = id(a);
    L4: Object d = id(b);
}
Object id(Object o) {
    L5: return id2(o);
}

Object id2(Object o) {
    return o;
}

void f() {
    L1: Object a = new Object();
    L2: Object b = new Object();
    L3: Object c = id(a);
    L4: Object d = id(b);
}
**k-call-site context sensitivity**

- **k-Call-Site**: keep last $k$ call sites
  - space/time complexity exponential in $k$
- **Full Call String**: keep full string of call sites
  - exponential number of call strings
  - recursion: infinite number of call strings
    - exclude any call site that is in a recursive cycle from string
    - still exponential
    - what if many call sites are in recursive cycle?
  - C: call graph is almost DAG $\Rightarrow$ works well
  - Java: half of call graph is one big SCC $\Rightarrow$ no precision
Object alloc() {
    L1: return new Object();
}

void f() {
    Object a = alloc();
    Object b = alloc();
}
Object alloc() {
L1: return new Object();
}

void f() {
L2: Object a = alloc();
L3: Object b = alloc();
}
Object alloc() {
    L1: return new Object();
}

void f() {
    L2: Object a = alloc();
    L3: Object b = alloc();
}
Comparison with 1CFA

Field-sensitive subset-based 1-call-site-sensitive points-to analysis with context-sensitive heap abstraction:

$$pt : (\text{Call} \times \text{Var} \cup (\text{Call} \times \text{Obj} \times \text{Field})) \rightarrow \wp(\text{Call} \times \text{Obj})$$

1CFA:

$$\Sigma = \text{Call} \times (\text{Var} \rightarrow \text{Addr}) \times (\text{Addr} \rightarrow \wp(\text{Lam} \times \text{Var} \rightarrow \text{Addr})) \times \text{Call}$$
class Container {
    private Item item;
    public void set(Item i) {
        this.item = i;
    }
}

L1: Container c1 = new Container();
L2: Item i1 = new Item();
L3: c1.set(i1);

L4: Container c2 = new Container();
L5: Item i2 = new Item();
L6: c2.set(i2);

[Milanova&Ryder]
class Container {
    private Item item;
    public void set(Item i) {
        this.item = i;
    }
}

L1: Container c1 = new Container();
L2: Item i1 = new Item();
L3: c1.set(i1);

L4: Container c2 = new Container();
L5: Item i2 = new Item();
L6: c2.set(i2);

[Milanova&Ryder]
Object-sensitive analysis

- Like call-site context-sensitive analysis:
  - can use strings of abstract objects
  - can make heap abstraction (object-)context-sensitive

- Call-site CS and object-sensitive CS have incomparable precision (neither is theoretically more precise)

- In practice, for OO programs, object sensitivity more precise than call-site sensitivity for the same context string length

[Milanova&Ryder]

A Tour of Pointer Analysis | Ondrej Lhotak | University of Waterloo
Effect of context-sensitivity in Java

• For call graph construction:
  – context sensitivity has some effect

• For cast safety analysis:
  – context sensitivity substantially improves precision
  – object sensitivity more precise than call sites
  – context sensitive heap abstraction further improves precision
  – context strings longer than 1 add little precision
  – $\infty$-call-site ignoring cycles less precise than 1-call-site
Context-sensitive equality-based analysis

```c
f(a, b, c) {
    d = a;
    d = b;
    e = c;
    e = a.f;
    return e;
}
```

L1: x = new A();
L2: y = new A();
L3: z = new A();
L4: w = new A();
x.f = w;
v = f(x, y, z);

[Lattner&Adve]
Context-sensitive equality-based analysis

```java
f(a, b, c) {
    d = a;
    d = b;
    e = c;
    e = a.f;
    return e;
}

L1: x = new A();
L2: y = new A();
L3: z = new A();
L4: w = new A();
x.f = w;
v = f(x, y, z);
```

[Lattner&Adve]
Context-sensitive equality-based analysis

```c
f(a, b, c) {
    d = a;
    d = b;
    e = c;
    e = a.f;
    return e;
}
```

L1: `x = new A();`
L2: `y = new A();`
L3: `z = new A();`
L4: `w = new A();`
    `x.f = w;`
    `v = f(x, y, z);`

[Lattner&Adve]
Context-sensitive equality-based analysis

```java
f(a, b, c) {
    d = a;
    d = b;
    e = c;
    e = a.f;
    return e;
}

L1: x = new A();
L2: y = new A();
L3: z = new A();
L4: w = new A();
x.f = w;
v = f(x, y, z);
```

[Lattner&Adve]
Context-sensitive equality-based analysis

```java
f(a, b, c) {
    d = a;
    d = b;
    e = c;
    e = a.f;
    return e;
}
```

```
L1: x = new A();
L2: y = new A();
L3: z = new A();
L4: w = new A();
x.f = w;
v = f(x, y, z);
```

[Lattnner&Adve]
Context-sensitive equality-based analysis

```java
f(a, b, c) {
    d = a;
    d = b;
    e = c;
    e = a.f;
    return e;
}
```

L1: `x = new A();`
L2: `y = new A();`
L3: `z = new A();`
L4: `w = new A();`
    `x.f = w;`
    `v = f(x, y, z);`

[Lattner&Adve]
f(a, b, c) {
    d = a;
    d = b;
    e = c;
    e = a.f;
    return e;
}

L1: x = new A();
L2: y = new A();
L3: z = new A();
L4: w = new A();
    x.f = w;
    v = f(x, y, z);
Context-sensitive equality-based analysis

```
f(a, b, c) {
    d = a;
    d = b;
    e = c;
    e = a.f;
    return e;
}
```

L1: x = new A();
L2: y = new A();
L3: z = new A();
L4: w = new A();
   x.f = w;
   v = f(x, y, z);

[Lattner&Adve]
Context-sensitive equality-based analysis

f(a, b, c) {
    d = a;
    d = b;
    e = c;
    e = a.f;
    return e;
}

L1: x = new A();
L2: y = new A();
L3: z = new A();
L4: w = new A();
    x.f = w;
    v = f(x, y, z);

[Lattner&Adve]
f(a, b, c) {
    d = a;
    d = b;
    e = c;
    e = a.f;
    return e;
}

L1: x = new A();
L2: y = new A();
L3: z = new A();
L4: w = new A();
x.f = w;
v = f(x, y, z);
Context-sensitive equality-based analysis

```java
f(a, b, c) {
    d = a;
    d = b;
    e = c;
    e = a.f;
    return e;
}
```

```
L1: x = new A();
L2: y = new A();
L3: z = new A();
L4: w = new A();
x.f = w;
v = f(x, y, z);
```

[Lattner&Adve]
Refinement demand-driven analysis

Object id(Object o) {
    return o;
}

void f() {
    L1: Object a = new Object();
    L2: Object b = new Object();
    L3: Object c = id(a);
    L4: Object d = id(b);
}

[Sridharan&Bodík]
Refinement demand-driven analysis

Object id(Object o) {
  return o;
}

void f() {
  L1: Object a = new Object();
  L2: Object b = new Object();
  L3: Object c = id(a);
  L4: Object d = id(b);
}

L1 ∈ pt(d) ⇔ ∃ balanced-parens path from L1 to d

[Sridharan&Bodík]
Refinement demand-driven analysis

Object id(Object o) {
    return o;
}

void f() {
    L1: Object a = new Object();
    L2: Object b = new Object();
    L3: Object c = id(a);
    L4: Object d = id(b);
}

L1 ∈ pt(d) ⇔ ∃ balanced-parens path from L1 to d
But there are lots of paths to search.

[Sridharan&Bodík]
Refinement demand-driven analysis

Object id(Object o) {
    return o;
}

void f() {
    L1: Object a = new Object();
    L2: Object b = new Object();
    L3: Object c = id(a);
    L4: Object d = id(b);
}

\[ L1 \in pt(d) \iff \exists \text{ balanced-parens path from } L1 \text{ to } d \]

But there are lots of paths to search.
Add shortcut edges to graph.
No path even with shortcuts \implies no path at all.

[Sridharan&Bodík]
Object id(Object o) {
    return o;
}

void f() {
    Object a = new Object();
    Object b = new Object();
    Object c = id(a);
    Object d = id(b);
}

L1 \in \text{pt}(d) \iff \exists \text{balanced-parens path from L1 to d}

But there are lots of paths to search.
Add shortcut edges to graph.
No path even with shortcuts \Rightarrow no path at all.
On balanced paths found, gradually remove shortcuts.

[Sridharan\&Bodík]
Design decisions for precision/efficiency

• The abstraction (affects precision and efficiency):
  – Type filtering
  – Field sensitivity
  – Directionality
  – Call graph construction
  – Context sensitivity
  – Flow sensitivity

• Algorithm and implementation (affects efficiency)
  – Propagation algorithm
  – Set implementation
L1: \( a = \text{new \: Object}() \);
L2: \( b = \text{new \: Object}() \);
L3: \( c = \text{new \: Object}() \);
L4: \( a = b \);
L5: \( b = a \);
L6: \( c = b \);
Flow sensitivity

L1: `a = new Object(); a -> {L1}`
L2: `b = new Object(); a -> {L1}, b -> {L2}`
L3: `c = new Object(); a -> {L1}, b -> {L2}, c -> {L3}`

L4: `a = b; a -> {L2}, b -> {L2}, c -> {L3}`
L5: `b = a; a -> {L2}, b -> {L2}, c -> {L3}`
L6: `c = b; a -> {L2}, b -> {L2}, c -> {L2}`

Strong updates: overwrite existing pt-set contents
Flow sensitivity using SSA form

Currently live reaching definition of each variable.
Flow sensitivity using SSA form

L1: a1 = new Object();  a1-> {L1}                        1
L2: b2 = new Object();  a1-> {L1}, b2-> {L2}             1 2
L3: c3 = new Object();  a1-> {L1}, b2-> {L2}, c3-> {L3}  1 2 3
L4: a4 = b2;            a4-> {L2},  b2-> {L2},  c3-> {L3}  4 2 3
L5: b5 = a4;            a4-> {L2},  b5-> {L2},  c3-> {L3}  4 5 3
L6: c6 = b5;            a4-> {L2},  b5-> {L2},  c6-> {L2}  4 5 6

For local variables, FI analysis on SSA form gives same result as FS analysis on original program.
Flow sensitivity using SSA form

Which definition of a is current at L7?
Which definition of $a$ is current at L7?
Use $\phi$ to create new definition of $a$.
$\text{pt}(a_6) = \text{pt}(a_2) \cup \text{pt}(a_4)$
### When does flow sensitivity matter?

<table>
<thead>
<tr>
<th></th>
<th>Java</th>
<th>C/C++</th>
</tr>
</thead>
<tbody>
<tr>
<td>local variables</td>
<td>no, use SSA form</td>
<td></td>
</tr>
<tr>
<td>address-taken local variables</td>
<td>no, don't exist</td>
<td>possibly</td>
</tr>
<tr>
<td>global variables</td>
<td>unlikely, values usually long-lived</td>
<td></td>
</tr>
<tr>
<td>fields of heap objects</td>
<td>no strong updates on heap objects unless analysis extended with single-concrete-object abstraction</td>
<td></td>
</tr>
</tbody>
</table>
Design decisions for precision/efficiency

• The abstraction (affects precision and efficiency):
  – Type filtering
  – Field sensitivity
  – Directionality
  – Call graph construction
  – Context sensitivity
  – Flow sensitivity

• Algorithm and implementation (affects efficiency)
  – Propagation algorithm
  – Set implementation