Static Program Analysis

Interprocedural Analysis

Nanjing University

Tian Tan

2020
Contents

1. Motivation
2. Call Graph Construction (CHA)
3. Interprocedural Control-Flow Graph
4. Interprocedural Data-Flow Analysis
1. Motivation
2. Call Graph Construction (CHA)
3. Interprocedural Control-Flow Graph
4. Interprocedural Data-Flow Analysis
Motivation of Interprocedural Analysis

So far, all analyses we learnt are *intraprocedural*. How to deal with method calls?

```c
void foo() {
    int n = bar(42);
}

int bar(int x) {
    int y = x + 1;
    return 10;
}
```

Constant Propagation
Motivation of Interprocedural Analysis

So far, all analyses we learnt are *intraprocedural*. How to deal with method calls?

- Make the **most conservative assumption** for method calls, for safe-approximation

```c
void foo() {
    int n = bar(42);
}

int bar(int x) {
    int y = x + 1;
    return 10;
}
```
Motivation of Interprocedural Analysis

Constant Propagation

```c
void foo() {
    int n = bar(42);
}

int bar(int x) {
    int y = x + 1;
    return 10;
}
```

So far, all analyses we learnt are intraprocedural. How to deal with method calls?

- Make the most conservative assumption for method calls, for safe-approximation
- Source of imprecision
  - $x = NAC$, $y = NAC$
  - $n = NAC$
Motivation of Interprocedural Analysis

So far, all analyses we learnt are **intraprocedural**. How to deal with method calls?

- Make the **most conservative assumption** for method calls, for safe-approximation
- **Source of imprecision**
  - \( x = \text{NAC}, \ y = \text{NAC} \)
  - \( n = \text{NAC} \)

For better precision, we need **Interprocedural analysis**: propagate data-flow information along **interprocedural control-flow edges** (i.e., call and return edges)
Motivation of Interprocedural Analysis

So far, all analyses we learnt are intraprocedural. How to deal with method calls?
- Make the most conservative assumption for method calls, for safe-approximation
- Source of imprecision
  - x = NAC, y = NAC
  - n = NAC

For better precision, we need Interprocedural analysis: propagate data-flow information along interprocedural control-flow edges (i.e., call and return edges)
- x = 42, y = 43
- n = 10
To perform interprocedural analysis, we need call graph.

So far, all analyses we learnt are intraprocedural. How to deal with method calls?
• Make the most conservative assumption for method calls, for safe-approximation
• Source of imprecision
  🚫 x = NAC, y = NAC
  🚫 n = NAC

For better precision, we need interprocedural analysis: propagate data-flow information along interprocedural control-flow edges (i.e., call and return edges)
  ➢ x = 42, y = 43
  ➢ n = 10

Motivation of Interprocedural Analysis

Constant Propagation

```c
void foo() {
    int n = bar(42);
}

int bar(int x) {
    int y = x + 1;
    return 10;
}
```
Contents

1. Motivation
2. Call Graph Construction (CHA)
3. Interprocedural Control-Flow Graph
4. Interprocedural Data-Flow Analysis
Call Graph

A representation of calling relationships in the program

• Essentially, a call graph is a set of call edges from call-sites to their target methods (callees)
Call Graph

A representation of calling relationships in the program

Essentially, a call graph is a set of call edges from call-sites to their target methods (callees)

```c
void foo() {
    bar();
    baz(123);
}

void bar(int x) {
    baz(666);
}

void baz() {
}
```
Applications of Call Graph

• Foundation of all interprocedural analyses
• Program optimization
• Program understanding
• Program debugging
• Program testing
• And many more …

Call graph is VERY important program information
Call Graph Construction for OOPLs (focus on Java)

- Class hierarchy analysis (CHA)
- Rapid type analysis (RTA)
- Variable type analysis (VTA)
- Pointer analysis ($k$-CFA)
Call Graph Construction for OOPLs (focus on Java)

- Class hierarchy analysis (CHA)
- Rapid type analysis (RTA)
- Variable type analysis (VTA)
- Pointer analysis ($k$-CFA)

More precise

More efficient
Call Graph Construction for OOPLs (focus on Java)

- Class hierarchy analysis (CHA)
- Rapid type analysis (RTA)
- Variable type analysis (VTA)
- Pointer analysis (k-CFA)

More precise

More efficient

Tian Tan @ Nanjing University
# Method Calls (Invocations) in Java

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Static call</th>
<th>Special call</th>
<th>Virtual call</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>invokestatic</td>
<td>invokespecial</td>
<td>invokeinterface</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>invokevirtual</td>
</tr>
</tbody>
</table>
## Method Calls (Invocations) in Java

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Static call</th>
<th>Special call</th>
<th>Virtual call</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>invokestatic</td>
<td>invokespecial</td>
<td>invokeinterface invokevirtual</td>
</tr>
<tr>
<td>Receiver objects</td>
<td>✗</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Target methods</td>
<td>• Static methods</td>
<td>• Constructors • Private instance methods • Superclass instance methods</td>
<td>• Other instance methods</td>
</tr>
</tbody>
</table>
# Method Calls (Invocations) in Java

<table>
<thead>
<tr>
<th></th>
<th>Static call</th>
<th>Special call</th>
<th>Virtual call</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instruction</td>
<td><code>invokestatic</code></td>
<td><code>invokespecial</code></td>
<td><code>invokeinterface</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td><code>invokevirtual</code></td>
</tr>
<tr>
<td>Receiver objects</td>
<td>×</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Target methods</td>
<td>• Static methods</td>
<td>• Constructors</td>
<td>• Other instance methods</td>
</tr>
<tr>
<td></td>
<td>• Private instance methods</td>
<td>• Private instance methods</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Superclass instance methods</td>
<td></td>
<td></td>
</tr>
<tr>
<td>#Target methods</td>
<td>1</td>
<td>1</td>
<td>≥1 (polymorphism)</td>
</tr>
<tr>
<td>Determinancy</td>
<td>Compile-time</td>
<td>Compile-time</td>
<td>Run-time</td>
</tr>
</tbody>
</table>

Tian Tan @ Nanjing University
## Method Calls (Invocations) in Java

<table>
<thead>
<tr>
<th></th>
<th>Static call</th>
<th>Special call</th>
<th>Virtual call</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Instruction</strong></td>
<td>invokestatic</td>
<td>invokepecial</td>
<td>invokeinterface</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>invokevirtual</td>
</tr>
<tr>
<td><strong>Receiver objects</strong></td>
<td>×</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td><strong>Target methods</strong></td>
<td>• Static methods</td>
<td>• Constructors</td>
<td>• Other instance</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Private instance</td>
<td>methods</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Superclass instance</td>
<td>methods</td>
</tr>
<tr>
<td></td>
<td></td>
<td>methods</td>
<td></td>
</tr>
<tr>
<td><strong>#Target methods</strong></td>
<td>1</td>
<td>1</td>
<td>≥1 (polymorphism)</td>
</tr>
<tr>
<td><strong>Determinancy</strong></td>
<td>Compile-time</td>
<td>Compile-time</td>
<td>Run-time</td>
</tr>
</tbody>
</table>

**Key** to call graph construction for OOPLs

Tian Tan @ Nanjing University
Method Dispatch of Virtual Calls

During run-time, a virtual call is resolved based on
1. type of the receiver object (pointed by o)
2. method signature at the call site
Method Dispatch of Virtual Calls

During run-time, a virtual call is resolved based on
1. type of the receiver object (pointed by o)
2. method signature at the call site

In this lecture, a signature acts as an identifier of a method
• Signature = class type + method name + descriptor
• Descriptor = return type + parameter types

\[ o^1. \texttt{foo(...)}^2; \]
Method Dispatch of Virtual Calls

During run-time, a virtual call is resolved based on
1. type of the receiver object (pointed by o)
2. method signature at the call site

In this lecture, a signature acts as an identifier of a method

```java
class C {
    T foo(P p, Q q, R r) { ... }
}
```

- Signature = class type + method name + descriptor
- Descriptor = return type + parameter types
Method Dispatch of Virtual Calls

During run-time, a virtual call is resolved based on
1. type of the receiver object (pointed by o)
2. method signature at the call site

In this lecture, a signature acts as an identifier of a method

```java
class C {
    T foo(P p, Q q, R r) { ... }
}

<C: T foo(P, Q, R)>  
C.foo(P, Q, R) for short
```

- Signature = class type + method name + descriptor
- Descriptor = return type + parameter types
Method Dispatch of Virtual Calls

During run-time, a virtual call is resolved based on
1. type of the receiver object (pointed by o): \( c \)
2. method signature at the call site: \( m \)

We define function \( \text{Dispatch}(c, m) \) to simulate the procedure of run-time method dispatch

\[
\text{Dispatch}(c, m) = \begin{cases} 
    m', & \text{if } c \text{ contains non-abstract method } m' \text{ that has the same name and descriptor as } m \\
    \text{Dispatch}(c', m), & \text{otherwise}
\end{cases}
\]

where \( c' \) is superclass of \( c \)

\(<C: T \text{ foo}(P, Q, R)>\)
Dispatch: An Example

```java
class A {
    void foo() {...}
}
class B extends A {
}
class C extends B {
    void foo() {...}
}

void dispatch() {
    A x = new B();
    x.foo();
    A y = new C();
    y.foo();
}
```

**Dispatch**

\[
\text{Dispatch}(c, m) = \begin{cases} 
    m', & \text{if } c \text{ contains non-abstract method } m' \text{ that has the same name and descriptor as } m \\
    \text{Dispatch}(c', m), & \text{otherwise}
\end{cases} \\
\text{where } c' \text{ is superclass of } c
\]
Dispatch: An Example

\[
\text{class } A \{
\quad \text{void } \text{foo() } \{ \ldots \}
\}
\]

\[
\text{class } B \text{ extends } A \{
\}
\]

\[
\text{class } C \text{ extends } B \{
\quad \text{void } \text{foo() } \{ \ldots \}
\}
\]

\[
\text{void } \text{dispatch() } \{ \\
\quad \text{A } x = \text{new } B(); \\
\quad x.\text{foo();} \\
\quad \text{A } y = \text{new } C(); \\
\quad y.\text{foo();} \\
\}
\]

\[
\text{Dispatch}(c, m) = \begin{cases} 
\quad m', \quad \text{if } c \text{ contains non-abstract method } m' \text{ that has the same name and descriptor as } m \\
\quad \text{Dispatch}(c', m), \quad \text{otherwise}
\end{cases}
\]

\[
\text{where } c' \text{ is superclass of } c
\]

\[
\text{Dispatch}(B, A.\text{foo()}) = A.\text{foo()}
\]

\[
\text{Dispatch}(C, A.\text{foo()}) = ?
\]
Dispatch: An Example

void dispatch() {
    A x = new B();
    x.foo();
    A y = new C();
    y.foo();
}

Dispatch(c, m) =
\[
\begin{cases}
    m', & \text{if } c \text{ contains non-abstract method } m' \text{ that has the same name and descriptor as } m \\
    \text{Dispatch}(c', m), & \text{otherwise}
\end{cases}
\]
where $c'$ is superclass of $c$

Dispatch(B, A.foo()) = A.foo()  
Dispatch(C, A.foo()) = C.foo()
Class Hierarchy Analysis* (CHA)

• Require the class hierarchy information (inheritance structure) of the whole program

• Resolve a virtual call based on the declared type of receiver variable of the call site

\[
\text{A a = ...} \\
\text{a. foo();}
\]

Class Hierarchy Analysis* (CHA)

• Require the class hierarchy information (inheritance structure) of the whole program

• Resolve a virtual call based on the declared type of receiver variable of the call site

```
A a = ...
a.foo();
```

• Assume the receiver variable `a` may point to objects of class `A` or all subclasses of `A`
  • Resolve target methods by looking up the class hierarchy of class `A`

Class Hierarchy Analysis* (CHA)

• Require the class hierarchy information (inheritance structure) of the whole program

• Resolve a virtual call based on the declared type of receiver variable of the call site

  A a = ...
  a. foo();

• Assume the receiver variable a may point to objects of class A or all subclasses of A
  • Resolve target methods by looking up the class hierarchy of class A

We define function \textbf{Resolve}(cs) to resolve possible target methods of a call site \textit{cs} by CHA

\begin{verbatim}
Resolve(cs)
    T = {}
    m = method signature at cs
    if \textit{cs} is a static call then
        T = { m }
    if \textit{cs} is special call then
        \textit{c}^m = class type of \textit{m}
        T = { \textbf{Dispatch}(\textit{c}^m, \textit{m}) }
    if \textit{cs} is a virtual call then
        \textit{c} = declared type of receiver variable at \textit{cs}
        foreach \textit{c}' that is a subclass of \textit{c} or \textit{c} itself do
            add \textbf{Dispatch}(\textit{c}', \textit{m}) to \textit{T}
    return \textit{T}
\end{verbatim}
Call Resolution of CHA

We define function **Resolve(cs)** to resolve possible target methods of a call site *cs* by CHA

```
class C {
    static T foo(P p, Q q)
    {...
}
}
C.foo(x, y);
```

```
T = {}

m = method signature at cs

if *cs* is a static call then
    T = { m }

if *cs* is special call then
    cm = class type of m
    T = { Dispatch(cm, m) }

if *cs* is a virtual call then
    c = declared type of receiver variable at cs
    foreach c' that is a subclass of c or c itself do
        add Dispatch(c', m) to T

return T
```
Call Resolution of CHA

We define function \textbf{Resolve}(cs) to resolve possible target methods of a call site \textit{cs} by CHA

\begin{verbatim}
Resolve(cs)

\begin{align*}
T &= \{ \} \\
\text{if } cs \text{ is a static call then} &\quad T = \{ m \} \\
\text{if } cs \text{ is special call then} &\quad c^m = \text{class type of } m \\
&\quad T = \{ \text{Dispatch}(c^m, m) \} \\
\text{if } cs \text{ is a virtual call then} &\quad c = \text{declared type of receiver variable at } cs \\
&\quad \text{foreach } c' \text{ that is a subclass of } c \text{ or } c \text{ itself do} \\
&\quad \text{add } \text{Dispatch}(c', m) \text{ to } T \\
\text{return } T
\end{align*}
\end{verbatim}

\text{class } C \text{ extends } B \{ \\
\text{T } \text{foo(P p, Q q) } \{ \\
&\text{...} \\
&\text{super.\text{foo}(p, q);} \\
\}
\}

\text{cs super.\text{foo}(p, q);} \\
\text{m } <B: \text{ T foo(P,Q)>} \\
\text{c}^m \text{ B}
Call Resolution of CHA

We define function $\text{Resolve}(cs)$ to resolve possible target methods of a call site $cs$ by CHA

$$
\text{Resolve}(cs) = \begin{cases} 
T = \{ \} & 
\text{if } cs \text{ is a static call then} \\
T = \{ m \} & 
\text{if } cs \text{ is a special call then} \\
T = \{ \text{Dispatch}(c^m, m) \} & 
\text{if } cs \text{ is a virtual call then} \\
& \quad \text{c = declared type of receiver variable at } cs \\
& \quad \text{foreach } c' \text{ that is a subclass of } c \text{ or } c \text{ itself do} \\
& \quad \text{add } \text{Dispatch}(c', m) \text{ to } T \\
\end{cases}
$$

return $T$

class C extends B {
    T foo(P p, Q q) {
        ...
        super.foo(p, q);
    }
}

A
\text{foo()}

B
\text{foo()}

C

$cs$ super.foo(p, q);

$m$ <B: T foo(P,Q)>

$c^m$ B
Call Resolution of CHA

We define function **Resolve**(cs) to resolve possible target methods of a call site cs by CHA

```java
class C extends B {
    T foo(P p, Q q) {
        ...
        this.bar();
    }
    private T bar()
}
C c = new C();
```

Special call
- Private instance method
- Constructor
- Superclass instance method

Resolve(cs)

\[
T = \{
\}
\]

\[m = \text{method signature at } cs\]

**if** cs is a static call **then**

\[
T = \{ m \}
\]

**if** cs is a special call **then**

\[c^m = \text{class type of } m\]

\[
T = \{ \text{Dispatch}(c^m, m) \}
\]

**if** cs is a virtual call **then**

\[c = \text{declared type of receiver variable at } cs\]

**foreach** \(c'\) that is a subclass of \(c\) or \(c\) itself **do**

add \(\text{Dispatch}(c', m)\) to \(T\)

**return** \(T\)
Call Resolution of CHA

We define function **Resolve**(*cs*) to resolve possible target methods of a call site *cs* by CHA.

**Resolve**(*cs*)

\[ T = \{ \} \]

\[ m = \text{method signature at } cs \]

**if** *cs* is a **static call** **then**

\[ T = \{ m \} \]

**if** *cs* is a **special call** **then**

\[ c^m = \text{class type of } m \]

\[ T = \{ \text{Dispatch} (c^m, m) \} \]

**if** *cs* is a **virtual call** **then**

\[ c = \text{declared type of receiver variable at } cs \]

**foreach** \( c' \) that is a subclass of \( c \) or \( c \) itself **do**

add \( \text{Dispatch} (c', m) \) to \( T \)

return \( T \)
Call Resolution of CHA

We define function $\text{Resolve}(cs)$ to resolve possible target methods of a call site $cs$ by CHA.

\begin{align*}
\text{Resolve}(cs) &= T = \{ \} \\
& \quad m = \text{method signature at } cs \\
& \quad \text{if } cs \text{ is a static call then} \\
& \qquad T = \{ m \} \\
& \quad \text{if } cs \text{ is special call then} \\
& \qquad c^m = \text{class type of } m \\
& \qquad T = \{ \text{Dispatch}(c^m, m) \} \\
& \quad \text{if } cs \text{ is a virtual call then} \\
& \qquad c = \text{declared type of receiver variable at } cs \\
& \qquad \text{foreach } c' \text{ that is a subclass of } c \text{ or } c \text{ itself do} \\
& \qquad \quad \text{add } \text{Dispatch}(c', m) \text{ to } T \\
& \text{return } T
\end{align*}

Subclasses includes all direct and indirect subclasses of $c$.
class A {
    void foo() {...}
}
class B extends A {}

class C extends B {
    void foo() {...}
}
class D extends B {
    void foo() {...}
}

void resolve() {
    C c = ...
    c.foo();

    A a = ...
    a.foo();

    B b = ...
    b.foo();
}
CHA: An Example

class A {
    void foo() {...}
}
class B extends A {}

class C extends B {
    void foo() {...}
}
class D extends B {
    void foo() {...}
}

void resolve() {
    C c = ...
    c.foo(); ?? Resolve(c.foo()) = ??

    A a = ...
    a.foo();

    B b = ...
    b.foo();
}
class A {
    void foo() {...}
}
class B extends A {}

class C extends B {
    void foo() {...}
}
class D extends B {
    void foo() {...}
}

void resolve() {
    C c = ...
    c.foo();

    Resolve(c.foo()) = {C.foo()}

    A a = ...
    a.foo();

    Resolve(a.foo()) = ?

    B b = ...
    b.foo();

}
class A {
    void foo() {...}
}
class B extends A {}

class C extends B {
    void foo() {...}
}
class D extends B {
    void foo() {...}
}

void resolve() {
    C c = ...
    c.foo();
    Resolve(c.foo()) = { C.foo() }

    A a = ...
    a.foo();
    Resolve(a.foo()) = { A.foo(), C.foo(), D.foo() }

    B b = ...
    b.foo();
    Resolve(b.foo()) = ?
}
class A {
    void foo() {...}
}
class B extends A {}

class C extends B {
    void foo() {...}
}
class D extends B {
    void foo() {...}
}

void resolve() {
    C c = ...
    c.foo();
    Resolve(c.foo()) = {C.foo()}

    A a = ...
    a.foo();
    Resolve(a.foo()) = {A.foo(), C.foo(), D.foo()}

    B b = ...
    b.foo();
    Resolve(b.foo()) = {A.foo(), C.foo(), D.foo()}
}
### CHA: An Example

```java
class A {
    void foo() {...}
}
class B extends A {}

class C extends B {
    void foo() {...}
}
class D extends B {
    void foo() {...}
}

void resolve() {
    C c = ...
    c.foo();
    A a = ...
    a.foo();
    B b = new B();
    b.foo();
}
```

- **Resolve**\(c\).foo()\) = \{C.foo()\}
- **Resolve**\(a\).foo()\) = \{A.foo(), C.foo(), D.foo()\}
- **Resolve**\(b\).foo()\) = ?

Tian Tan @ Nanjing University
class A {
    void foo() {...}
}
class B extends A {}
class C extends B {
    void foo() {...}
}
class D extends B {
    void foo() {...}
}

void resolve() {
    C c = ...
c.foo();
    A a = ...
a.foo();
    B b = new B();
b.foo();

    Resolve(c.foo()) = {C.foo()}
    Resolve(a.foo()) = {A.foo(), C.foo(), D.foo()}
    Resolve(b.foo()) = {A.foo(), C.foo(), D.foo()}

    Spurious call targets
Features of CHA

• Advantage: fast
  • Only consider the declared type of receiver variable at the call-site, and its inheritance hierarchy
  • Ignore data- and control-flow information
Features of CHA

• Advantage: fast
  • Only consider the declared type of receiver variable at the call-site, and its inheritance hierarchy
  • Ignore data- and control-flow information

• Disadvantage: imprecise
  • Easily introduce spurious target methods
  • Addressed in next lectures
Features of CHA

• Advantage: fast
  • Only consider the declared type of receiver variable at the call-site, and its inheritance hierarchy
  • Ignore data- and control-flow information

• Disadvantage: imprecise
  • Easily introduce spurious target methods
  • Addressed in next lectures

Common usage: IDE
CHA in IDE (IntelliJ IDEA)
Call Graph Construction

Build call graph for whole program via CHA

• Start from entry methods (focus on main method)
• For each reachable method $m$, resolve target methods for each call site $cs$ in $m$ via CHA ($\text{Resolve}(cs)$)
• Repeat until no new method is discovered
Call Graph Construction: Algorithm

```
BuildCallGraph(m^{entry})
WL = [m^{entry}], CG = {}, RM = {}
while WL is not empty do
    remove m from WL
    if m \notin RM then
        add m to RM
    foreach call site cs in m do
        T = Resolve(cs)
        foreach target method m' in T do
            add cs \rightarrow m' to CG
            add m' to WL
    return CG
```

- **WL**: Work list, containing the methods to be processed
- **CG**: Call graph, a set of call edges
- **RM**: A set of reachable methods
Call Graph Construction: Algorithm

**BuildCallGraph**($m_{entry}$)

```plaintext
WL = [$m_{entry}$], $CG = {}$, $RM = {}$  
while $WL$ is not empty do
    remove $m$ from $WL$
    if $m \notin RM$ then
        add $m$ to $RM$
        foreach call site $cs$ in $m$ do
            $T = \text{Resolve}(cs)$
            foreach target method $m'$ in $T$ do
                add $cs \rightarrow m'$ to $CG$
                add $m'$ to $WL$
        return $CG$
```

**WL** Work list, containing the methods to be processed

**CG** Call graph, a set of call edges

**RM** A set of reachable methods
Call Graph Construction: Algorithm

**BuildCallGraph**(m<sub>entry</sub>)

\[ WL = [m_{entry}], \ CG = \{\}, \ RM = \{\} \]

**while** \( WL \) is not empty **do**

\[ \text{remove } m \text{ from } WL \]

**if** \( m \not\in RM \) **then**

\[ \text{add } m \text{ to } RM \]

**foreach** call site \( cs \) in \( m \) **do**

\[ T = \text{Resolve}(cs) \]

**foreach** target method \( m' \) in \( T \) **do**

\[ \text{add } cs \rightarrow m' \text{ to } CG \]

\[ \text{add } m' \text{ to } WL \]

**return** \( CG \)

---

**WL**  Work list, containing the methods to be processed

**CG**  Call graph, a set of call edges

**RM**  A set of reachable methods
Call Graph Construction: Algorithm

**BuildCallGraph**(\(m^{entry}\))

\[WL = [m^{entry}], \ CG = {}, \ RM = {}\]

while \(WL\) is not empty do

remove \(m\) from \(WL\)

if \(m \not\in RM\) then

add \(m\) to \(RM\)

foreach call site \(cs\) in \(m\) do

\(T = \text{Resolve}(cs)\)

foreach target method \(m'\) in \(T\) do

add \(cs \rightarrow m'\) to \(CG\)

add \(m'\) to \(WL\)

return \(CG\)

---

**Variables**

- **\(WL\)**: Work list, containing the methods to be processed
- **\(CG\)**: Call graph, a set of call edges
- **\(RM\)**: A set of reachable methods

---

Tian Tan @ Nanjing University 55
Call Graph Construction: Algorithm

\begin{align*}
\text{BuildCallGraph}(m_{\text{entry}}) & \quad \text{Initialize the algorithm} \\
WL = [m_{\text{entry}}], \ CG = \{\}, \ RM = \{} & \\
\text{while } WL \text{ is not empty do} & \\
\text{remove } m \text{ from } WL & \\
\text{if } m \notin RM \text{ then} & \\
\text{add } m \text{ to } RM & \\
\text{foreach call site } cs \text{ in } m \text{ do} & \\
T = \text{Resolve}(cs) & \\
\text{foreach target method } m' \text{ in } T \text{ do} & \\
\text{add } cs \rightarrow m' \text{ to } CG & \\
\text{add } m' \text{ to } WL & \\
\text{return } CG & \\
\end{align*}

- **WL**: Work list, containing the methods to be processed
- **CG**: Call graph, a set of call edges
- **RM**: A set of reachable methods

Tian Tan @ Nanjing University
Call Graph Construction: An Example

```java
class A {
    static void main() {
        A.foo();
    }
    static void foo() {
        A a = new A();
        a.bar();
    }
    void bar() {
        C c = new C();
        c.bar();
    }
}

class B extends A {
    void bar() {}
}
class C extends A {
    void bar() {
        if (...) A.foo();
    }
    void m() {}
}
```

Initialization with main method

\[ WL = [A.main()] \]
Call Graph Construction: An Example

class A {
    static void main() {
        A.foo();
    }
    static void foo() {
        A a = new A();
        a.bar();
    }
    void bar() {
        C c = new C();
        c.bar();
    }
}

class B extends A {
    void bar() {}
}
class C extends A {
    void bar() {
        if (...) A.foo();
    }
    void m() {}
}

\[
\begin{align*}
WL &= [ ] \\
\text{Resolve}(A.\text{foo}) &= ?
\end{align*}
\]
Call Graph Construction: An Example

class A {
    static void main() {
        A.foo();
    }
    static void foo() {
        A a = new A();
        a.bar();
    }
    void bar() {
        C c = new C();
        c.bar();
    }
}
class B extends A {
    void bar() {}}
class C extends A {
    void bar() {
        if (...) A.foo();
    }
    void m() {}
}

Resolve(A.foo()) = { A.foo() }

WL = [A.foo()]
class A {
    static void main() {
        A.foo();
    }
    static void foo() {
        A a = new A();
        a.bar();
    }
    void bar() {
        C c = new C();
        c.bar();
    }
}
class B extends A {
    void bar() {} }
class C extends A {
    void bar() {
        if (...) A.foo();
    }
    void m() {}
}
class A {
    static void main() {
        A.foo();
    }
    static void foo() {
        A a = new A();
        a.bar();
    }
    void bar() {
        C c = new C();
        c.bar();
    }
}

class B extends A {
    void bar() {}}

class C extends A {
    void bar() {
        if (...) A.foo();
    }
    void m() {}
}
Call Graph Construction: An Example

class A {
    static void main() {
        A.foo();
    }

    static void foo() {
        A a = new A();
        a.bar();
    }

    void bar() {
        C c = new C();
        c.bar();
    }
}

class B extends A {
    void bar() {}
}
class C extends A {
    void bar() {
        if (...) A.foo();
    }
    void m() {}
}

WL = [ ]

Resolve(a.bar()) = ?
Call Graph Construction: An Example

class A {
    static void main() {
        A.foo();
    }

    static void foo() {
        A a = new A();
        a.bar();
    }

    void bar() {
        C c = new C();
        c.bar();
    }
}

class B extends A {
    void bar() {} }

class C extends A {
    void bar() {
        if (...) A.foo();
    }

    void m() {}
}

WL = [A.bar(), B.bar(), C.bar()]

Resolve(a.bar()) = { A.bar(), B.bar(), C.bar() }
class A {
    static void main() {
        A.foo();
    }
    static void foo() {
        A a = new A();
        a.bar();
    }
    void bar() {
        C c = new C();
        c.bar();
    }
}

class B extends A {
    void bar() {}
}

class C extends A {
    void bar() {
        if (...) A.foo();
    }
    void m() {}
}

WL = [B.bar(), C.bar()]

Resolve(c.bar()) = ?
class A {
    static void main() {
        A.foo();
    }
    static void foo() {
        A a = new A();
        a.bar();
    }
    void bar() {
        C c = new C();
        c.bar();
    }
}

class B extends A {
    void bar() {}
}
class C extends A {
    void bar() {
        if (...) A.foo();
    }
    void m() {}
}
Call Graph Construction: An Example

class A {
    static void main() {
        A.foo();
    }
    static void foo() {
        A a = new A();
        a.bar();
    }
    void bar() {
        C c = new C();
        c.bar();
    }
}
class B extends A {
    void bar() {}
} class C extends A {
    void bar() {
        if (...) A.foo();
    }
    void m() {}
}
class A {
    static void main() {
        A.foo();
    }
    static void foo() {
        A a = new A();
        a.bar();
    }
    void bar() {
        C c = new C();
        c.bar();
    }
}
class B extends A {
    void bar() {}}
class C extends A {
    void bar() {
        if (...) A.foo();
    }
    void m() {}
}

Call Graph Construction: An Example

WL = [C.bar()]
class A {
    static void main() {
        A.foo();
    }
    static void foo() {
        A a = new A();
        a.bar();
    }
    void bar() {
        C c = new C();
        c.bar();
    }
}

class B extends A {
    void bar() {}
}
class C extends A {
    void bar() {
        if (...) A.foo();
    }
    void m() {}
}

WL = [C.bar(), A.foo()]

Resolve(A.foo()) = { A.foo() }
class A {
    static void main() {
        A.foo();
    }
    static void foo() {
        A a = new A();
        a.bar();
    }
    void bar() {
        C c = new C();
        c.bar();
    }
}
class B extends A {
    void bar() {}
}
class C extends A {
    void bar() {
        if (...) A.foo();
    }
    void m() {}
}
class A {
    static void main() {
        A.foo();
    }
    static void foo() {
        A a = new A();
        a.bar();
    }
    void bar() {
        C c = new C();
        c.bar();
    }
}
class B extends A {
    void bar() {} }
class C extends A {
    void bar() {
        if (...) A.foo();
    }
    void m() {}
}
class A {
    static void main() {
        A.foo();
    }
    static void foo() {
        A a = new A();
a.bar();
    }
    void bar() {
        C c = new C();
c.bar();
    }
}

class B extends A {
    void bar() {} }

class C extends A {
    void bar() {
        if (...) A.foo();
    }
    void m() {} }

WL = [ ]
Call Graph Construction: An Example

class A {
    static void main() {
        A.foo();
    }
    static void foo() {
        A a = new A();
        a.bar();
    }
    void bar() {
        C c = new C();
        c.bar();
    }
}
class B extends A {
    void bar() {} }
class C extends A {
    void bar() {
        if (...) A.foo();
    }
    void m() {}
}
Contents

1. Motivation
2. Call Graph Construction (CHA)
3. Interprocedural Control-Flow Graph
4. Interprocedural Data-Flow Analysis
Interprocedural Control-Flow Graph

• CFG represents structure of an individual method
• ICFG represents structure of the whole program
  • With ICFG, we can perform interprocedural analysis
Interprocedural Control-Flow Graph

- CFG represents structure of an individual method
- ICFG represents structure of the whole program
  - With ICFG, we can perform interprocedural analysis
- An ICFG of a program consists of CFGs of the methods in the program, plus two kinds of additional edges:
  - **Call edges**: from call sites to the entry nodes of their callees
  - **Return edges**: from return statements of the callees to the statements following their call sites (i.e., return sites)
Interprocedural Control-Flow Graph

• CFG represents structure of an individual method
• ICFG represents structure of the whole program
  • With ICFG, we can perform interprocedural analysis
• An ICFG of a program consists of CFGs of the methods in the program, plus two kinds of additional edges:
  ➢ Call edges: from call sites to the entry nodes of their callees
  ➢ Return edges: from return statements of the callees to the statements following their call sites (i.e., return sites)

```c
void foo() {
    bar(...);  // call site
    int n = 3;  // return site
}
```
Interprocedural Control-Flow Graph

• CFG represents structure of an individual method
• ICFG represents structure of the whole program
  • With ICFG, we can perform interprocedural analysis
• An ICFG of a program consists of CFGs of the methods in the program, plus two kinds of additional edges:
  ➢ Call edges: from call sites to the entry nodes of their callees
  ➢ Return edges: from return statements of the callees to the statements following their call sites (i.e., return sites)

ICFG = CFGs + call & return edges

The information for connecting these two kinds of edges comes from call graph
static void main() {
    int a, b, c;
    a = 6;
    b = addOne(a);
    c = b - 3;
    b = ten();
    c = a * b;
}

int addOne(int x) {
    int y = x + 1;
    return y;
}

int ten() {
    return 10;
}
ICFG: An Example

```java
static void main() {
    int a, b, c;
    a = 6;
    b = addOne(a);
    c = b - 3;
    b = ten();
    c = a * b;
}

int addOne(int x) {
    int y = x + 1;
    return y;
}

int ten() {
    return 10;
}
```

ICFG = CFGs + call & return edges
static void main() {
    int a, b, c;
    a = 6;
    b = addOne(a);
    c = b - 3;
    b = ten();
    c = a * b;
}

int addOne(int x) {
    int y = x + 1;
    return y;
}

int ten() {
    return 10;
}

ICFG = CFGs + call & return edges
Contents

1. Motivation
2. Call Graph Construction (CHA)
3. Interprocedural Control-Flow Graph
4. Interprocedural Data-Flow Analysis
Interprocedural Data-Flow Analysis

Analyzing the whole program with method calls based on interprocedural control-flow graph (ICFG)

<table>
<thead>
<tr>
<th></th>
<th>Intra-procedural</th>
<th>Inter-procedural</th>
</tr>
</thead>
<tbody>
<tr>
<td>Program representation</td>
<td>CFG</td>
<td>ICFG = CFGs + call &amp; return edges</td>
</tr>
</tbody>
</table>
Interprocedural Data-Flow Analysis

Analyzing the whole program with method calls based on interprocedural control-flow graph (ICFG)

<table>
<thead>
<tr>
<th></th>
<th>Intraprocedural</th>
<th>Interprocedural</th>
</tr>
</thead>
<tbody>
<tr>
<td>Program representation</td>
<td>CFG</td>
<td>ICFG = CFGs + call &amp; return edges</td>
</tr>
<tr>
<td>Transfer functions</td>
<td>Node transfer</td>
<td>Node transfer + edge transfer</td>
</tr>
</tbody>
</table>
Interprocedural Data-Flow Analysis

Analyzing the whole program with method calls based on interprocedural control-flow graph (ICFG)

<table>
<thead>
<tr>
<th></th>
<th>Intra-procedural</th>
<th>Inter-procedural</th>
</tr>
</thead>
<tbody>
<tr>
<td>Program representation</td>
<td>CFG</td>
<td>ICFG = CFGs + call &amp; return edges</td>
</tr>
<tr>
<td>Transfer functions</td>
<td>Node transfer</td>
<td>Node transfer + edge transfer</td>
</tr>
</tbody>
</table>

Edge transfer

- **Call edge transfer**: transfer data flow from call node to the entry node of callee (along call edges)
- **Return edge transfer**: transfer data flow from return node of the callee to the return site (along return edges)
Interprocedural Constant Propagation

• **Call edge transfer**: pass argument values
• **Return edge transfer**: pass return values

• **Node transfer**: same as intra-procedural constant propagation, plus that
  • For each call node, kill data-flow value for the LHS variable. Its value will flow to return site along the return edges

```
a=..., b=...

b = addOne(a)
```

Tian Tan @ Nanjing University
Interprocedural Constant Propagation: An Example

```c
static void main() {
    int a, b, c;
    a = 6;
    b = addOne(a);
    c = b - 3;
    b = ten();
    c = a * b;
}

static int addOne(int x) {
    int y = x + 1;
    return y;
}

static int ten() {
    return 10;
}
```

Tian Tan @ Nanjing University
Interprocedural Constant Propagation: An Example

```java
static void main() {
    int a, b, c;
    a = 6;
    b = addOne(a);
    c = b - 3;
    b = ten();
    c = a * b;
}

static int addOne(int x) {
    int y = x + 1;
    return y;
}

static int ten() {
    return 10;
}
```
Interprocedural Constant Propagation: An Example

```java
static void main() {
    int a, b, c;
    a = 6;
    b = addOne(a);
    c = b - 3;
    b = ten();
    c = a * b;
}

static int addOne(int x) {
    int y = x + 1;
    return y;
}

static int ten() {
    return 10;
}
```

Tian Tan @ Nanjing University
Interprocedural Constant Propagation: An Example

```java
static void main() {
    int a, b, c;
    a = 6;
    b = addOne(a);
    c = b - 3;
    b = ten();
    c = a * b;
}

static int addOne(int x) {
    int y = x + 1;
    return y;
}

static int ten() {
    return 10;
}
```

Tian Tan @ Nanjing University
Interprocedural Constant Propagation: An Example

```java
static void main() {
    int a, b, c;
    a = 6;
    b = addOne(a);
    c = b - 3;
    b = ten();
    c = a * b;
}

static int addOne(int x) {
    int y = x + 1;
    return y;
}

static int ten() {
    return 10;
}
```

Tian Tan @ Nanjing University
Interprocedural Constant Propagation: An Example

```java
static void main() {
    int a, b, c;
    a = 6;
    b = addOne(a);
    c = b - 3;
    b = ten();
    c = a * b;
}

static int addOne(int x) {
    int y = x + 1;
    return y;
}

static int ten() {
    return 10;
}
```

Tian Tan @ Nanjing University
Interprocedural Constant Propagation: An Example

```java
static void main() {
    int a, b, c;
    a = 6;
    b = addOne(a);
    c = b - 3;
    b = ten();
    c = a * b;
}

static int addOne(int x) {
    int y = x + 1;
    return y;
}

static int ten() {
    return 10;
}
```

Tian Tan @ Nanjing University
Interprocedural Constant Propagation: An Example

static void main() {
    int a, b, c;
    a = 6;
    b = addOne(a);
    c = b - 3;
    b = ten();
    c = a * b;
}

static int addOne(int x) {
    int y = x + 1;
    return y;
}

static int ten() {
    return 10;
}
Interprocedural Constant Propagation: An Example

```java
static void main() {
    int a, b, c;
    a = 6;
    b = addOne(a);
    c = b - 3;
    b = ten();
    c = a * b;
}

static int addOne(int x) {
    int y = x + 1;
    return y;
}

static int ten() {
    return 10;
}
```

Tian Tan @ Nanjing University
static void main() {
    int a, b, c;
    a = 6;
    b = addOne(a);
    c = b - 3;
    b = ten();
    c = a * b;
}

static int addOne(int x) {
    int y = x + 1;
    return y;
}

static int ten() {
    return 10;
}

Such edge (from call site to return site) is named call-to-return edge. It allows the analysis to propagate local data-flow (a=6 in this case) on ICFG.
static void main() {
    int a, b, c;
    a = 6;
    b = addOne(a);
    c = b - 3;
    b = ten();
    c = a * b;
}

static int addOne(int x) {
    int y = x + 1;
    return y;
}

static int ten() {
    return 10;
}

Such edge (from call site to return site) is named call-to-return edge. It allows the analysis to propagate local data-flow (a=6 in this case) on ICFG.

Without such edges, we have to propagate local data-flow across other methods, which is very inefficient.
Interprocedural Constant Propagation: An Example

```java
static void main() {
    int a, b, c;
    a = 6;
    b = addOne(a);
    c = b - 3;
    b = ten();
    c = a * b;
}

static int addOne(int x) {
    int y = x + 1;
    return y;
}
```

Such edge (from call site to return site) is named **call-to-return edge**. It allows the analysis to propagate local data-flow (a=6 in this case) on ICFG.

Without such edges, we have to propagate local data-flow across other methods, which is very inefficient.
static void main() {
    int a, b, c;
    a = 6;
    b = addOne(a);
    c = b - 3;
    b = ten();
    c = a * b;
}

static int addOne(int x) {
    int y = x + 1;
    return y;
}

static int ten() {
    return 10;
}
Interprocedural Constant Propagation: An Example

```java
static void main() {
    int a, b, c;
    a = 6;
    b = addOne(a);
    c = b - 3;
    b = ten();
    c = a * b;
}

static int addOne(int x) {
    int y = x + 1;
    return y;
}

static int ten() {
    return 10;
}
```

Tian Tan @ Nanjing University
Interprocedural Constant Propagation: An Example

```java
static void main() {
    int a, b, c;
    a = 6;
    b = addOne(a);
    c = b - 3;
    b = ten();
    c = a * b;
}

static int addOne(int x) {
    int y = x + 1;
    return y;
}

static int ten() {
    return 10;
}
```

For each call node, kill data-flow value of the LHS variable. Its value will flow to return site along the return edges.
Interprocedural Constant Propagation: An Example

```java
static void main() {
    int a, b, c;
    a = 6;
    b = addOne(a);
    c = b - 3;
    b = ten();
    c = a * b;
}

static int addOne(int x) {
    int y = x + 1;
    return y;
}

static int ten() {
    return 10;
}
```

For each call node, kill data-flow value of the LHS variable. Its value will flow to return site along the return edges.
Interprocedural Constant Propagation: An Example

```java
static void main() {
    int a, b, c;
    a = 6;
    b = addOne(a);
    c = b - 3;
    b = ten();
    c = a * b;
}

static int addOne(int x) {
    int y = x + 1;
    return y;
}

static int ten() {
    return 10;
}
```

For each call node, kill data-flow value of the LHS variable. Its value will flow to return site along the return edges.
Interprocedural Constant Propagation: An Example

```java
static void main() {
    int a, b, c;
    a = 6;
    b = addOne(a);
    c = b - 3;
    b = ten();
    c = a * b;
}

static int addOne(int x) {
    int y = x + 1;
    return y;
}

static int ten() {
    return 10;
}
```

For each call node, kill data-flow value of the LHS variable. Its value will flow to return site along the return edges. Otherwise, it may cause imprecision.

Tian Tan @ Nanjing University
Interprocedural Constant Propagation: An Example

```java
static void main() {
    int a, b, c;
    a = 6;
    b = addOne(a);
    c = b - 3;
    b = ten();
    c = a * b;
}

static int addOne(int x) {
    int y = x + 1;
    return y;
}

static int ten() {
    return 10;
}
```
**Intra-procedural Constant Propagation: An Example**

```java
static void main() {
    int a, b, c;
    a = 6;
    b = addOne(a);
    c = b - 3;
    b = ten();
    c = a * b;
}

static int addOne(int x) {
    int y = x + 1;
    return y;
}

static int ten() {
    return 10;
}
```

The diagram shows the flow of values during constant propagation. The variables `a`, `b`, and `c` are assigned values and operations are performed, with the results being propagated through the function calls `addOne()` and `ten()`.

**Flow Diagram**:
- **IN Flow**: Arrow indicating input flow.
- **OUT Flow**: Arrow indicating output flow.
- **NAC**: Not a Constant, indicating the variable value is not known.

**Intra-procedural Analysis**:
- `a = 6`
- `b = addOne(a)`
- `c = b - 3`
- `b = ten()`
- `c = a * b`
static void main() {
    int a, b, c;
    a = 6;
    b = addOne(a);
    c = b - 3;
    b = ten();
    c = a * b;
}

static int addOne(int x) {
    int y = x + 1;
    return y;
}

static int ten() {
    return 10;
}

Interprocedural constant propagation is more precise than Intraprocedural constant propagation
The X You Need To Understand in This Lecture

- How to build call graph via class hierarchy analysis
- Concept of interprocedural control-flow graph
- Concept of interprocedural data-flow analysis
- Interprocedural constant propagation

注意注意！
划重点了！
南京大学
计算机科学与技术系
程序设计语言与
静态分析研究组
李樾
谭添