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

Tian Tan @ Nanjing University
1. Motivation
2. Call Graph Construction (CHA)
3. Interprocedural Control-Flow Graph
4. Interprocedural Data-Flow Analysis
Motivation of Interprocedural Analysis

Constant Propagation

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;
}
```
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
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**
  - $x = \text{NAC}$, $y = \text{NAC}$
  - $n = \text{NAC}$

Tian Tan @ Nanjing University
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)
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
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)

- \( x = 42, \ y = 43 \)
- \( n = 10 \)

To perform interprocedural analysis, we need **call graph**
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 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

this lecture

next lectures

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></th>
<th>Static call</th>
<th>Special call</th>
<th>Virtual call</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instruction</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,</td>
<td>• Other instance methods</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Private instance</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>methods,</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Superclass instance methods</td>
<td></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>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</td>
<td>• Other instance methods</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Private instance methods</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Superclass instance methods</td>
<td></td>
</tr>
<tr>
<td>#Target methods</td>
<td>1</td>
<td>1</td>
<td>≥1 (\text{polymorphism})</td>
</tr>
<tr>
<td>Determinacy</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>invokespecial</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</td>
<td>methods</td>
</tr>
<tr>
<td></td>
<td></td>
<td>instance 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>Determinacy</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
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

```cpp
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

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

<C: T foo(P, Q, R)>

• Signature = class type + method name + descriptor
• Descriptor = return type + parameter types
```

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): $c$
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 $c$

$$\langle C: T \text{ foo}(P, Q, R) \rangle$$
Dispatch: An Example

```
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();
}
```

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

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

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

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

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

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

\[
\text{Dispatch}(C, A.\text{foo}()) = ?
\]
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();
}
```

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

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

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

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

```java
A a = ...;
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

\[ A \ a = \ldots \]
\[ \( a \).\text{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 \( A \)

Call Resolution of CHA

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

$$\text{Resolve}(cs)$$

$T = \{ \}$

$m =$ method signature at $cs$

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

$T = \{ m \}$

**if** $cs$ is **special call** **then**

$c^m =$ class type of $m$

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

**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 $\text{Dispatch}(c', m)$ 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)
    \hspace{1cm} T = {}
    \hspace{1cm} m = method signature at cs
    \hspace{1cm} if \textit{cs} is a static call then
        \hspace{1cm} T = \{ m \}
    \hspace{1cm} if \textit{cs} is special call then
        \hspace{1cm} c^m = class type of \textit{m}
        \hspace{1cm} T = \{ Dispatch(c^m, m) \}
    \hspace{1cm} if \textit{cs} is a virtual call then
        \hspace{1cm} c = declared type of receiver variable at cs
        \hspace{1cm} foreach \textit{c}' that is a subclass of \textit{c} or \textit{c} itself do
            \hspace{1cm} add Dispatch(\textit{c}', \textit{m}) to \textit{T}
    \end{verbatim}

return \textit{T}

\begin{verbatim}
class C {
    static T foo(P p, Q q) {
        ...
    }
}
C.foo(x, y);  \rightarrow
\end{verbatim}

\begin{verbatim}
\textit{cs} C\text{.foo}(x, y);
\textit{m} <C: T \text{foo}(P,Q)>
\end{verbatim}
Call Resolution of CHA

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

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

\begin{verbatim}
class C extends B {
  T foo(P p, Q q) {
    ...
    super.foo(p, q);
  }
}
\end{verbatim}
Call Resolution of CHA

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

\textbf{Resolve}(cs)

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

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

\begin{verbatim}
cs super.foo(p, q);
m <B: T foo(P,Q)>
c^m B
\end{verbatim}
We define function \texttt{Resolve}(cs) to resolve possible target methods of a call site \textit{cs} by CHA.

\begin{align*}
\text{ Resolve}(cs) & \\
T &= \{ \} \\
& \text{ if \textit{cs} is a static call then} \\
& \quad T = \{ m \} \\
& \text{ if \textit{cs} is a special call then} \\
& \quad c^m = \text{class type of } m \\
& \quad T = \{ \text{Dispatch}(c^m, m) \} \\
& \text{ if \textit{cs} 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 \quad \text{add Dispatch}(c', m) \text{ to } T \\
& \text{return } T
\end{align*}

\text{class C extends B \{ \\
& \quad T \text{ foo}(P \ p, Q \ q) \{ \\
& \quad \quad \text{...} \\
& \quad \quad \text{this.bar();} \quad \leftarrow \\
& \quad \} \\
& \quad \text{private T \ bar() \} \\
& \}} 
\text{C c = new C();} \quad \leftarrow

\text{Special call}
\begin{itemize}
  \item Private instance method
  \item Constructor
  \item Superclass instance method
\end{itemize}
Call Resolution of CHA

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

```
class A {
    T foo(P p, Q q) {...}
}
A a = ...
a.foo(x, y); ←
```

```
cs a.foo(x, y);
m <A: T foo(P,Q)>
c A
```

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

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

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

\begin{verbatim}
class A {
    T foo(P p, Q q) {...}
}
A a = ...
A.a.foo(x, y);
\end{verbatim}
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();
}
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();
}

Resolve(c.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();  \textbf{Resolve}(c.\textit{foo}()) = \{C.\textit{foo}()\}

    A a = ...
    a.foo();  \textbf{Resolve}(a.\textit{foo}()) = \textbf{?}

    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();  \textbf{Resolve}(c.\texttt{foo()}) = \{C.\texttt{foo()}\}

    A a = …
    a.foo();  \textbf{Resolve}(a.\texttt{foo()}) = \{A.\texttt{foo()}, C.\texttt{foo()}, D.\texttt{foo()}\}

    B b = …
    b.foo();  \textbf{Resolve}(b.\texttt{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();       \textbf{Resolve}(c.\texttt{foo}()) = \{C.\texttt{foo}()\}

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

    B b = ...  
    b.foo();     \textbf{Resolve}(b.\texttt{foo}()) = \{A.\texttt{foo}(), C.\texttt{foo}(), D.\texttt{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();
    A a = ...
    a.foo();
    B b = new B();
    b.foo();
}

\[
\text{Resolve}(c.\text{foo}()) = \{\text{C.}\text{foo}()\}
\]
\[
\text{Resolve}(a.\text{foo}()) = \{\text{A.}\text{foo}(), \text{C.}\text{foo}(), \text{D.}\text{foo}()\}
\]
\[
\text{Resolve}(b.\text{foo}()) = ?
\]
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();
    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

Tian Tan @ Nanjing University
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

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

\begin{align*}
WL &= [m^{entry}], \ C \ G = \{\}, \ R \ M = \{} \\
\text{while} \ WL \ \text{is not empty} \ \text{do} \\
&\quad \text{remove} \ m \ \text{from} \ WL \\
&\quad \text{if} \ m \ \notin \ R \ M \ \text{then} \\
&\qquad \text{add} \ m \ \text{to} \ R \ M \\
&\qquad \text{foreach} \ \text{call site} \ cs \ \text{in} \ m \ \text{do} \\
&\qquad \quad T = \text{Resolve}(cs) \\
&\qquad \quad \text{foreach} \ \text{target method} \ m' \ \text{in} \ T \ \text{do} \\
&\qquad \qquad \text{add} \ cs \rightarrow m' \ \text{to} \ CG \\
&\qquad \qquad \text{add} \ m' \ \text{to} \ WL \\
\text{return} \ CG
\end{align*}
Call Graph Construction: Algorithm

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

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

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 \(CG\)

---

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

- **\(WL = [m^{\text{entry}}]\)**, **\(CG = {}\)**, **\(RM = {}\)** (Initialize the algorithm)
- **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

Tian Tan @ Nanjing University
Call Graph Construction: Algorithm

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

\text{Initialize the algorithm}

\text{Resolve target methods via CHA}

\text{Add call edges to call graph}

**Variables:**

- **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

\[ \text{BuildCallGraph}(m^{entry}) \]

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

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

Initialize the algorithm

Resolve target methods via CHA

Add call edges to call graph

May discover new method, add it to work list

Tian Tan @ Nanjing University
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() {}
}

A.main()
A.foo();
A.foo()
a.bar();

WL = []

Resolve(A.foo()) = ?
B.Bar()
C.bar()
A.foo()
C.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 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() {
    }
}
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() {}
}

Methods in RM

Methods not in RM
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() {}
}
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

Call Graph:
- A.main() → A.foo()
- A.foo() → a.bar();
- A.bar() → c.bar();
- C.m()
- B.Bar()
- C.bar() → A.foo();
- C.bar() → Resolve(c.bar()) = { C.bar() }

WL = [B.bar(), C.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() {}
}

Call Graph Construction: An Example

WL = [C.bar(), C.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() {}
}
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 = [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() {}
}

WL = [ ]
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() {}
}
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)

```cpp
text
void foo() {
    bar(...); // call site
    int n = 3; // return site
}
text
```
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
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;
}
```

Tian Tan @ Nanjing University
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
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

Tian Tan @ Nanjing University
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>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>
</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

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

b = addOne(a)
```

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;
}
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;
}

IN Flow
OUT Flow
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

```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
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;
}
```

IN Flow
OUT Flow

Return edge transfer: pass return values
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;
}

IN Flow
OUT Flow

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

Such edge (from call site to return site) is named \textit{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 \textit{across other methods}, which is very inefficient.

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;
}
```

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.
```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

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

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.
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
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;
}
```

Enter `main()`

```
IN Flow
a = 6
OUT Flow
```

Enter `addOne(int x)`

```
x = NAC
y = x + 1
x = NAC, y = NAC
```

Enter `ten()`

```
x = NAC
return 10
```
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 is more precise than Intra procedural 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