软件分析

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

Static Program Analysis

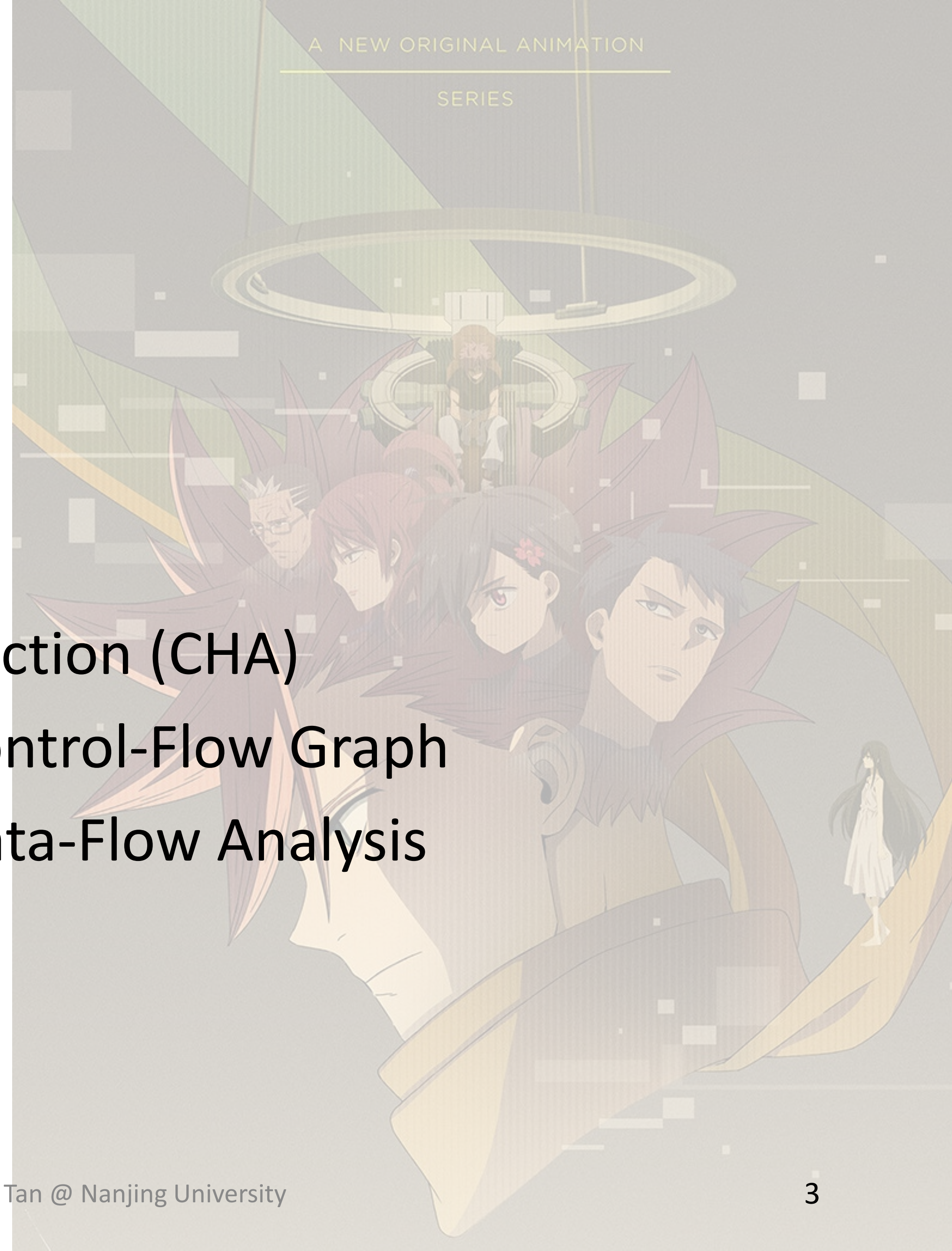
Interprocedural Analysis

Nanjing University

Tian Tan

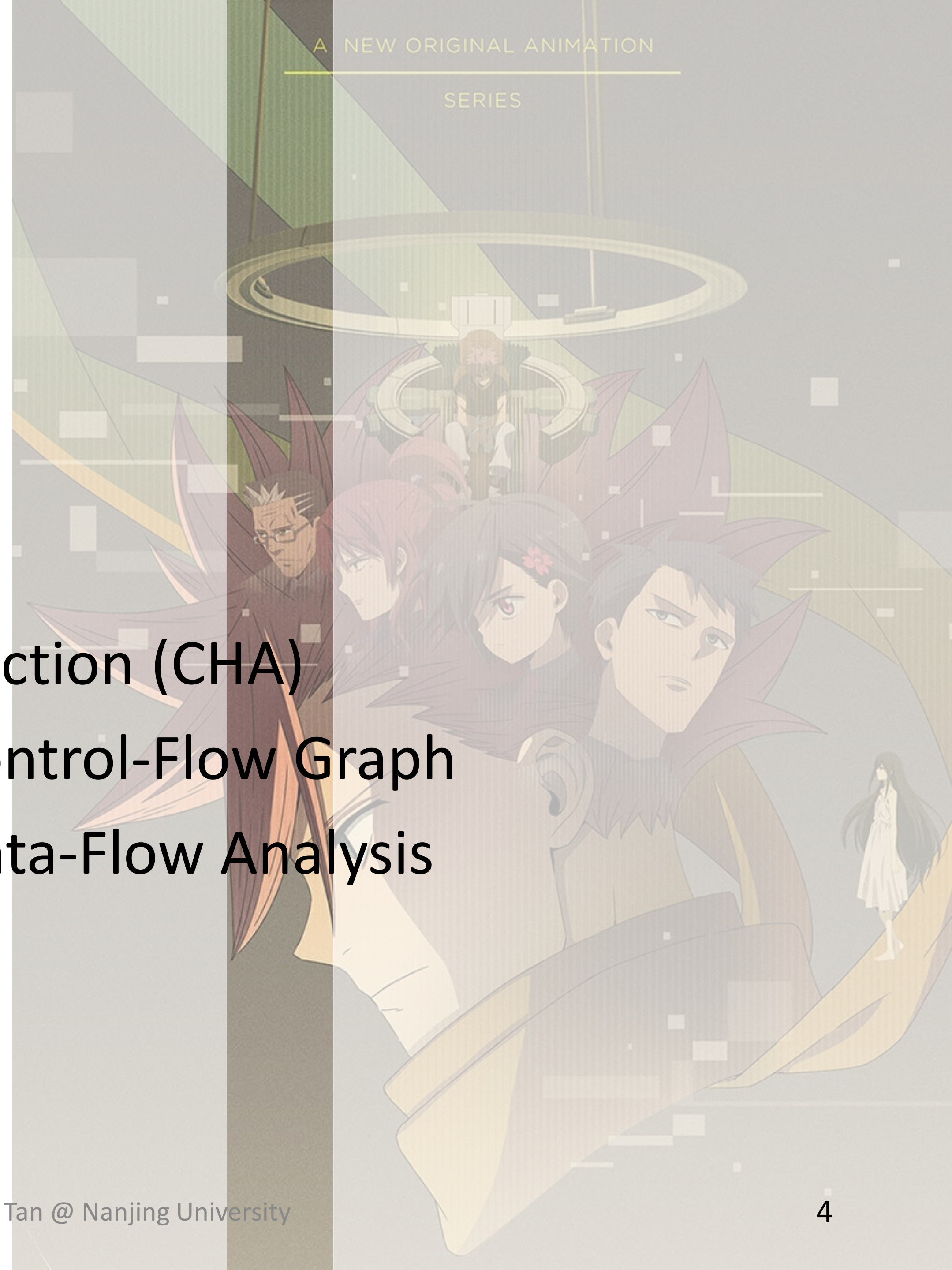
2021

Contents



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

Contents



Motivation of Interprocedural Analysis

Constant Propagation

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

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

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

Motivation of Interprocedural Analysis

Constant Propagation

```
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

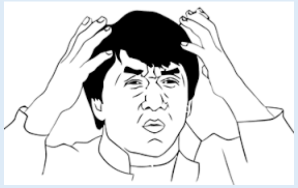
Motivation of Interprocedural Analysis

Constant Propagation

```
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

Constant Propagation

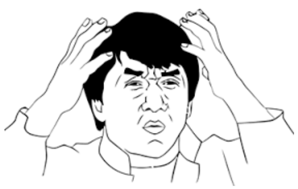
```
void foo() {
  int n = bar(42);
}

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

Call edges

Return edges

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

Constant Propagation

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

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

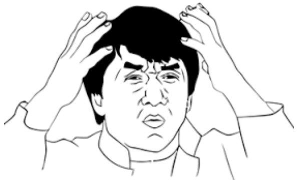
Return edges

Call edges

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

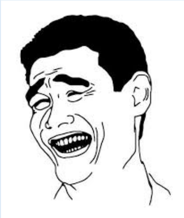
- Make the most conservative assumption for method calls, for safe-approximation
- Source of imprecision

➢ 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

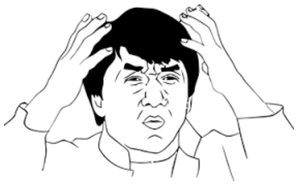
```
void foo() {
    int n = bar(42);
}

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

Return edges

Call edges

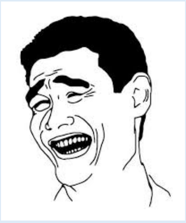
So far, all analyses we learnt are intraprocedural.
How to deal with method calls?

- 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**

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

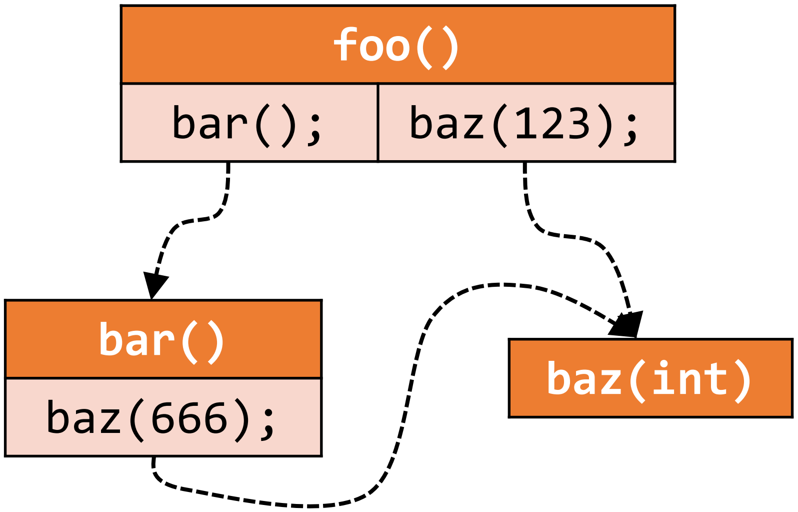
- Essentially, a call graph is a set of call edges from call-sites to their target methods (callees)

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)

```
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)

More efficient

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

More precise

Call Graph Construction for OOPLs (focus on Java)

this lecture

More efficient

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

More precise

next lectures

Method Calls (Invocations) in Java

	Static call	**Special call**	**Virtual call**
Instruction	`invokestatic`	`invokespecial`	`invokeinterface` `invokevirtual`

Method Calls (Invocations) in Java

	Static call	**Special call**	**Virtual call**
Instruction	`invokestatic`	`invokespecial`	`invokeinterface` `invokevirtual`
Receiver objects	×	✓	✓
Target methods	• Static methods	• Constructors • Private instance methods • Superclass instance methods	• Other instance methods

Method Calls (Invocations) in Java

	Static call	Special call	Virtual call
Instruction	`invokestatic`	`invokespecial`	`invokeinterface` `invokevirtual`
Receiver objects	×	✓	✓
Target methods	• Static methods	• Constructors • Private instance methods • Superclass instance methods	• Other instance methods
#Target methods	1	1	≥1 (**polymorphism**)
Determinacy	Compile-time	Compile-time	Run-time

Method Calls (Invocations) in Java

	Static call	**Special call**	**Virtual call**
Instruction	`invokestatic`	`invokespecial`	`invokeinterface` `invokevirtual`
Receiver objects	×	✓	✓
Target methods	• Static methods	• Constructors • Private instance methods • Superclass instance methods	• Other instance methods
#Target methods	1	1	≥1 (**polymorphism**)
Determinacy	Compile-time	Compile-time	Run-time

Key to call graph construction for OOPLs

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

```
o¹.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

```
o¹.foo(…)²;
```

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

$o^{1}.\mathtt{foo}(\ldots)^{2};$

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

```
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

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

```
o¹.foo(…)²;
```

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

```
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***

```
o¹.foo(…)²;
```

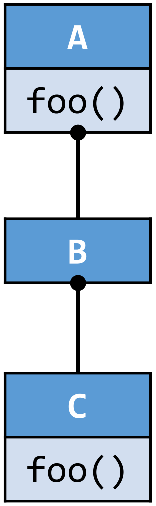
We define function Dispatch(c, m) to simulate the procedure of run-time method dispatch

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

where c' is superclass of c

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

Dispatch: An Example



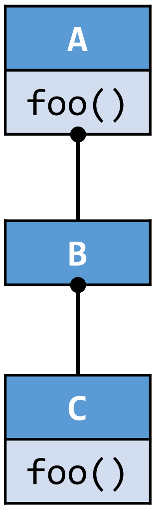

```
class A {
  void foo() {…}
}

class B extends A {
}

class C extends B {
  void foo() {…}
}

void dispatch() {
  A x = new B();
  x.foo();?

  A y = new C();
  y.foo();
}
```

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

where c' is superclass of c

$$\text{Dispatch}(\texttt{B}, \texttt{A.foo()}) = ?$$

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();?
}
```

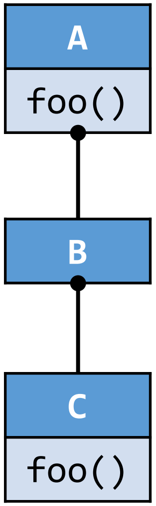
$$\text{Dispatch}(c, m) = \begin{cases} m', & \textbf{if } c \text{ contains non-abstract method } m' \text{ that has the same \underline{name} and \underline{descriptor} as } m \\ \text{Dispatch}(c', m), & \textbf{otherwise} \end{cases}$$

where c' is superclass of c

$\text{Dispatch}(\texttt{B}, \texttt{A.foo()}) = \texttt{A.foo()}$

$\text{Dispatch}(\texttt{C}, \texttt{A.foo()}) = ?$

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();
}
```

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

where c' is superclass of c

$\text{Dispatch}(\texttt{B}, \texttt{A.foo()}) = \texttt{A.foo()}$

$\text{Dispatch}(\texttt{C}, \texttt{A.foo()}) = \texttt{C.foo()}$

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

```
A a = …
a.foo();
```

* Jeffrey Dean, David Grove, Craig Chambers, "*Optimization of Object-Oriented Programs Using Static Class Hierarchy Analysis*". ECOOP 1995.

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

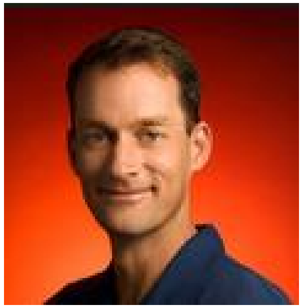
* Jeffrey Dean, David Grove, Craig Chambers, "*Optimization of Object-Oriented Programs Using Static Class Hierarchy Analysis*". ECOOP 1995.

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



* **Jeffrey Dean**, David Grove, Craig Chambers, "*Optimization of Object-Oriented Programs Using Static Class Hierarchy Analysis*". ECOOP 1995.

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 = 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
        $T = \{$ 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 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 = method signature at cs
$\boxed{\begin{array}{l}\textbf{if } cs \text{ is a } \underline{\text{static call}} \textbf{ then} \\ \quad T = \{ m \}\end{array}}$
if cs is <u>special call</u> **then**
c^m = class type of m
$T = \{$ Dispatch(c^m, m) $\}$
if cs is a <u>virtual call</u> **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

```
class C {
  static T foo(P p, Q q)
  {…}
}

C.foo(x, y);
```

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

Call Resolution of CHA

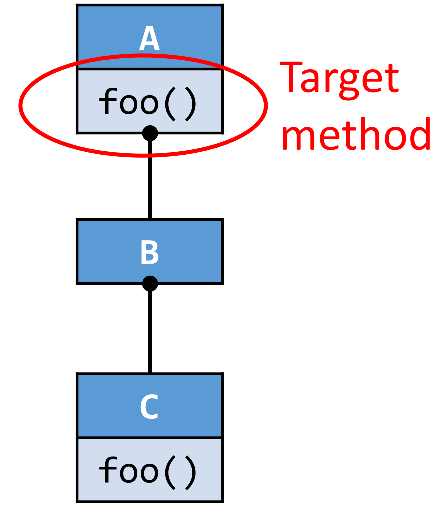
We define function Resolve(cs) to resolve possible target methods of a call site cs by CHA

Resolve(cs)
    $T = \{\}$
    m = 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 Dispatch(c', m) to T
    return T

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

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

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
    $T = \{$ 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 Dispatch(c', m) to T
  return T

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

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

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 Dispatch(c', m) to T
  return T

```
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

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 = 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
    $T = \{$ 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 Dispatch(c', m) to T
  return T

```
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`

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 = 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
    $T = \{$ 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 Dispatch(c', m) to T
  return T

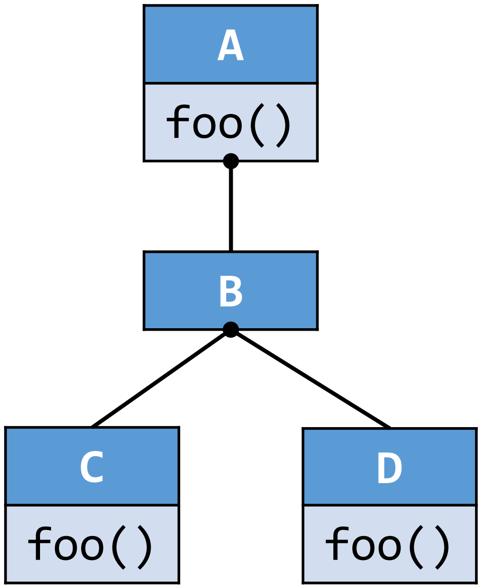
```
class A {
  T foo(P p, Q q) {…}
}
A a = …
a.foo(x, y);
```

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

Subclasses includes all **direct** and **indirect** subclasses of c

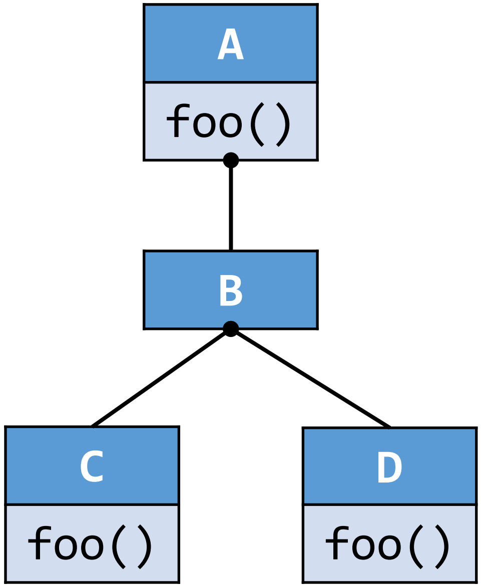
CHA: An Example

```
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();

  A a = …
  a.foo();

  B b = …
  b.foo();
}
```



CHA: An Example

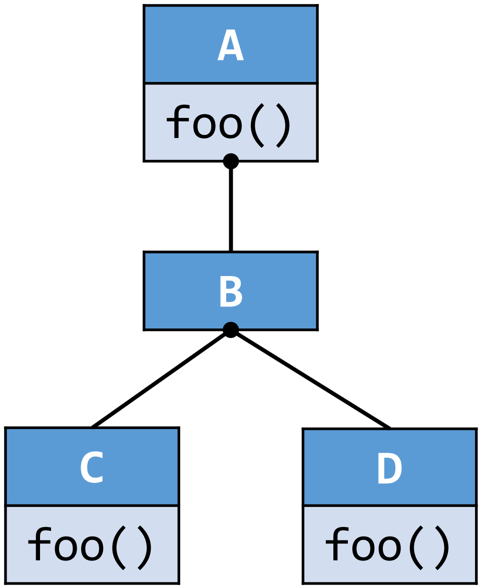
```
class A {
  void foo() {…}
}
class B extends A {}

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

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

  A a = …
  a.foo();

  B b = …
  b.foo();
}
```



Resolve(`c.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();

  A a = …
  a.foo();

  B b = …
  b.foo();
}
```

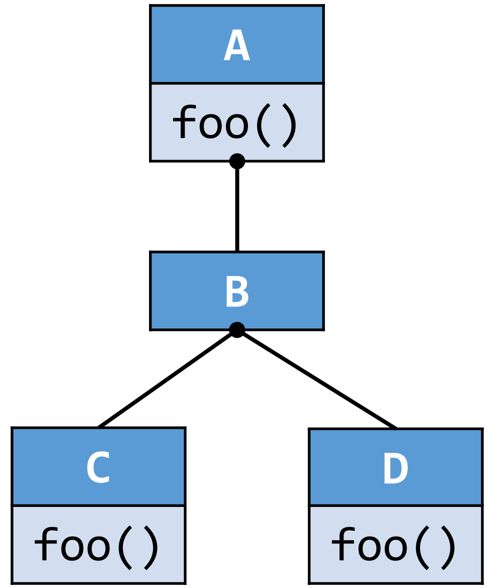


Resolve(`c.foo()`) = {`C.foo()`}

Resolve(`a.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()) = {C.foo()}

```
  A a = …
  a.foo();
```

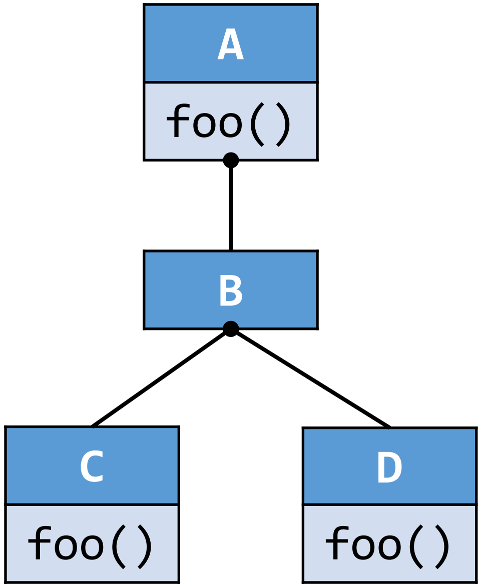
Resolve(a.foo()) = {A.foo(), C.foo(), D.foo()}

```
  B b = …
  b.foo(); ?
}
```

Resolve(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();

  A a = …
  a.foo();

  B b = …
  b.foo();
}
```

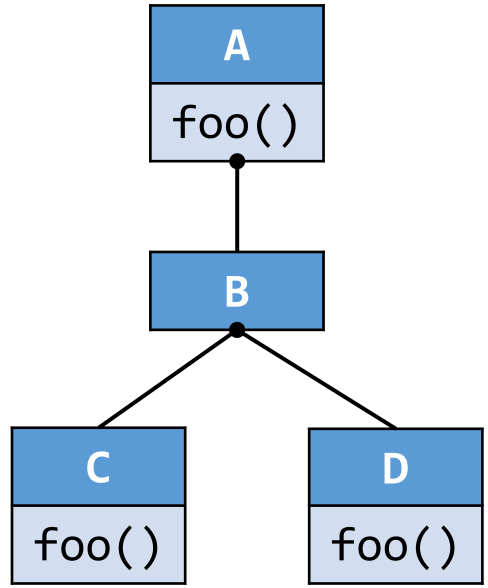
Resolve(c.foo()) = {C.foo()}

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

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

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();

  A a = …
  a.foo();

  B b = new B();
  b.foo(); ?
}
```

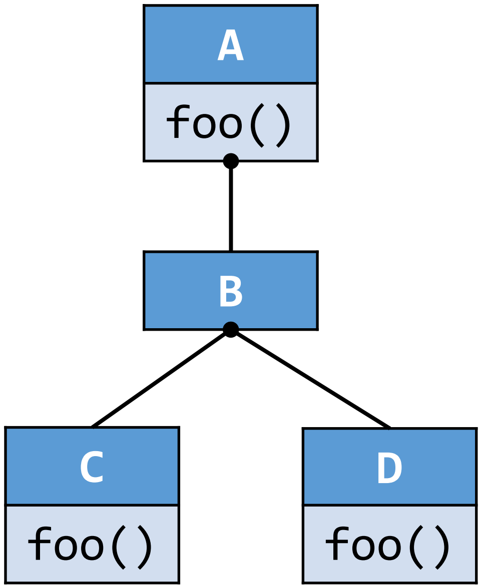
Resolve(`c.foo()`) = {`C.foo()`}

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

Resolve(`b.foo()`) = ?

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();

  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)

TestCHA.java

```java
public class TestCHA {
    void test() {
        B b = new B();
        b.foo();
    }
}
class A {
    void foo() {}
}
class B extends A {}
class C extends B {
    void foo() {}
}
class D extends B {
    void foo() {}
}
```

Hierarchy: Callees of foo

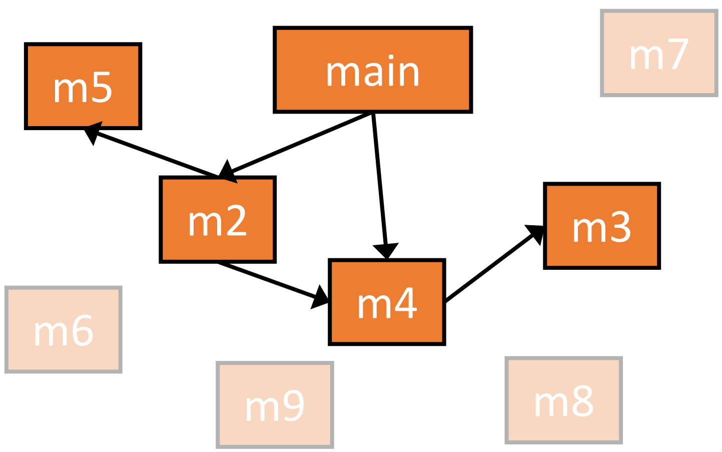
Scope: All

- A.foo() ()
  - C.foo() ()
  - D.foo() ()

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 ∉ 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 → 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

$\text{BuildCallGraph}(m^{entry})$
  $WL = [m^{entry}]$, $CG = \{\}$, $RM = \{\}$
  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**
        $T = \text{Resolve}(cs)$
        foreach target method m' in T **do**
          add $cs \rightarrow m'$ to CG
          add m' to WL
  return CG

Initialize the algorithm

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

$\textcolor{red}{\text{BuildCallGraph}}(m^{entry})$
    $WL = [m^{entry}]$, $CG = \{\}$, $RM = \{\}$ — 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**
                $T = \textcolor{red}{\text{Resolve}}(cs)$ — Resolve target methods via CHA
                foreach target method m' in T **do**
                    add $cs \to m'$ 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

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

Initialize the algorithm

Resolve target methods via CHA

Add call edges to call graph

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

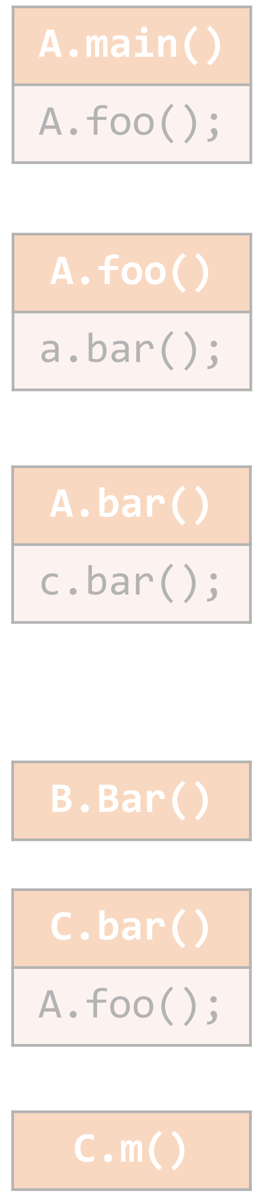
Call Graph Construction: Algorithm

BuildCallGraph(m^{entry})
    $WL = [m^{entry}]$, $CG = \{\}$, $RM = \{\}$ — 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 = Resolve(cs) — Resolve target methods via CHA
                foreach target method m' in T **do**
                    add $cs \rightarrow m'$ to CG — Add call edges to call graph
                    add m' to WL — May discover new method, add it to work list
    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: 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() {}
}
```



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() {}
}
```

A.main()
A.foo();

A.foo()
a.bar();

A.bar()
c.bar();

B.Bar()

C.bar()
A.foo();

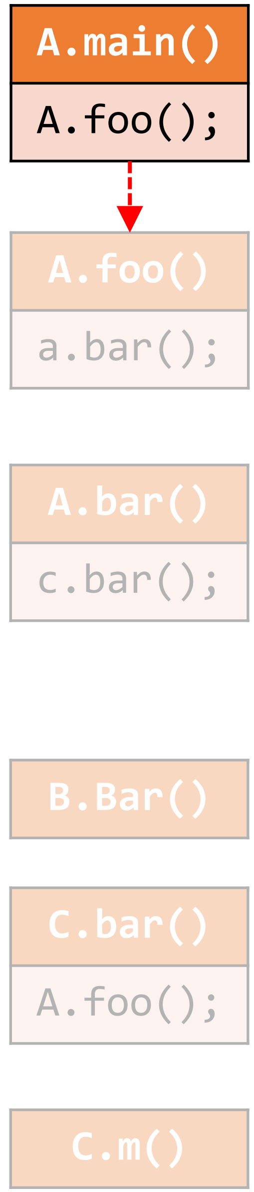
C.m()

$WL = [\ \ ]$

Resolve(A.foo()) = ?

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() {}
}
```

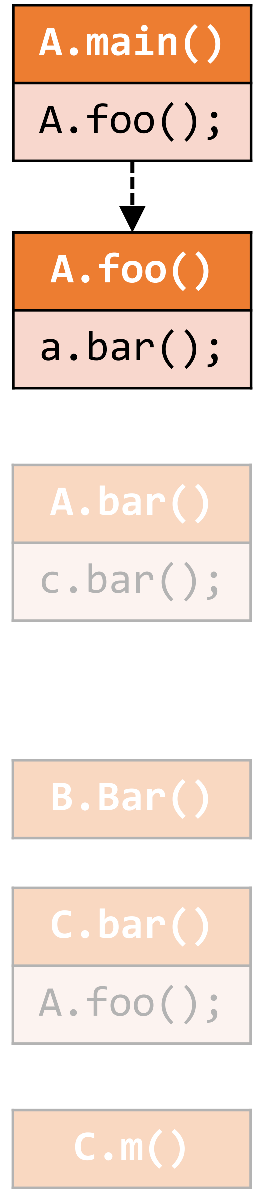


$WL = [$`A.foo()`$]$

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

Call Graph Construction: An Example

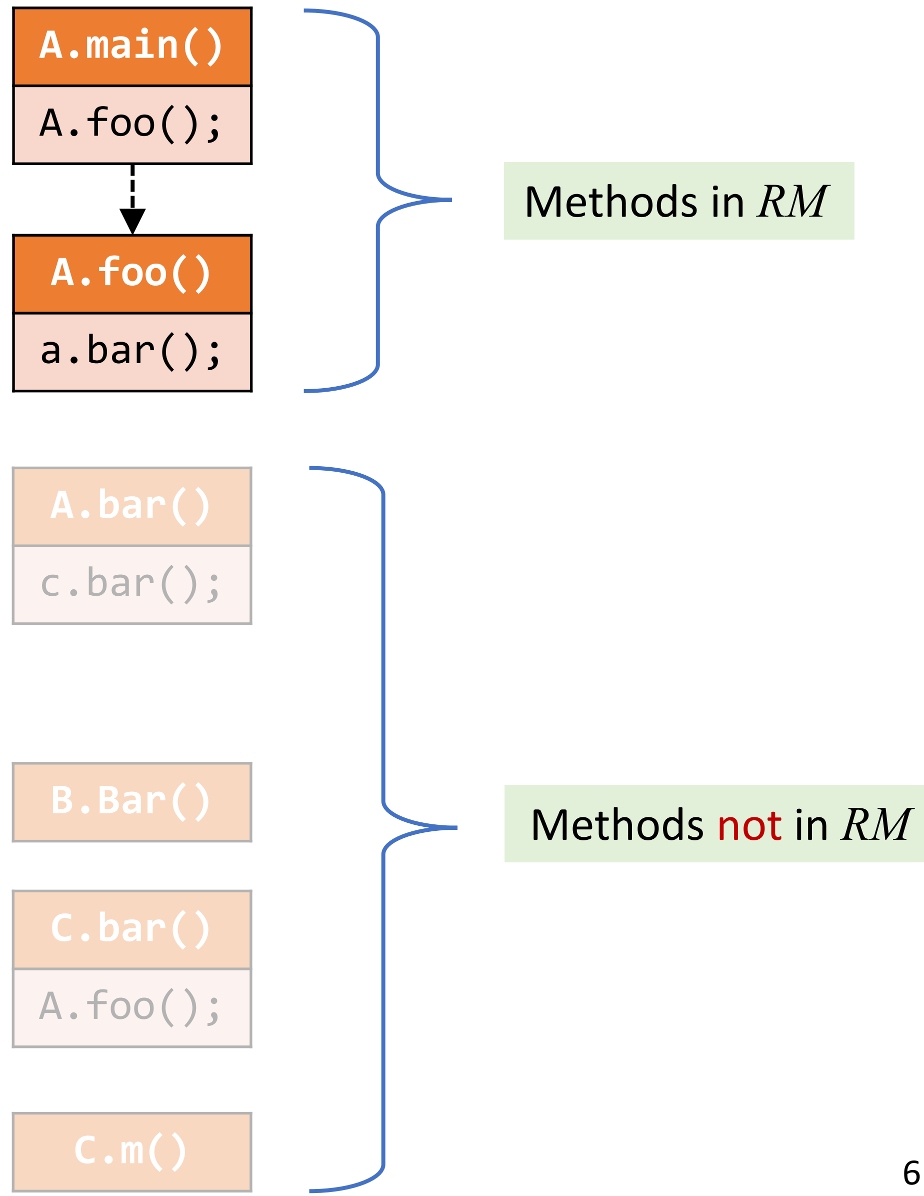
```
class A {
  static void main() {
    A.foo();
  }
  static void foo() {
    A a = new A();
    a.bar();
  }
  void bar() {
    C c = new C();
    c.bar();
  }
}
class B extends A {
  void bar() {} }
class C extends A {
  void bar() {
    if (…) A.foo();
  }
  void m() {}
}
```



$WL = [\ ]$

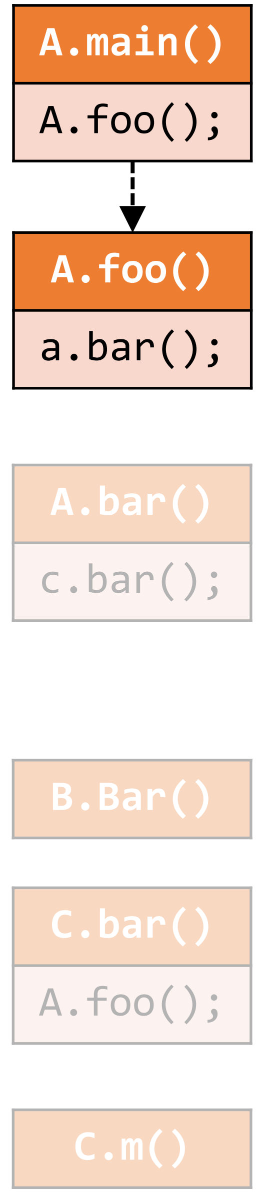
Call Graph Construction: An Example

```
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() {}
}
```

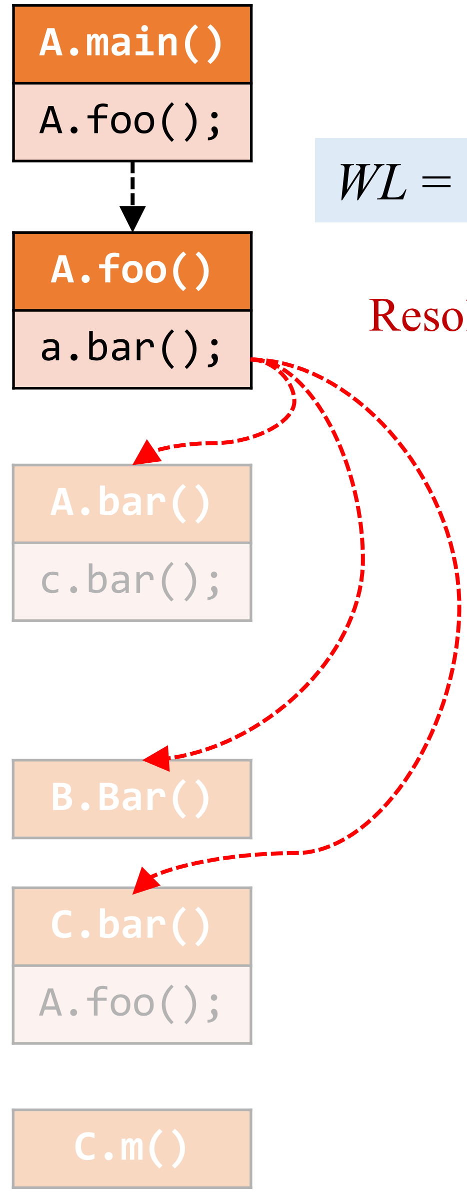


$WL = [\ \ ]$

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() {}
}
```

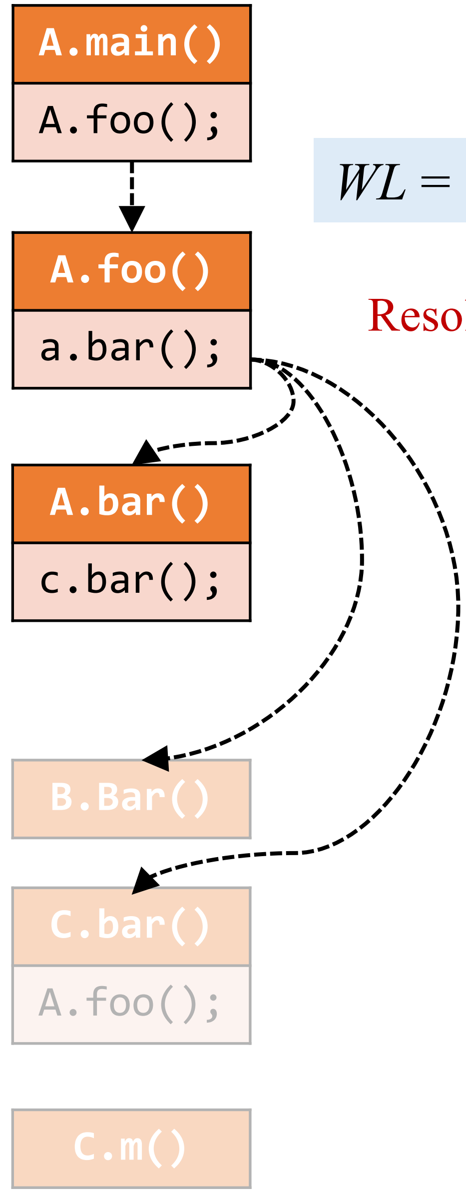


$WL = [$`A.bar(), B.bar(), C.bar()`$]$

Resolve(`a.bar()`) = { `A.bar()`, `B.bar()`, `C.bar()` }

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() {}
}
```

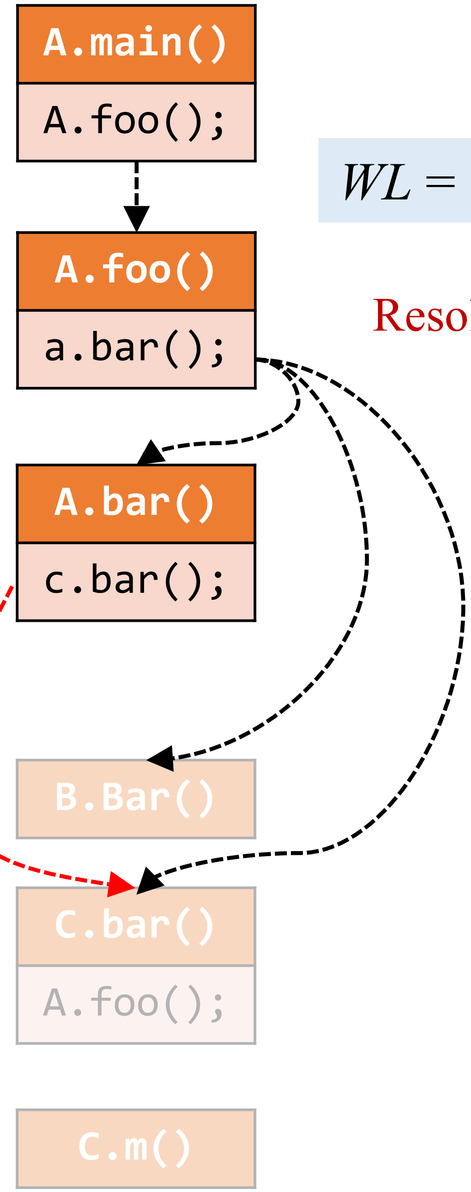


$WL = [$`B.bar(), C.bar()`$]$

Resolve(`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() {}
}
```

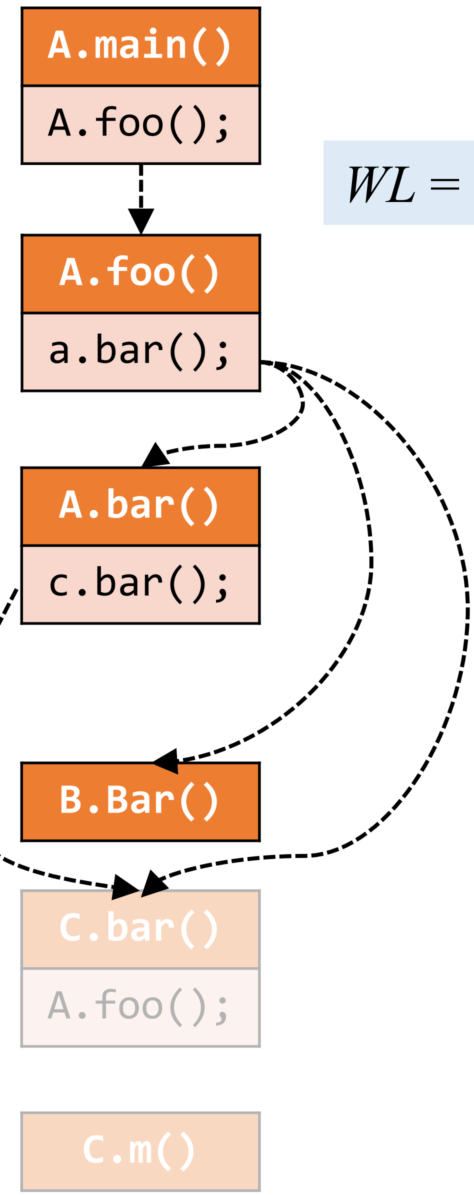


$WL = [$`B.bar()`, `C.bar()`, `C.bar()`$]$

Resolve(`c.bar()`) = { `C.bar()` }

Call Graph Construction: An Example

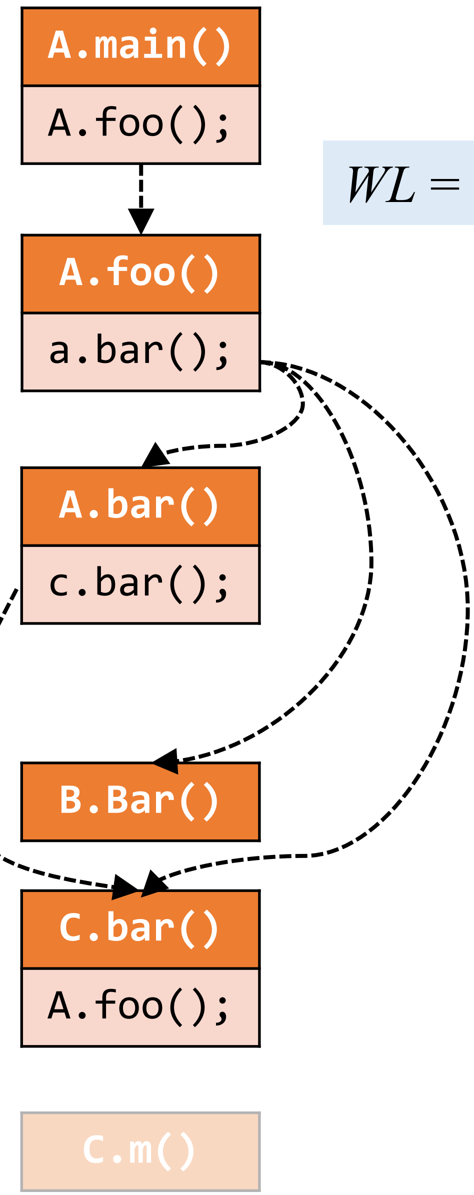
```
class A {
  static void main() {
    A.foo();
  }
  static void foo() {
    A a = new A();
    a.bar();
  }
  void bar() {
    C c = new C();
    c.bar();
  }
}
class B extends A {
  void bar() {} }
class C extends A {
  void bar() {
    if (…) A.foo();
  }
  void m() {}
}
```



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

Call Graph Construction: An Example

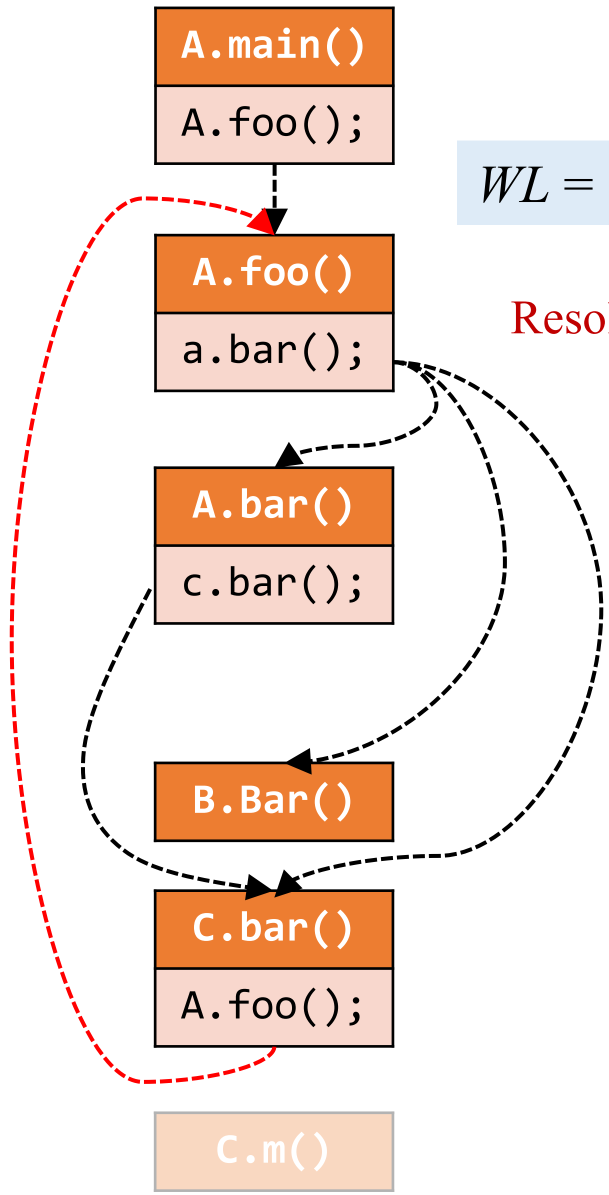
```
class A {
  static void main() {
    A.foo();
  }
  static void foo() {
    A a = new A();
    a.bar();
  }
  void bar() {
    C c = new C();
    c.bar();
  }
}
class B extends A {
  void bar() {} }
class C extends A {
  void bar() {
    if (…) A.foo();
  }
  void m() {}
}
```



$WL = [$`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() {}
}
```

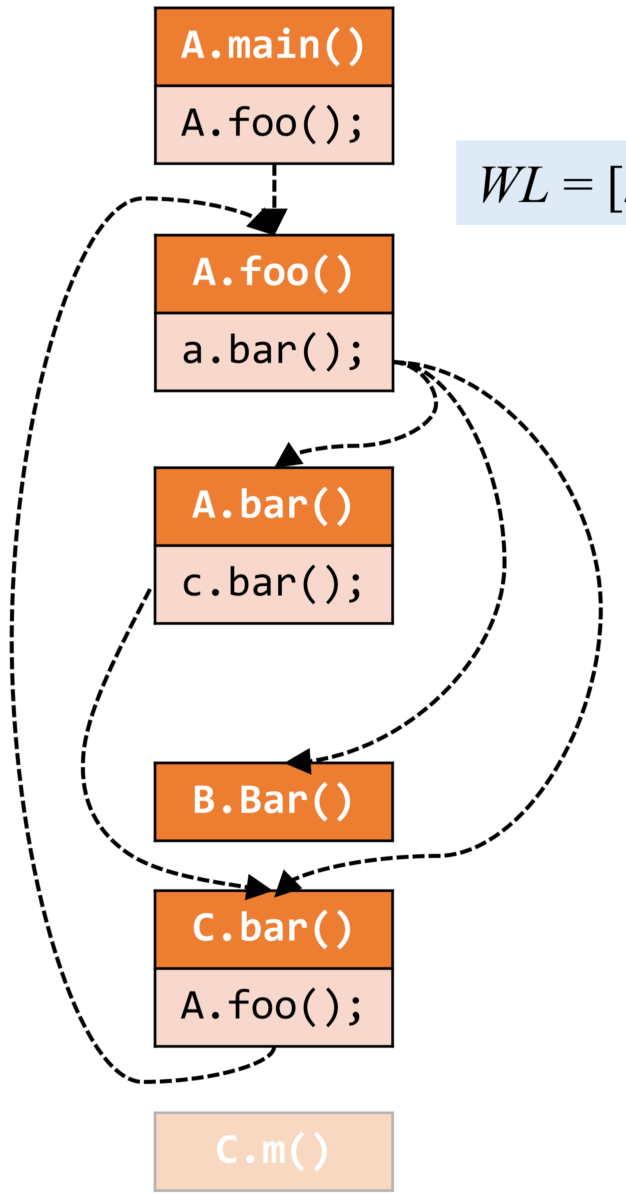


$WL = [$`C.bar(),A.foo()`$]$

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

Call Graph Construction: An Example

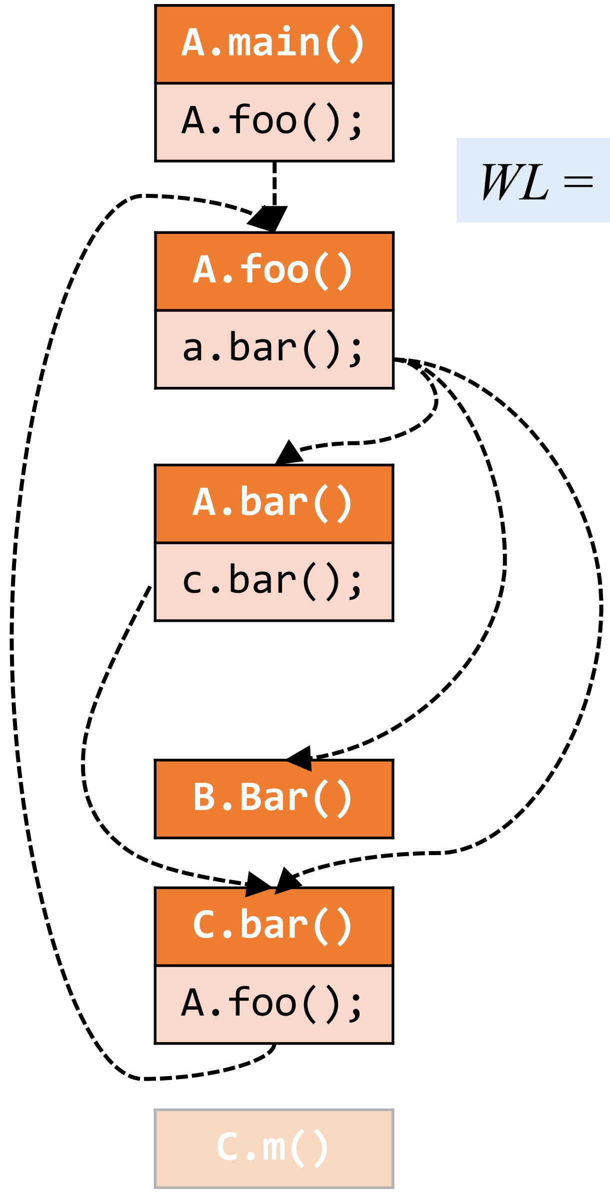
```
class A {
  static void main() {
    A.foo();
  }
  static void foo() {
    A a = new A();
    a.bar();
  }
  void bar() {
    C c = new C();
    c.bar();
  }
}
class B extends A {
  void bar() {} }
class C extends A {
  void bar() {
    if (…) A.foo();
  }
  void m() {}
}
```



$WL = [$A.foo()$]$

Call Graph Construction: An Example

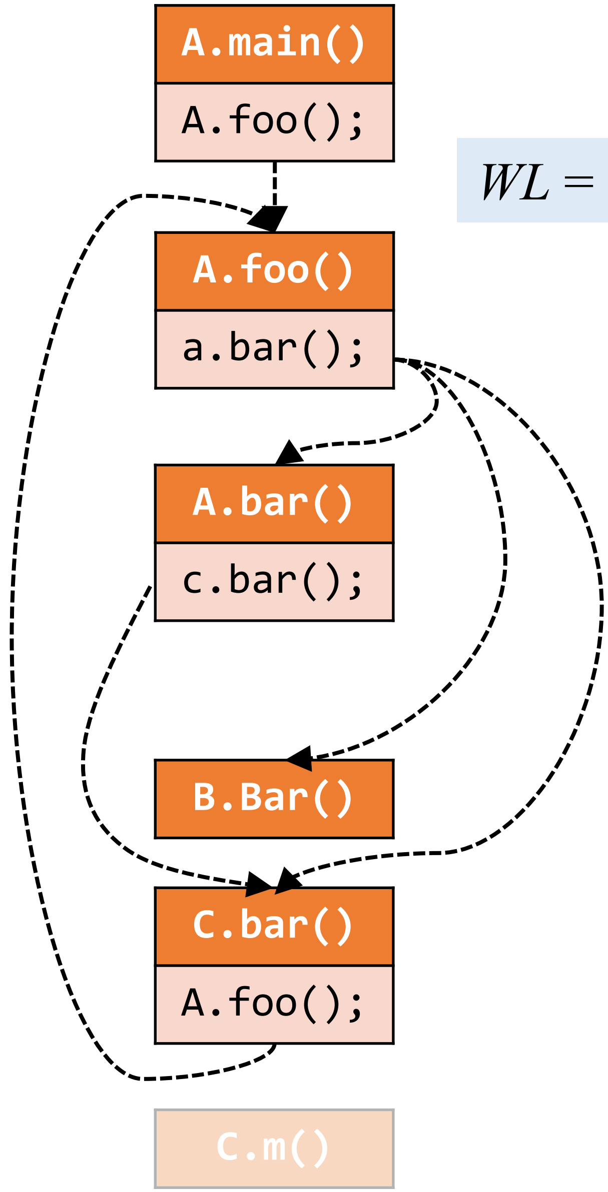
```
class A {
  static void main() {
    A.foo();
  }
  static void foo() {
    A a = new A();
    a.bar();
  }
  void bar() {
    C c = new C();
    c.bar();
  }
}
class B extends A {
  void bar() {} }
class C extends A {
  void bar() {
    if (…) A.foo();
  }
  void m() {}
}
```



$WL = [\ ]$

Call Graph Construction: An Example

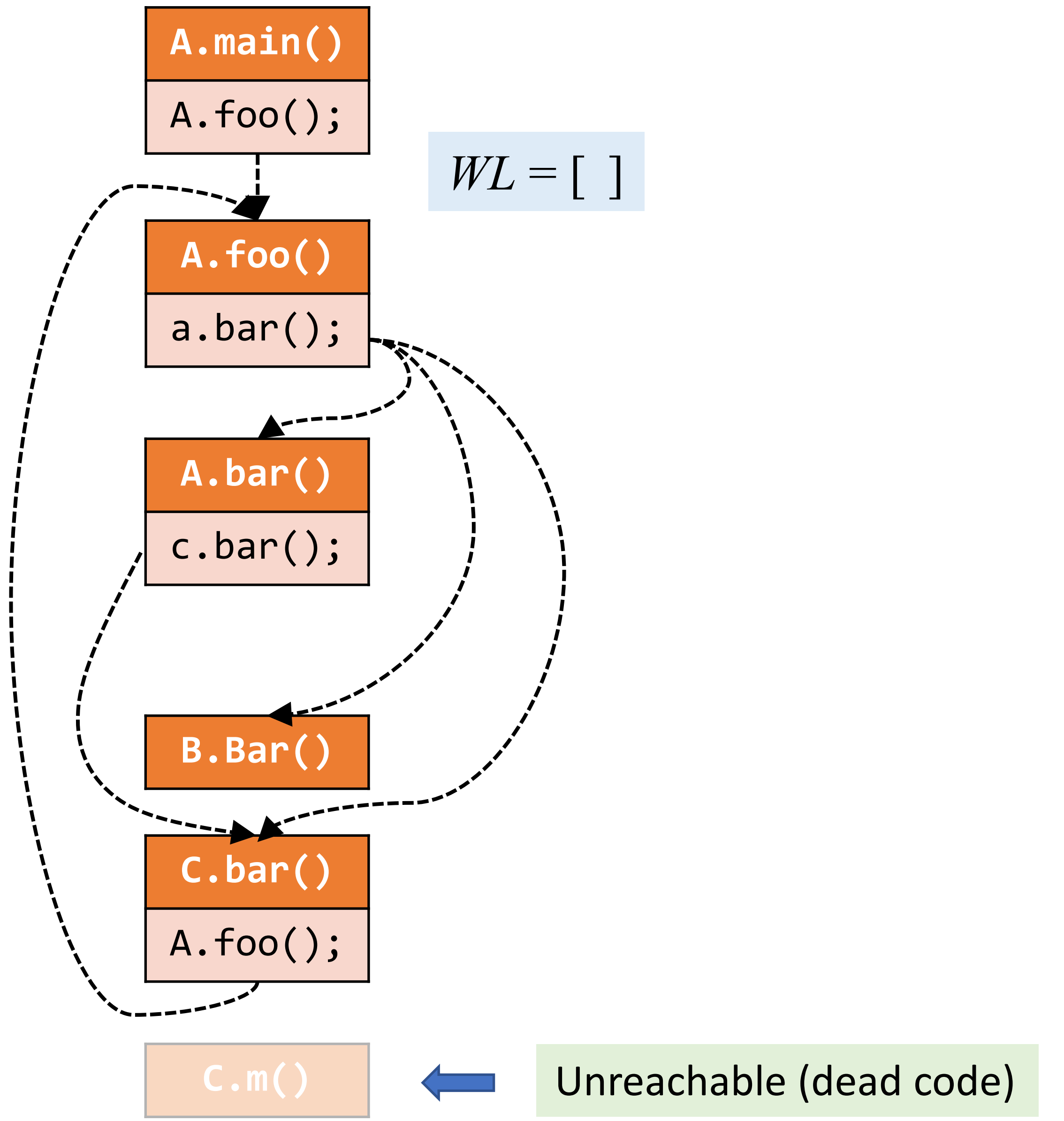
```
class A {
  static void main() {
    A.foo();
  }
  static void foo() {
    A a = new A();
    a.bar();
  }
  void bar() {
    C c = new C();
    c.bar();
  }
}
class B extends A {
  void bar() {} }
class C extends A {
  void bar() {
    if (…) A.foo();
  }
  void m() {}
}
```



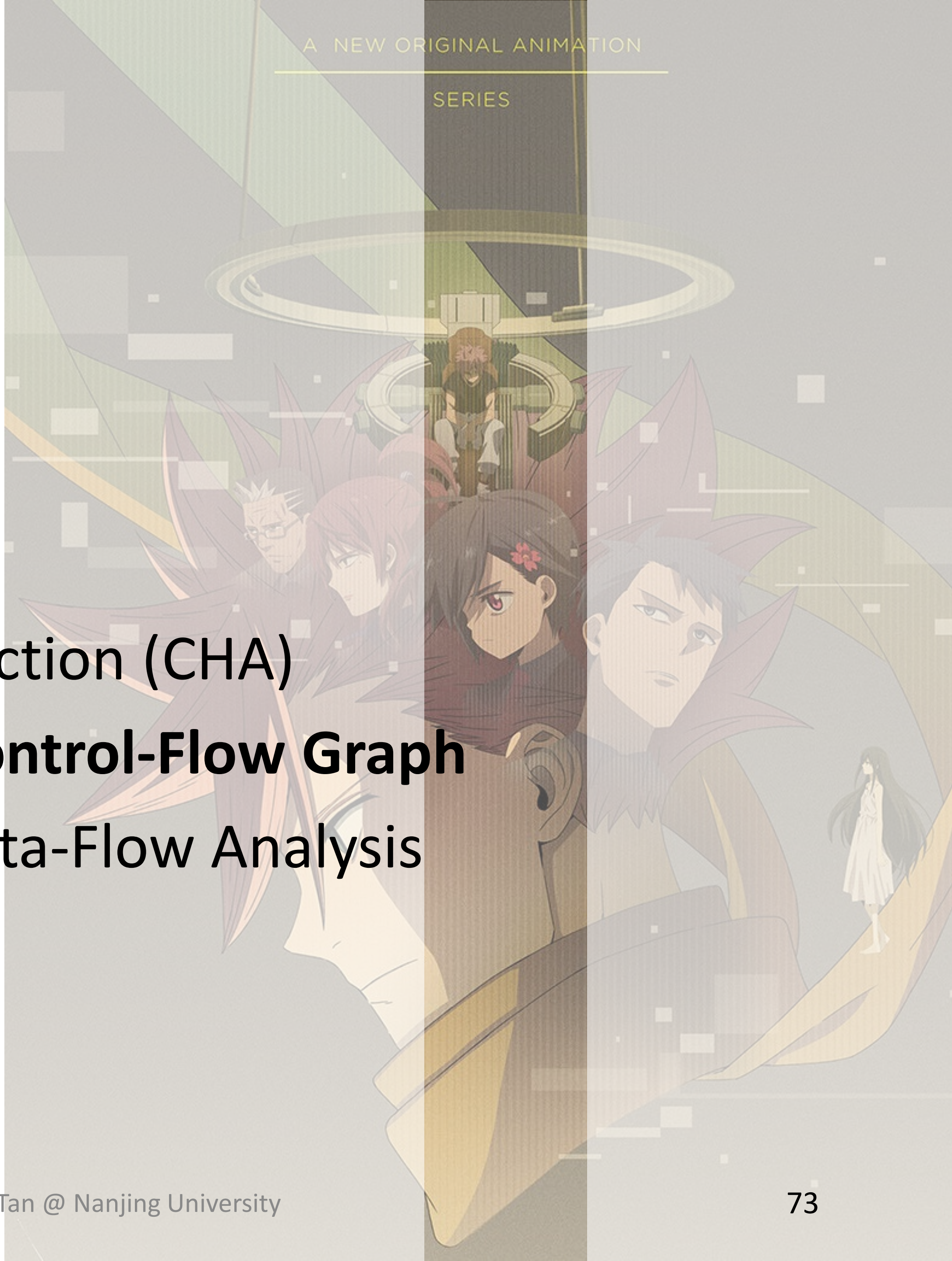
$WL = [\ ]$

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



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 exit nodes 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 exit nodes of the callees to the statements following their call sites (i.e., return sites)

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

Interprocedural Control-Flow Graph

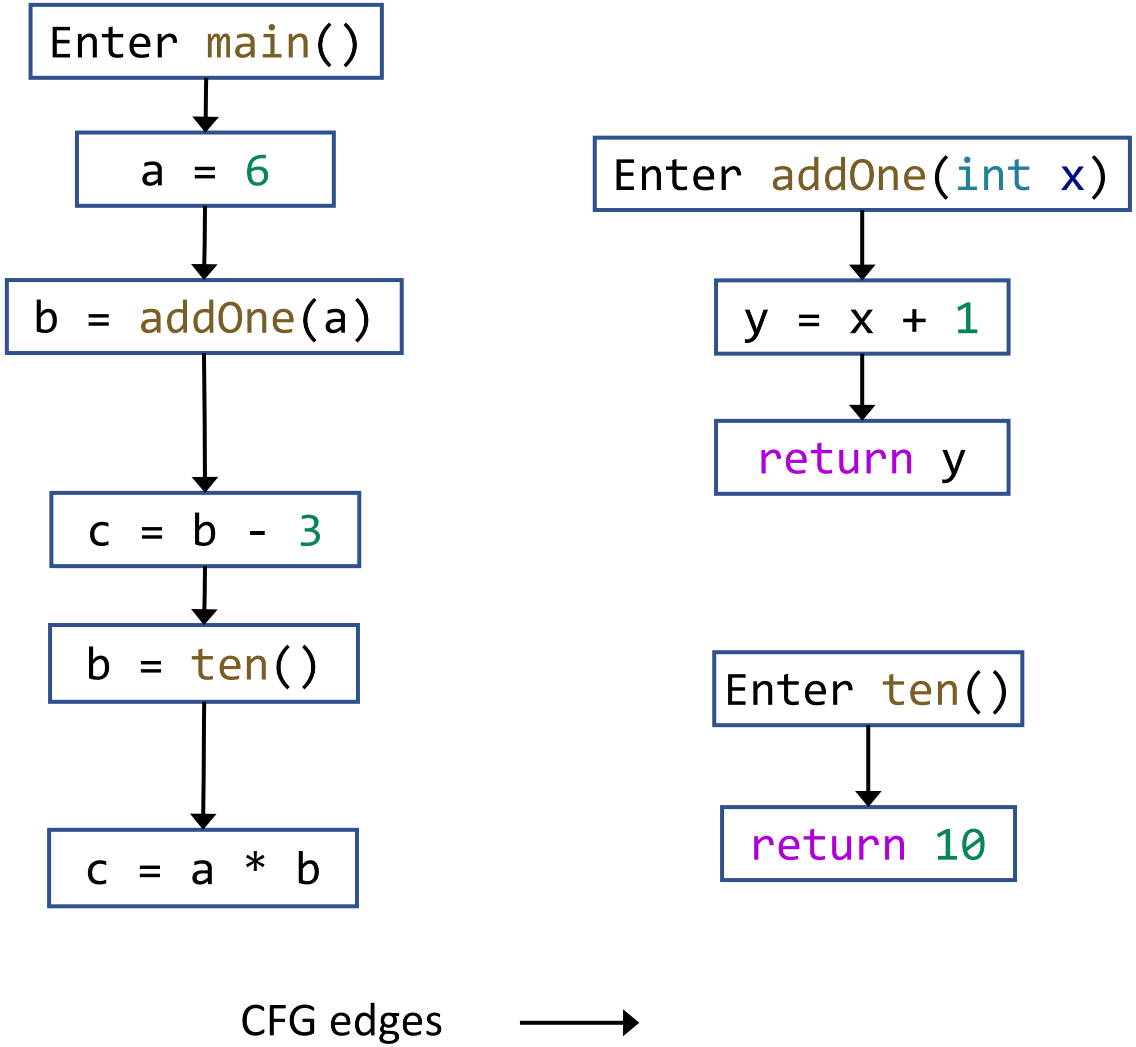
- CFG represents structure of an individual method
- ICFG represents structure of the whole program
  - With ICFG, we can perform interprocedural analysis
- An ICFG of a program consists of CFGs of the methods in the program, plus two kinds of additional edges:
  - ➢ Call edges: from call sites to the entry nodes of their callees
  - ➢ Return edges: from exit nodes 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

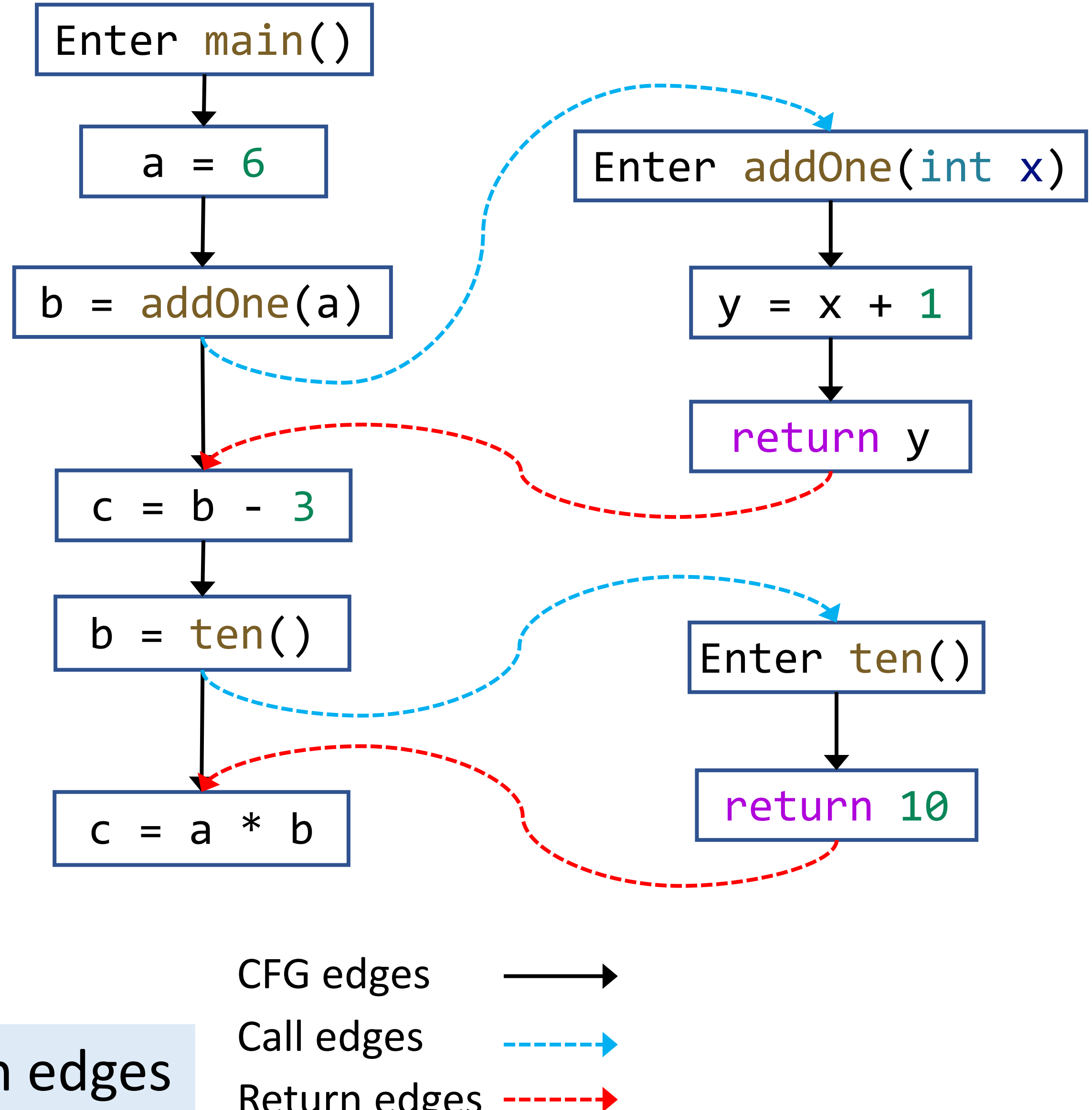
ICFG: An Example

```
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

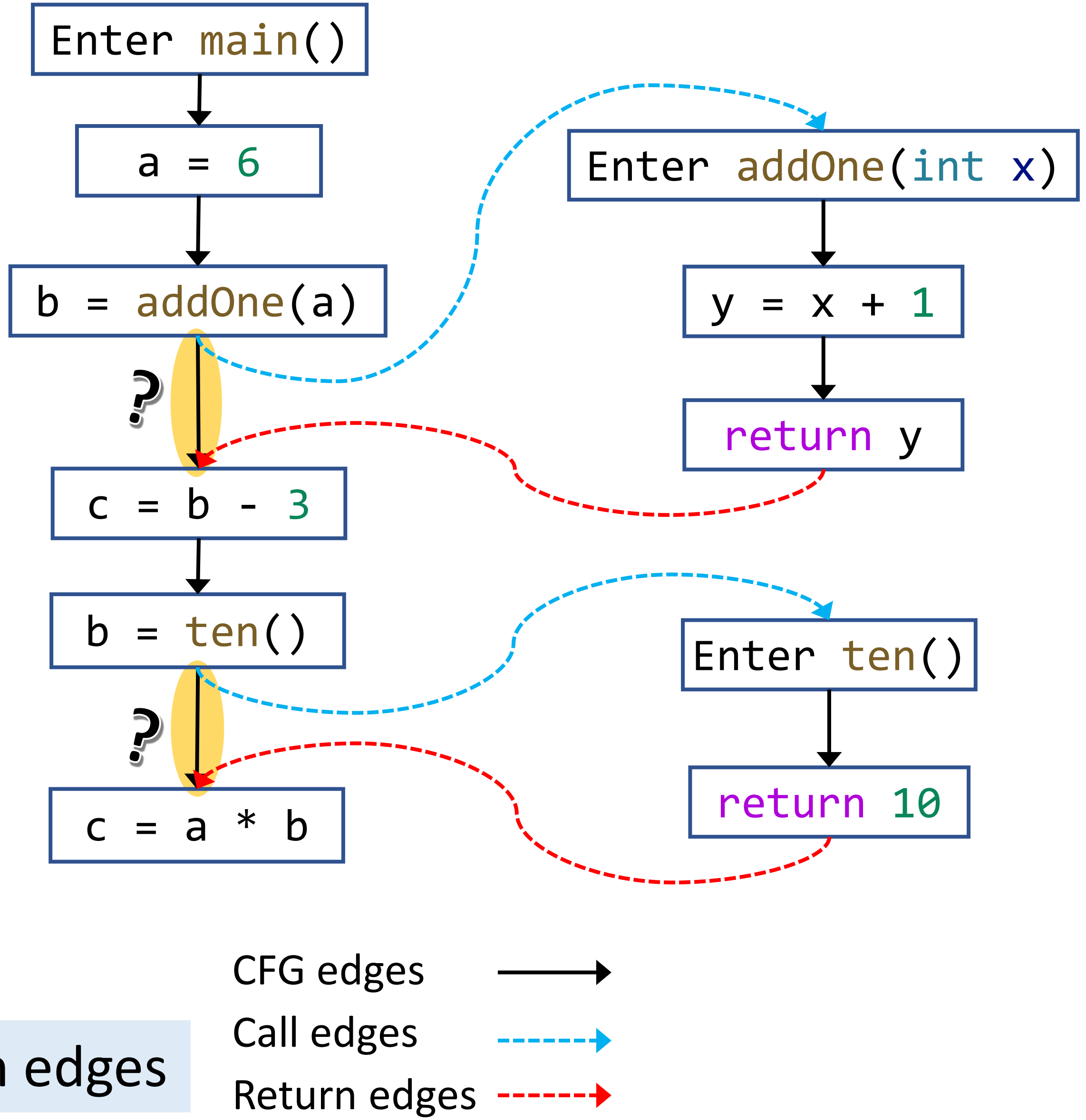
```
static void main() {
  int a, b, c;
  a = 6;
  b = addOne(a);
  c = b - 3;
  b = ten();
  c = a * b;
}

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

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



ICFG = CFGs + call & return edges

ICFG: An Example

```
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;
}
```



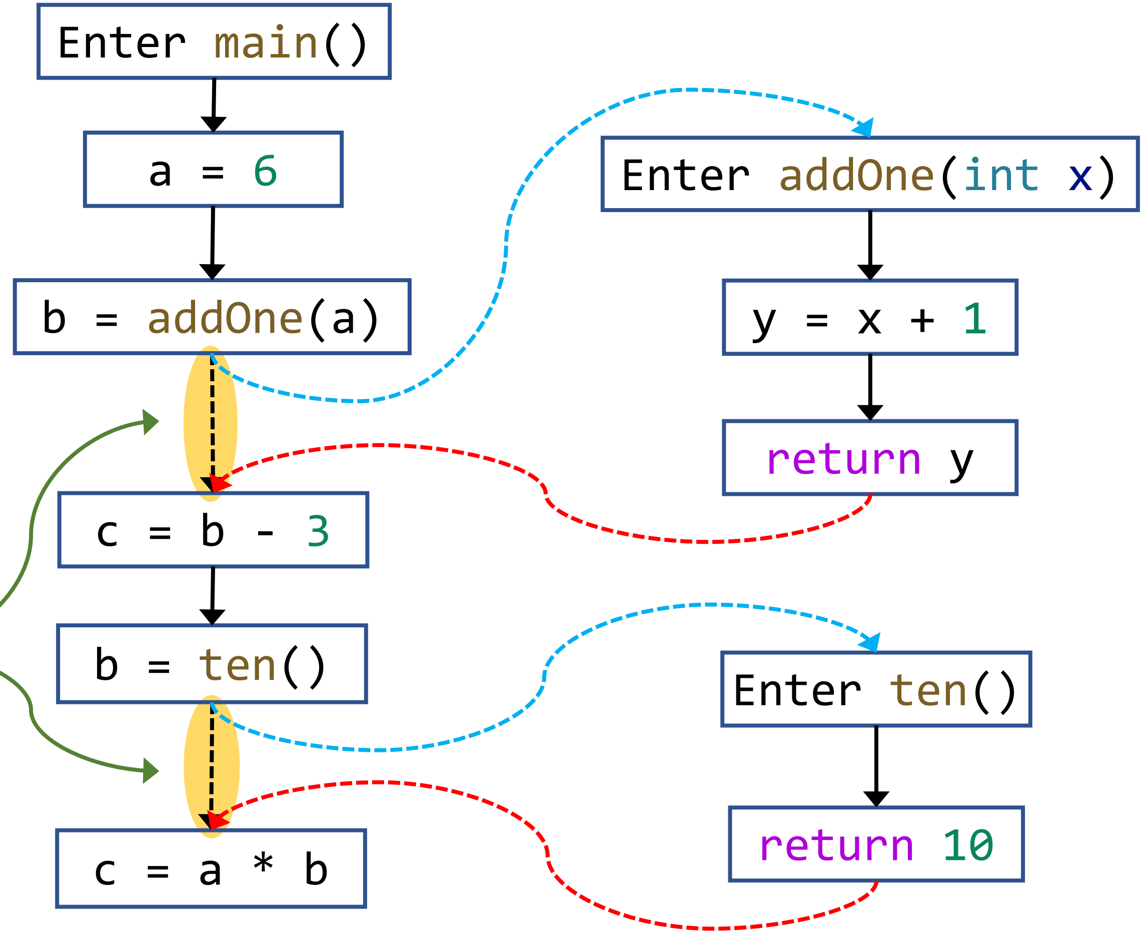
ICFG = CFGs + call & return edges

ICFG: An Example

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

Such edges (from call site to return site) are called *call-to-return edges*

```
}

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



CFG edges
- **Normal edges**
- **Call-to-return edges**

Call edges

Return edges

ICFG = CFGs + call & return edges

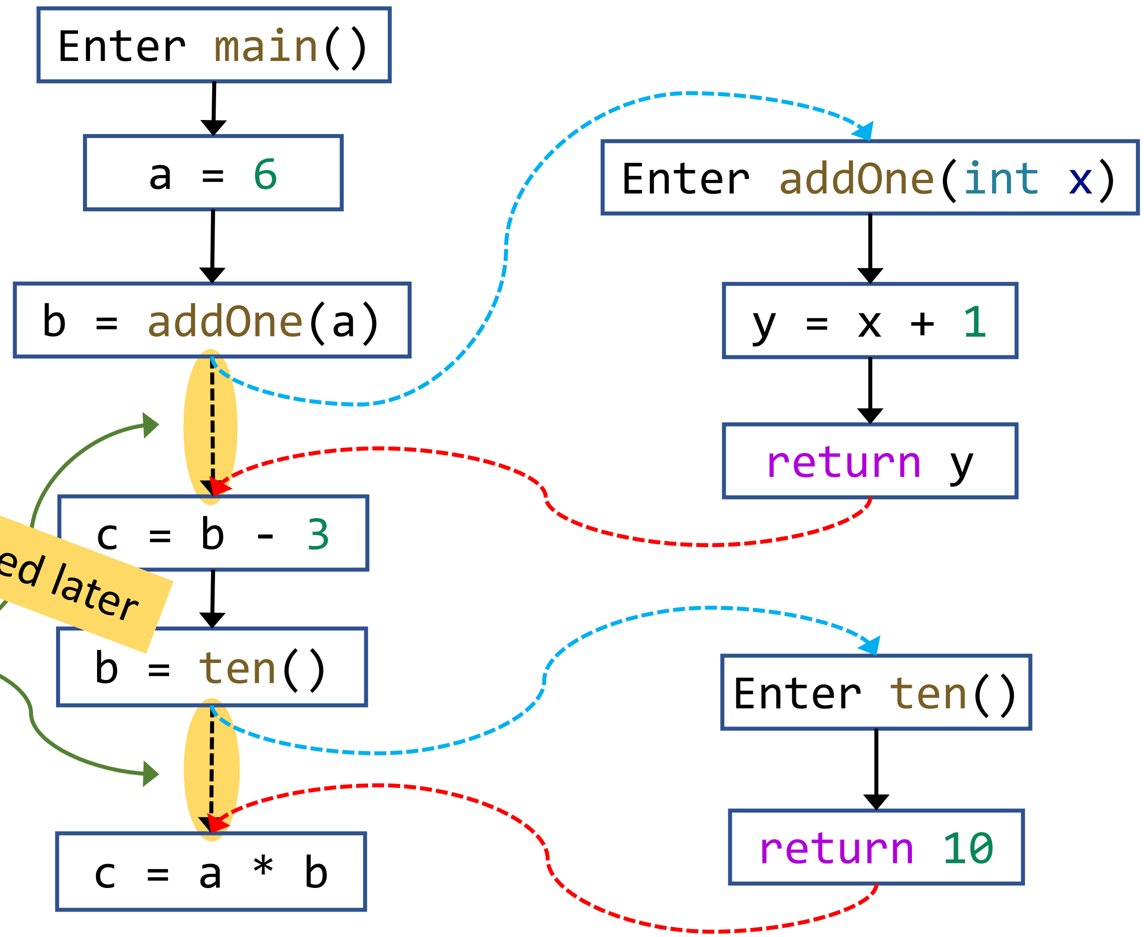
ICFG: An Example

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

Such edges (from call site to return site) are called *call-to-return edges*

```
}

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

Will be explained later



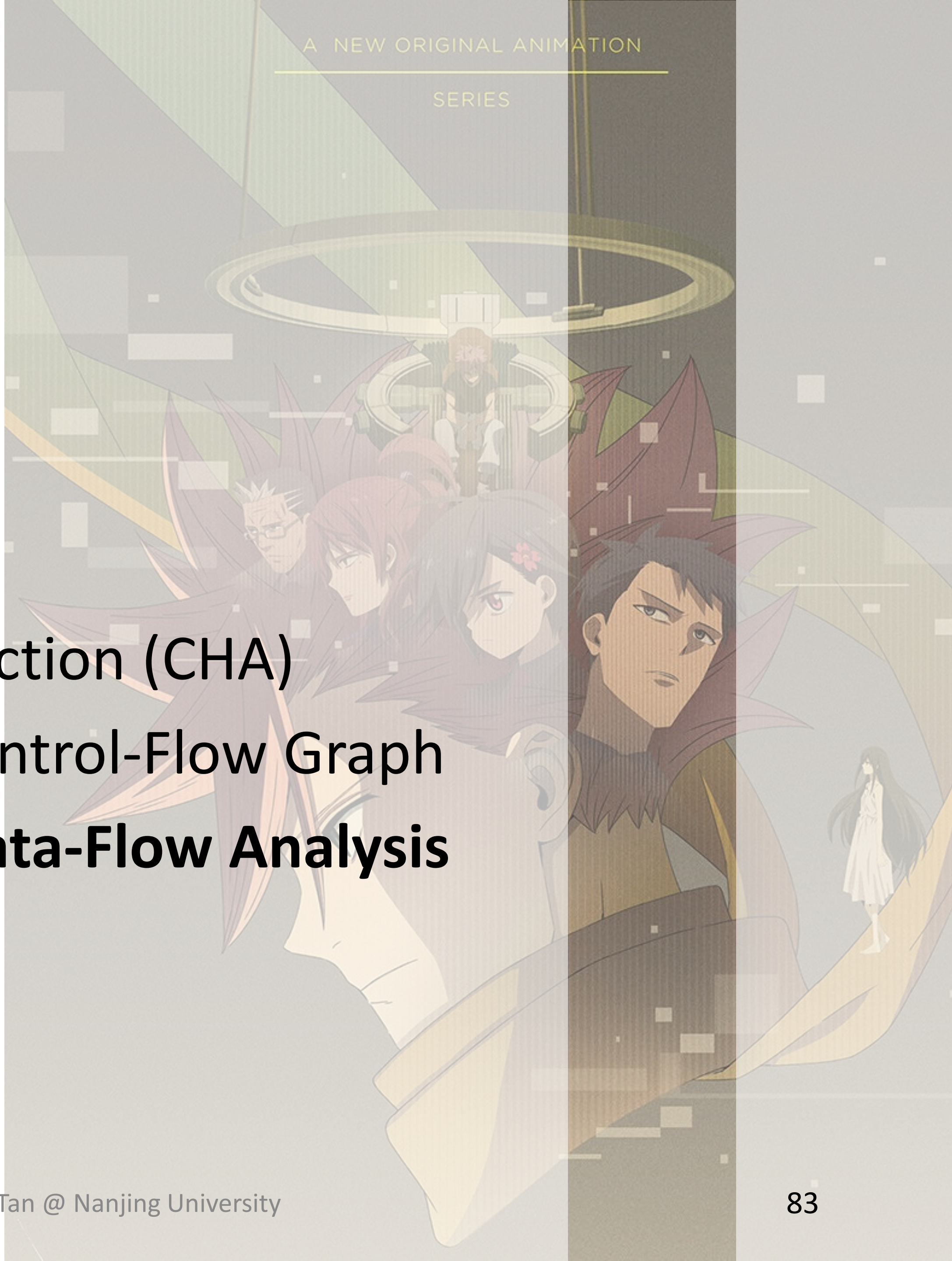
CFG edges
- **Normal edges**
- **Call-to-return edges**

Call edges

Return edges

ICFG = CFGs + call & return edges

Contents



Interprocedural Data-Flow Analysis

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

	Intra**procedural**	***Inter*****procecdural**
Program representation	CFG	ICFG = CFGs + call & return edges

Interprocedural Data-Flow Analysis

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

	Intra**procedural**	***Inter*****procecdural**
Program representation	CFG	ICFG = CFGs + call & return edges
Transfer functions	Node transfer	Node transfer + edge transfer

Interprocedural Data-Flow Analysis

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

	*Intra*procedural	*Inter*procecdural
Program representation	CFG	ICFG = CFGs + call & return edges
Transfer functions	Node transfer	Node transfer + edge transfer

Edge transfer

- Call edge transfer: transfer data flow from call site to the entry node of callee (along call edges)
- Return edge transfer: transfer data flow from exit 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 intraprocedural constant propagation, except that
  - For call nodes, the transfer function is *identity* function

Intraprocedural

a=…, b=…

a=…, b=NAC

`b = addOne(a)`

Interprocedural

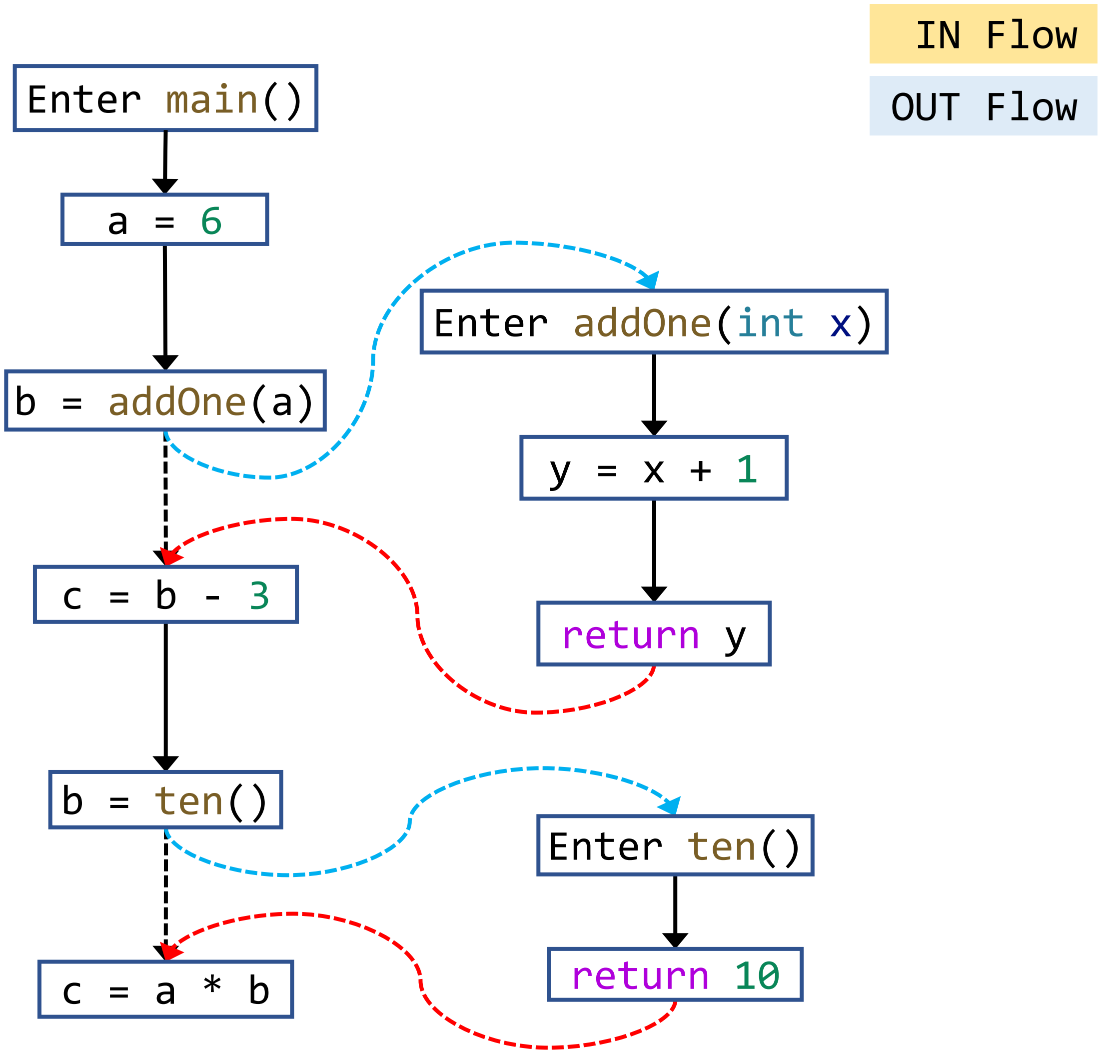
a=…, b=…

a=…, b=…

Leave the handling of the LHS variable (return value) to edge transfer

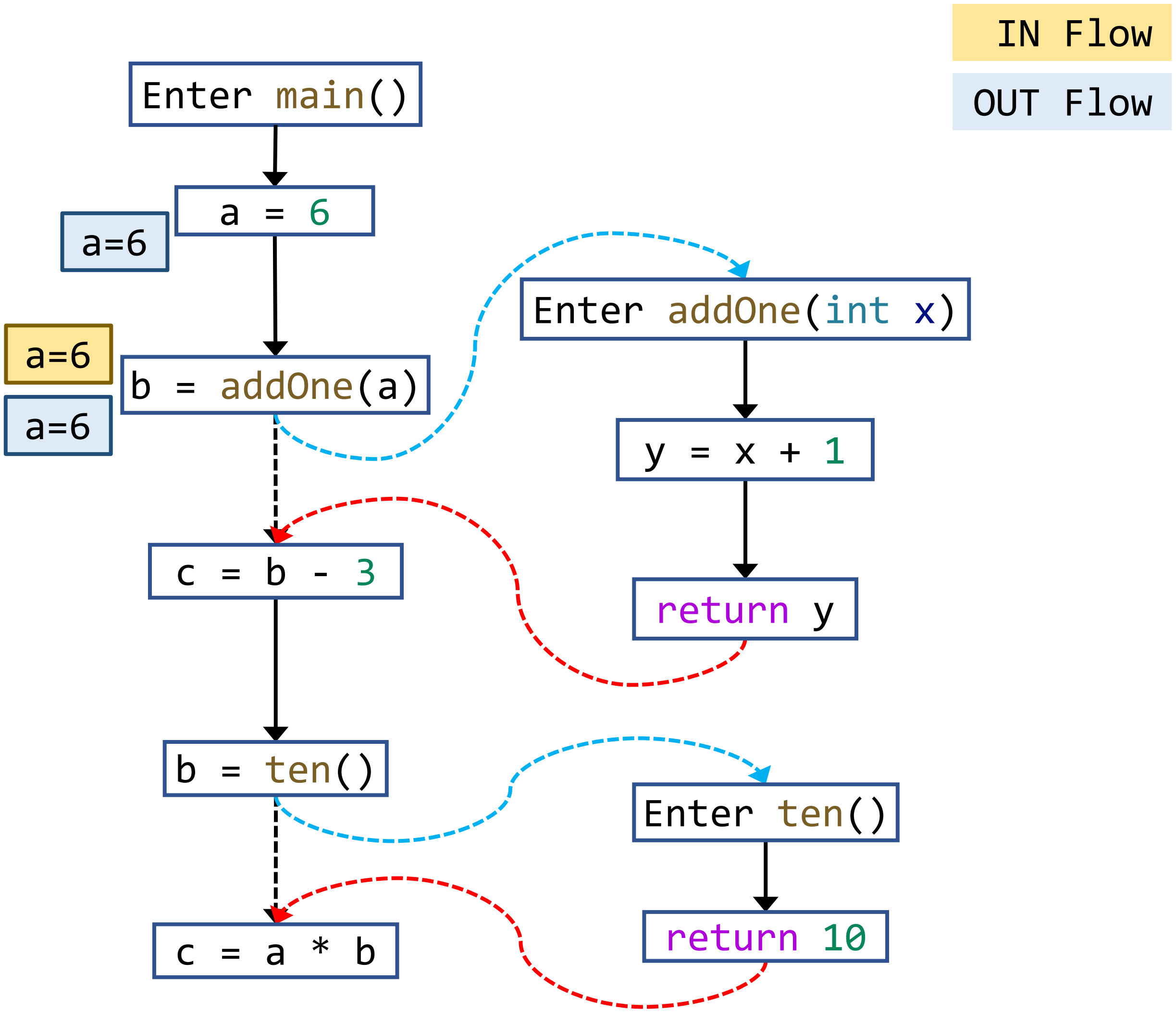
Interprocedural Constant Propagation: An Example

```
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;
}
```



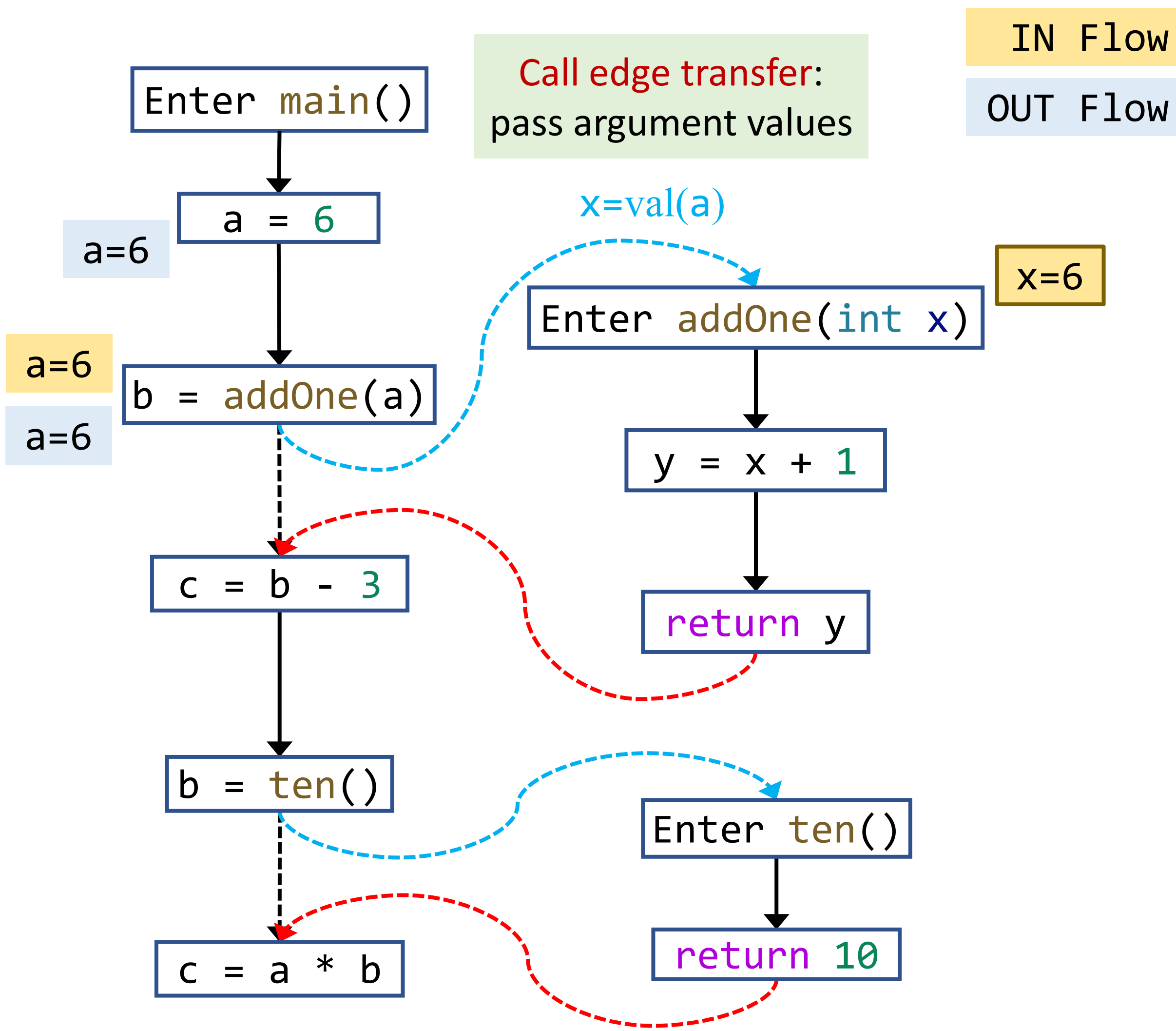
Interprocedural Constant Propagation: An Example

```
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;
}
```



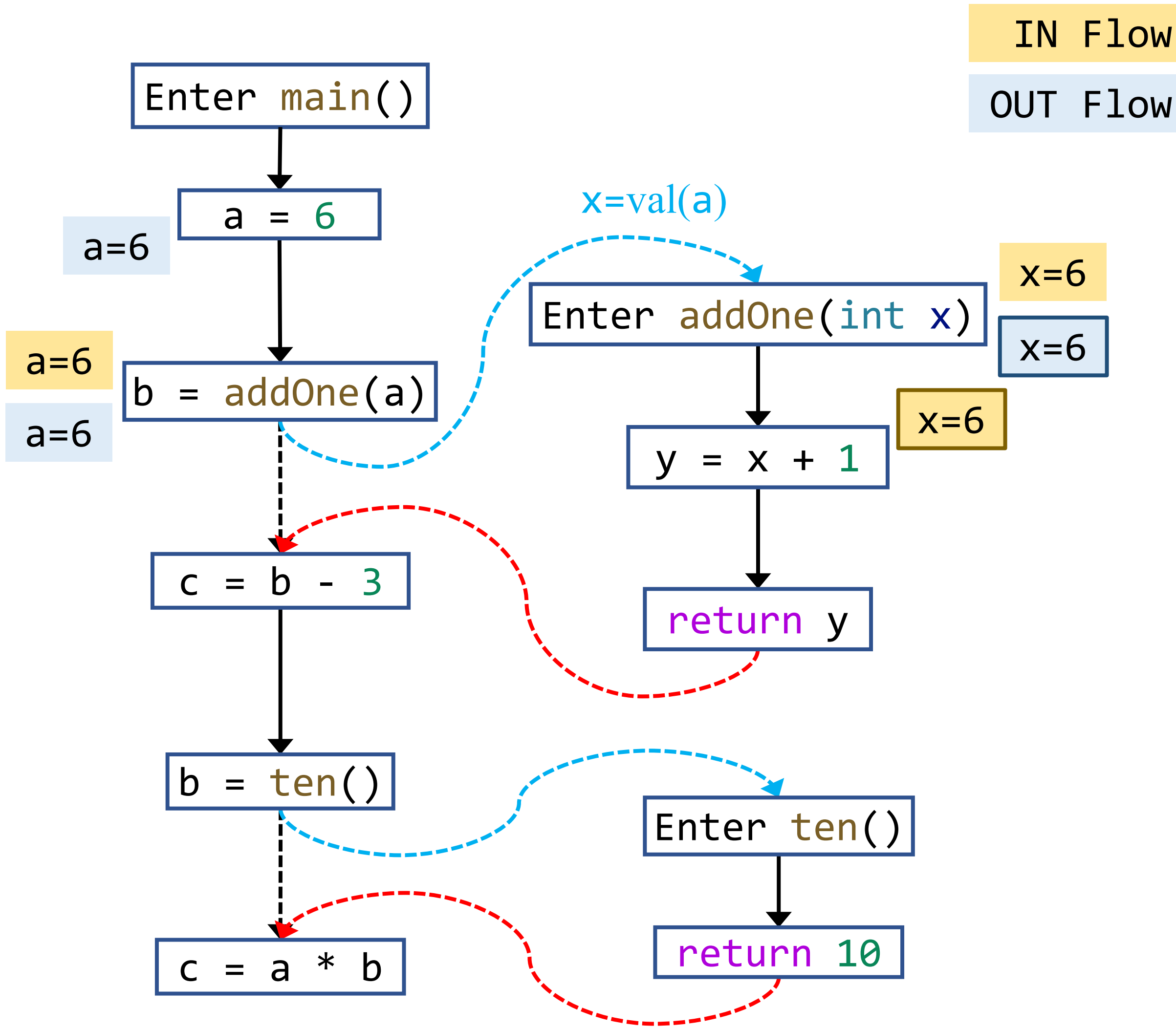
Interprocedural Constant Propagation: An Example

```
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;
}
```



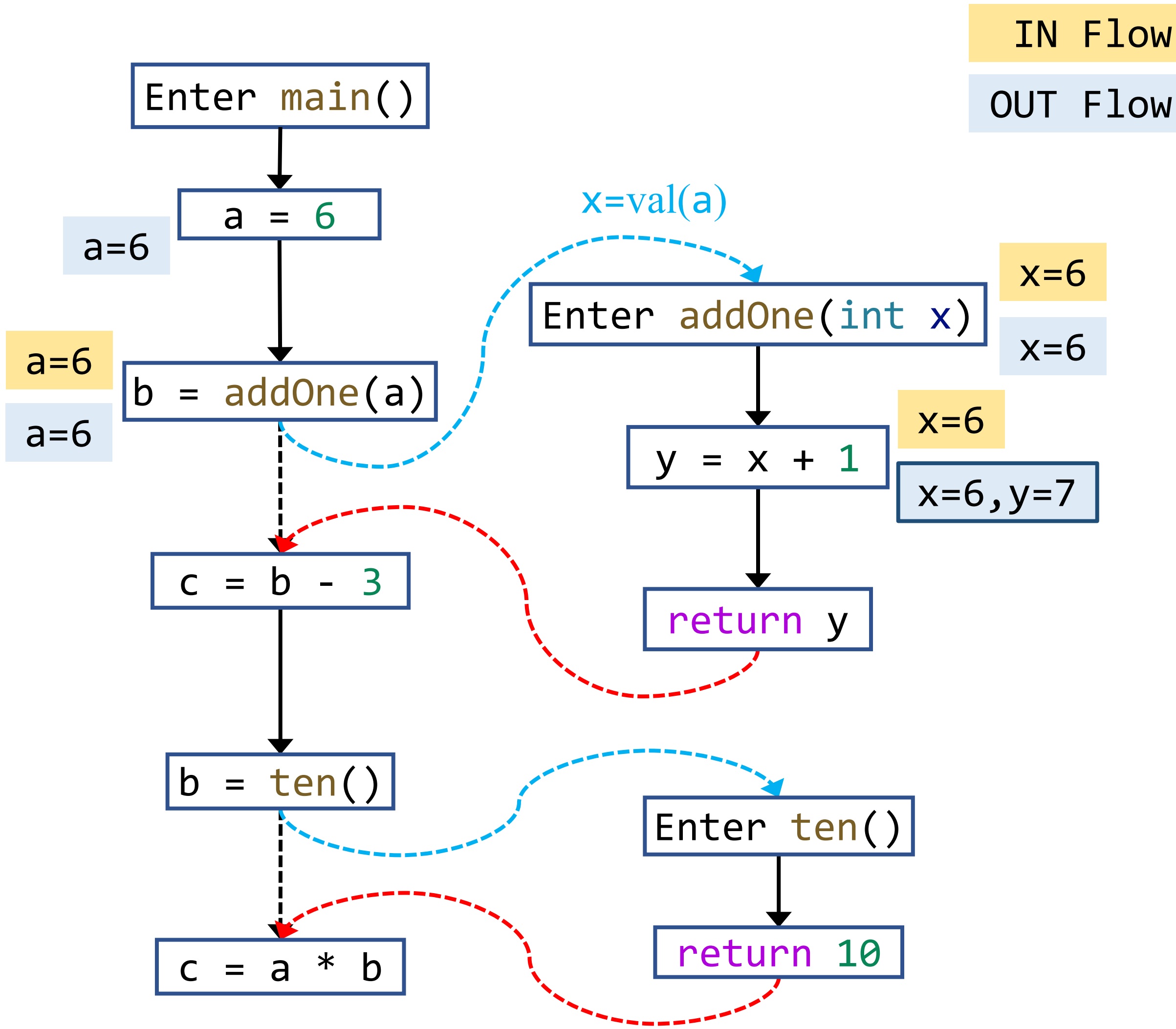
Interprocedural Constant Propagation: An Example

```
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;
}
```



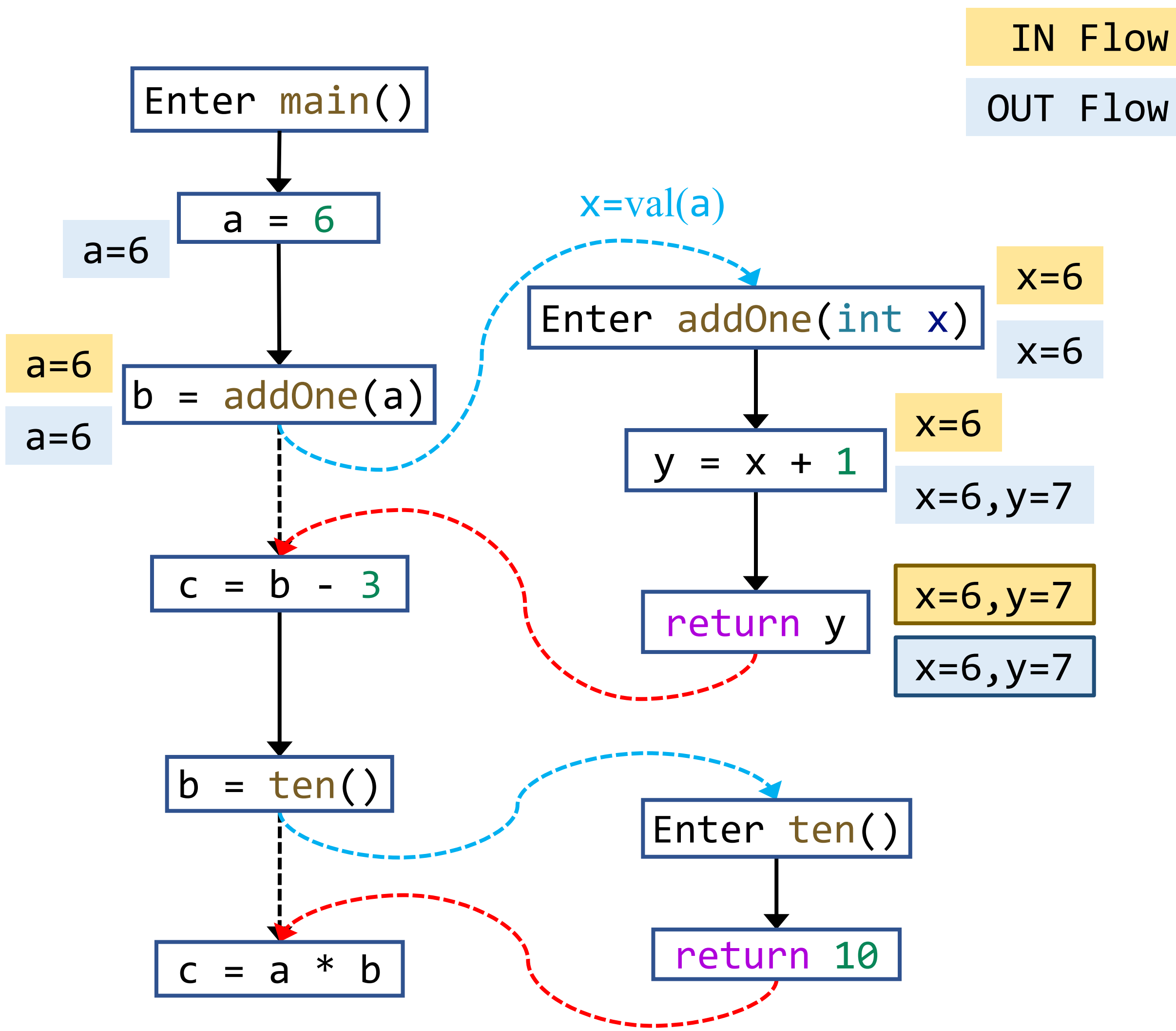
Interprocedural Constant Propagation: An Example

```
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;
}
```



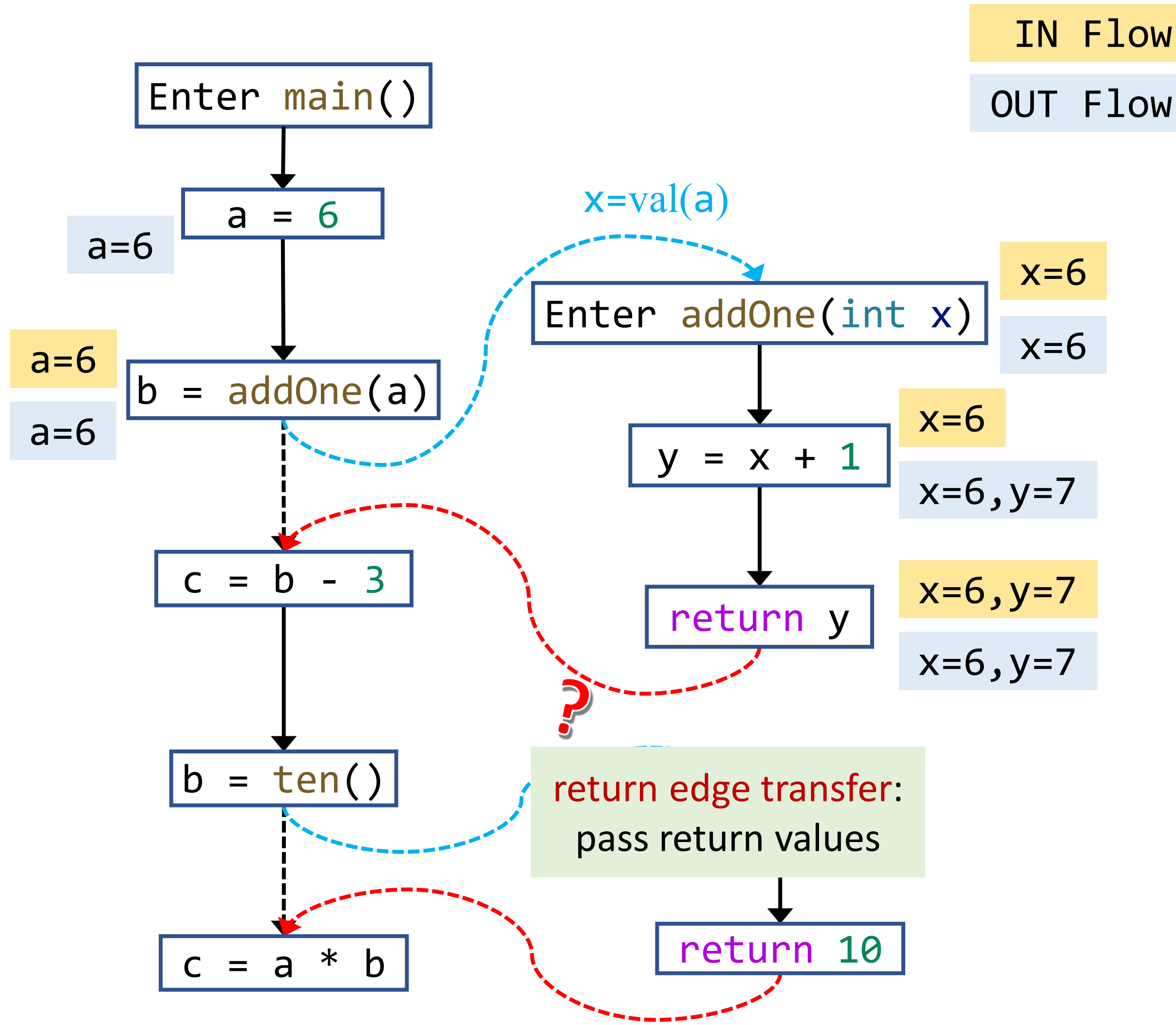
Interprocedural Constant Propagation: An Example

```
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;
}
```



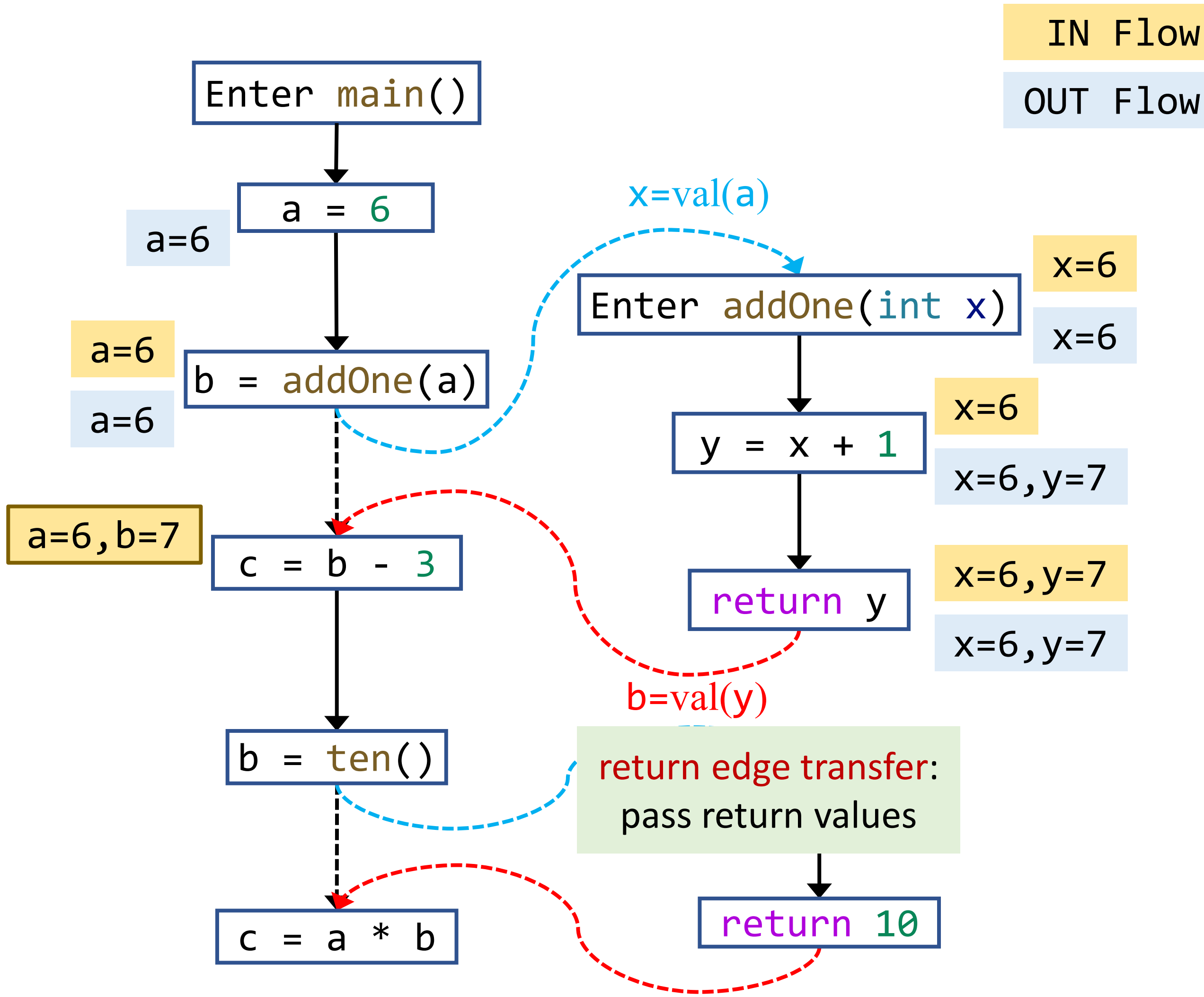
Interprocedural Constant Propagation: An Example

```
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;
}
```



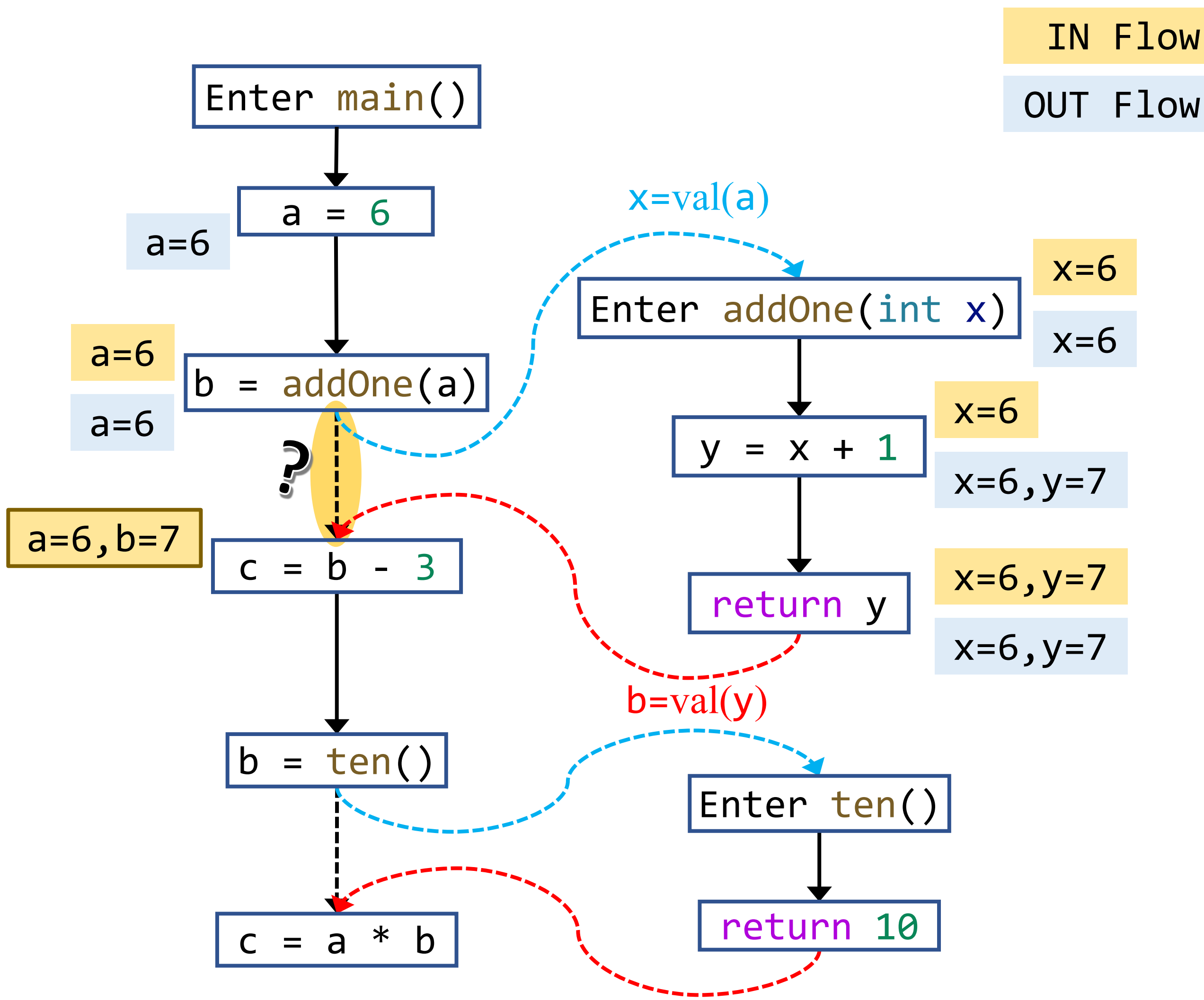
Interprocedural Constant Propagation: An Example

```
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;
}
```



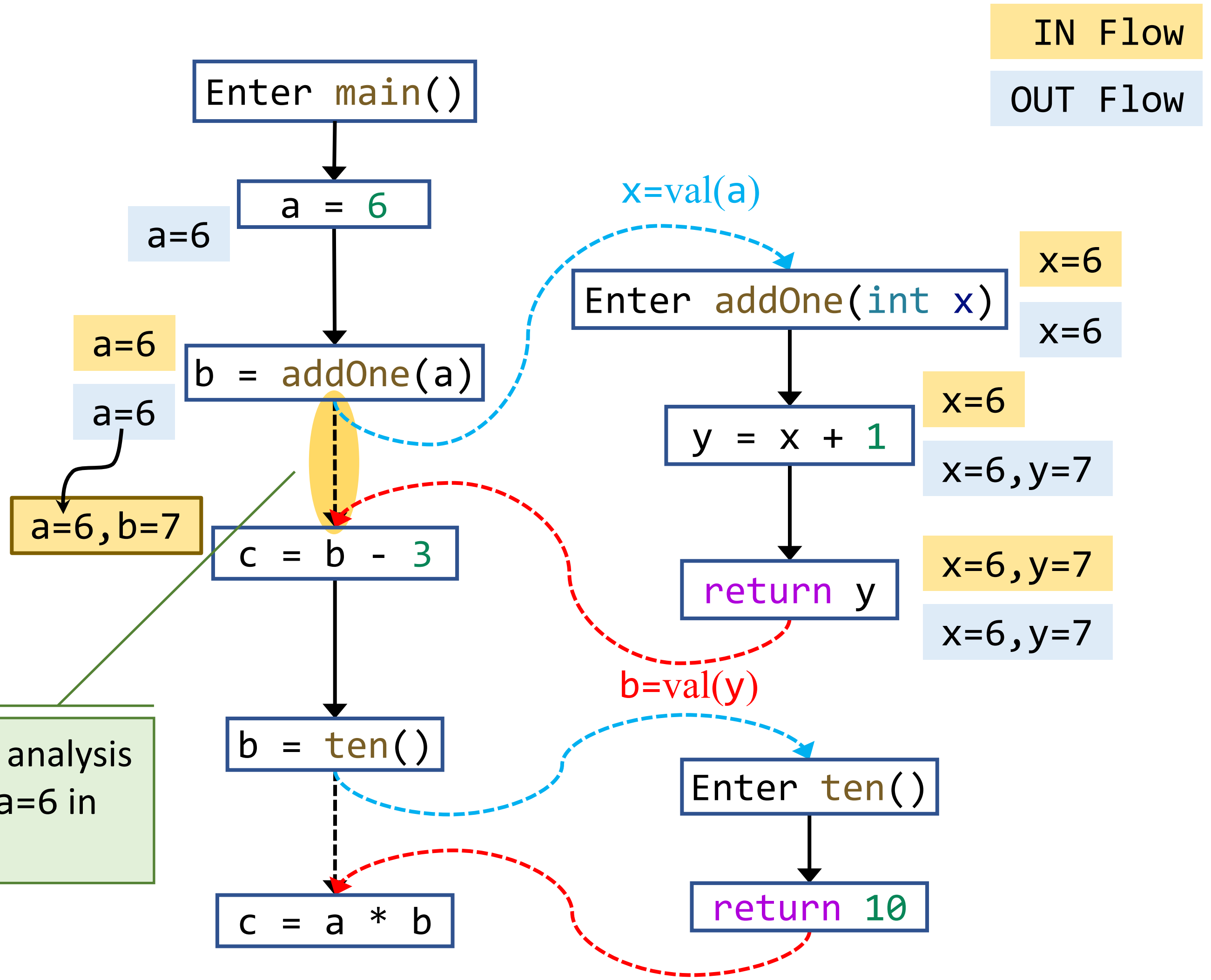
Interprocedural Constant Propagation: An Example

```
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

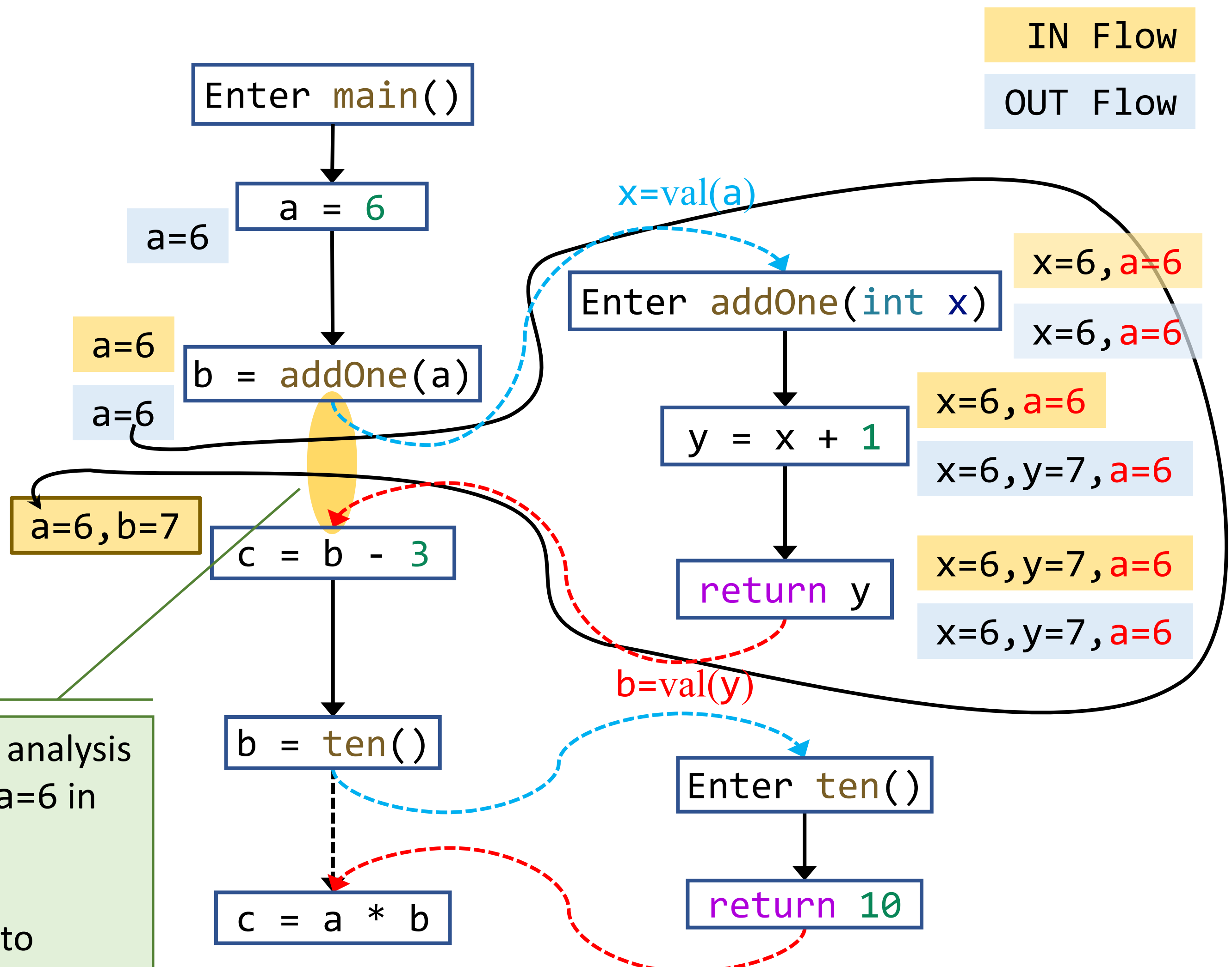
```
static void main() {
  int a, b, c;
  a = 6;
  b = addOne(a);
  c = b - 3;
  b = ten();
  c = a * b;
}

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



Call-to-return edges allow 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;
}
```

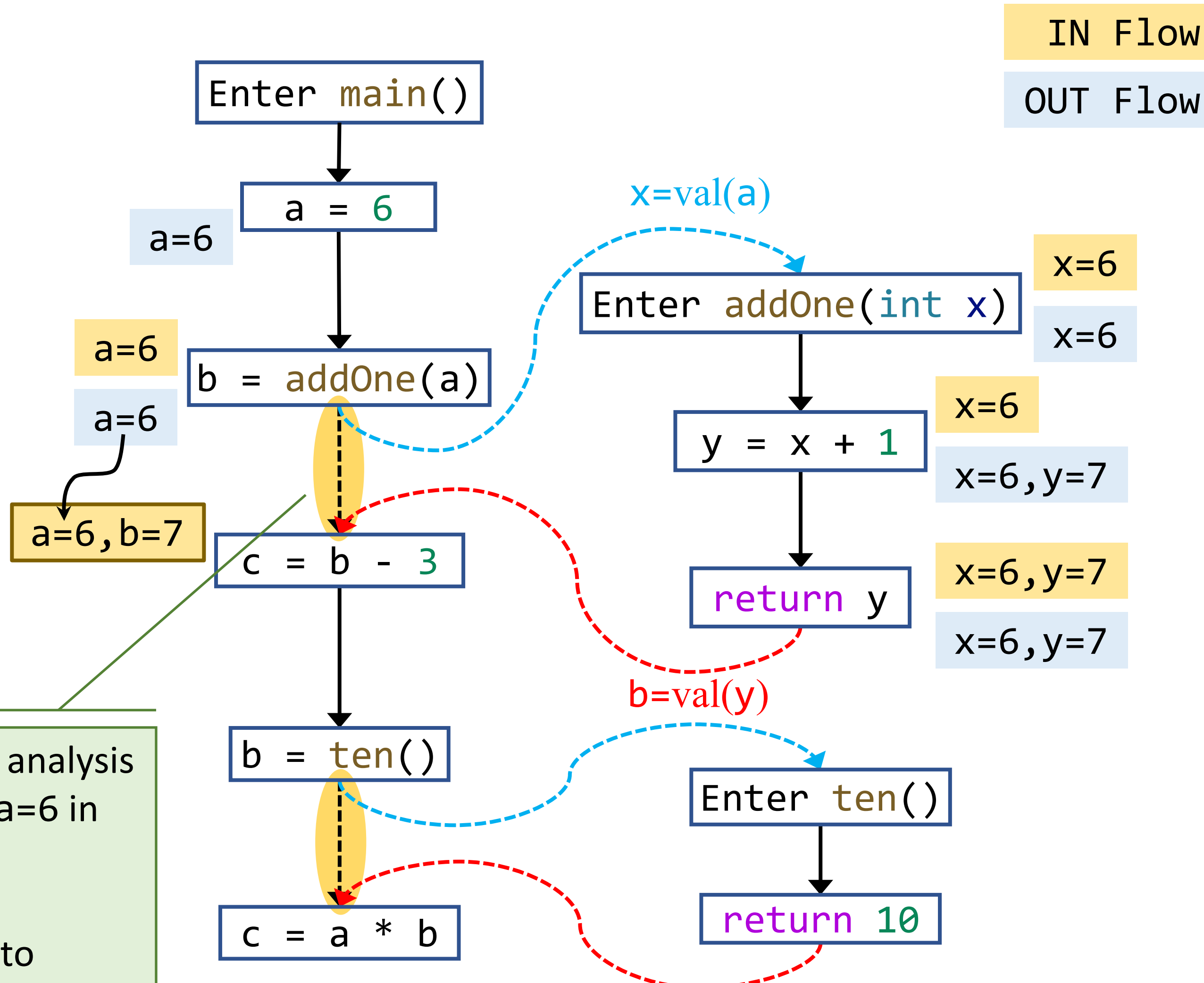


Call-to-return edges allow the analysis to propagate local data-flow (a=6 in this case) on ICFG.

Without such edges, we have to propagate local data-flow across other methods, which is very inefficient.

Interprocedural Constant Propagation: An Example

```
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;
}
```

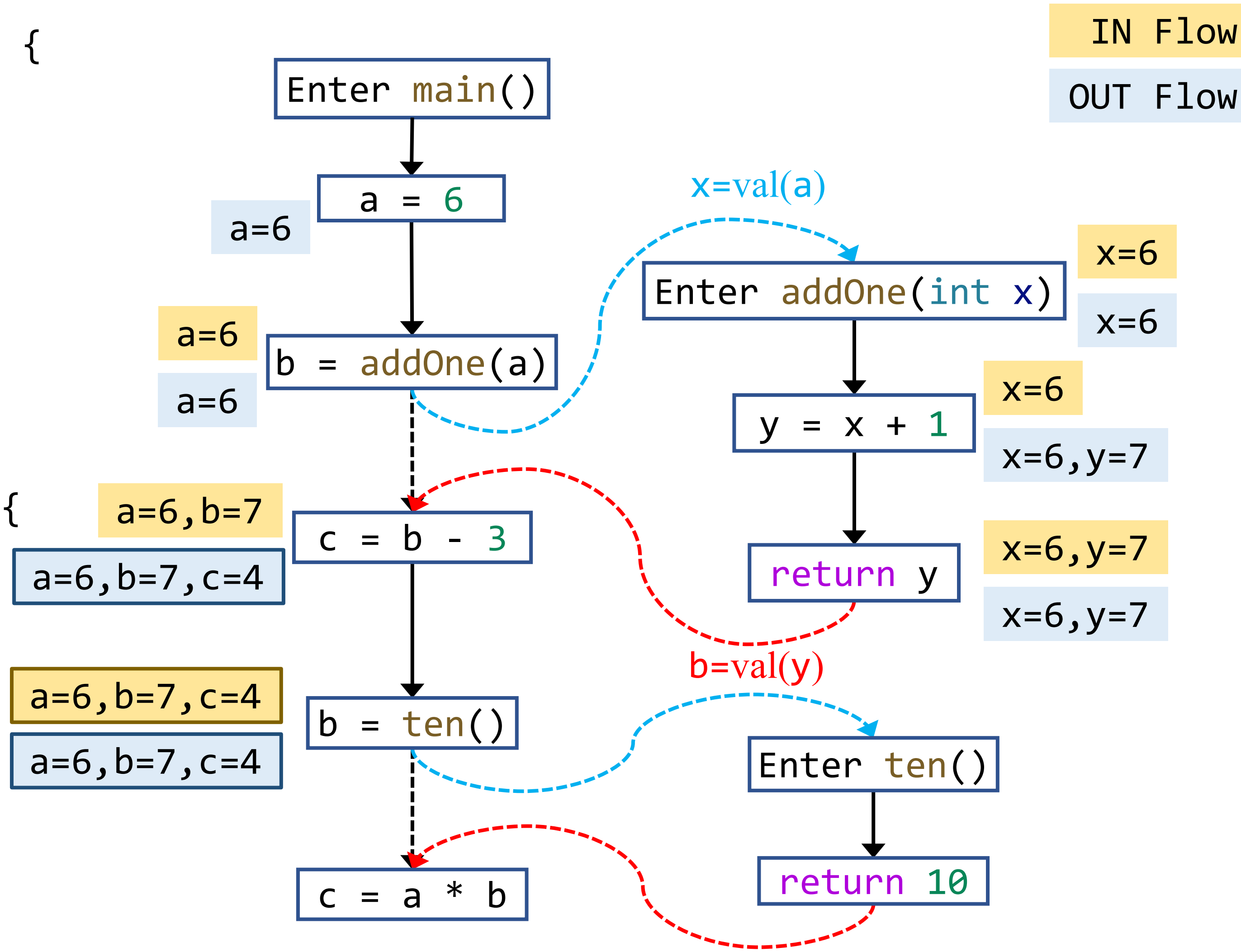


Call-to-return edges allow the analysis to propagate local data-flow (a=6 in this case) on ICFG.

Without such edges, we have to propagate local data-flow across other methods, which is very inefficient.

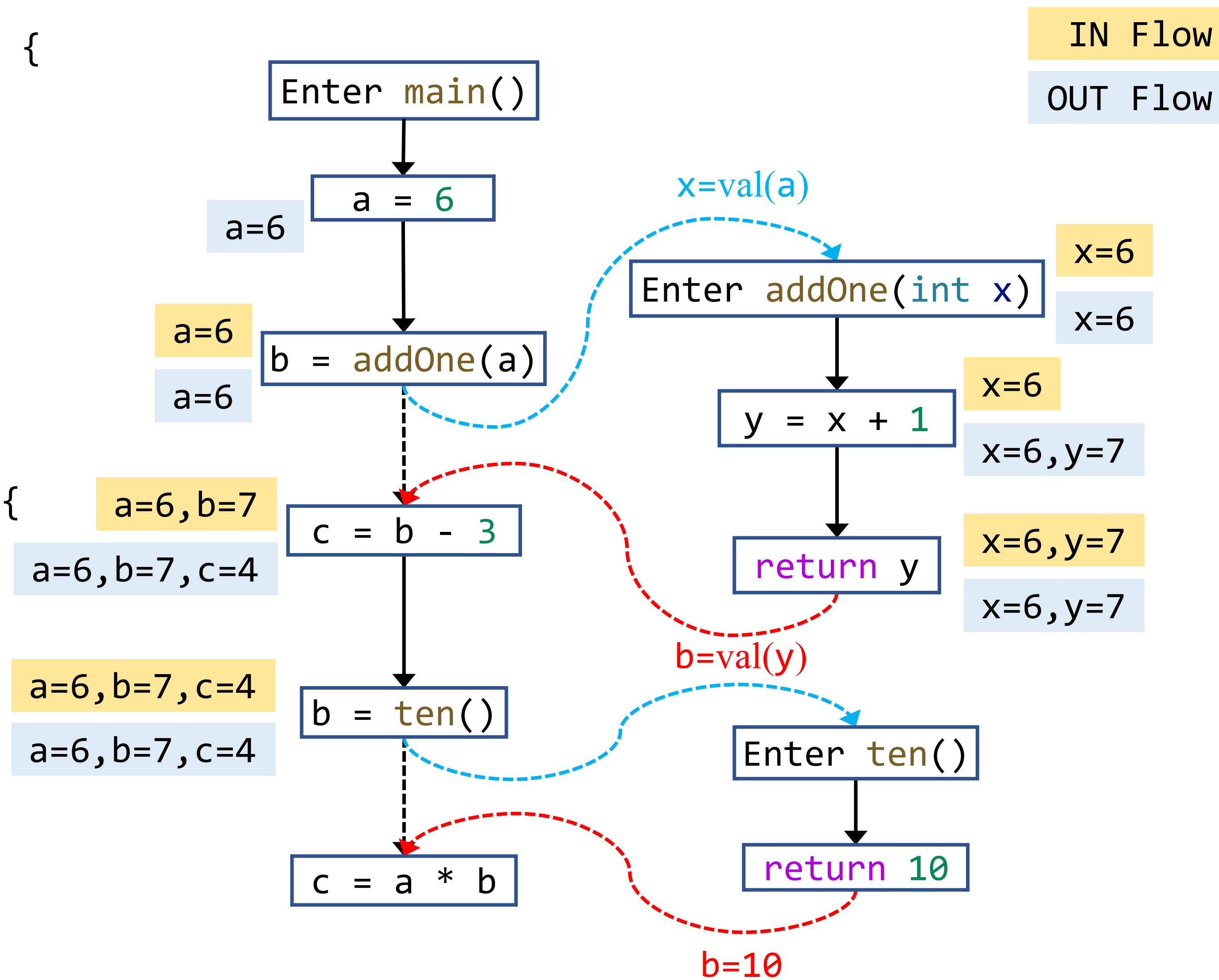
Interprocedural Constant Propagation: An Example

```
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

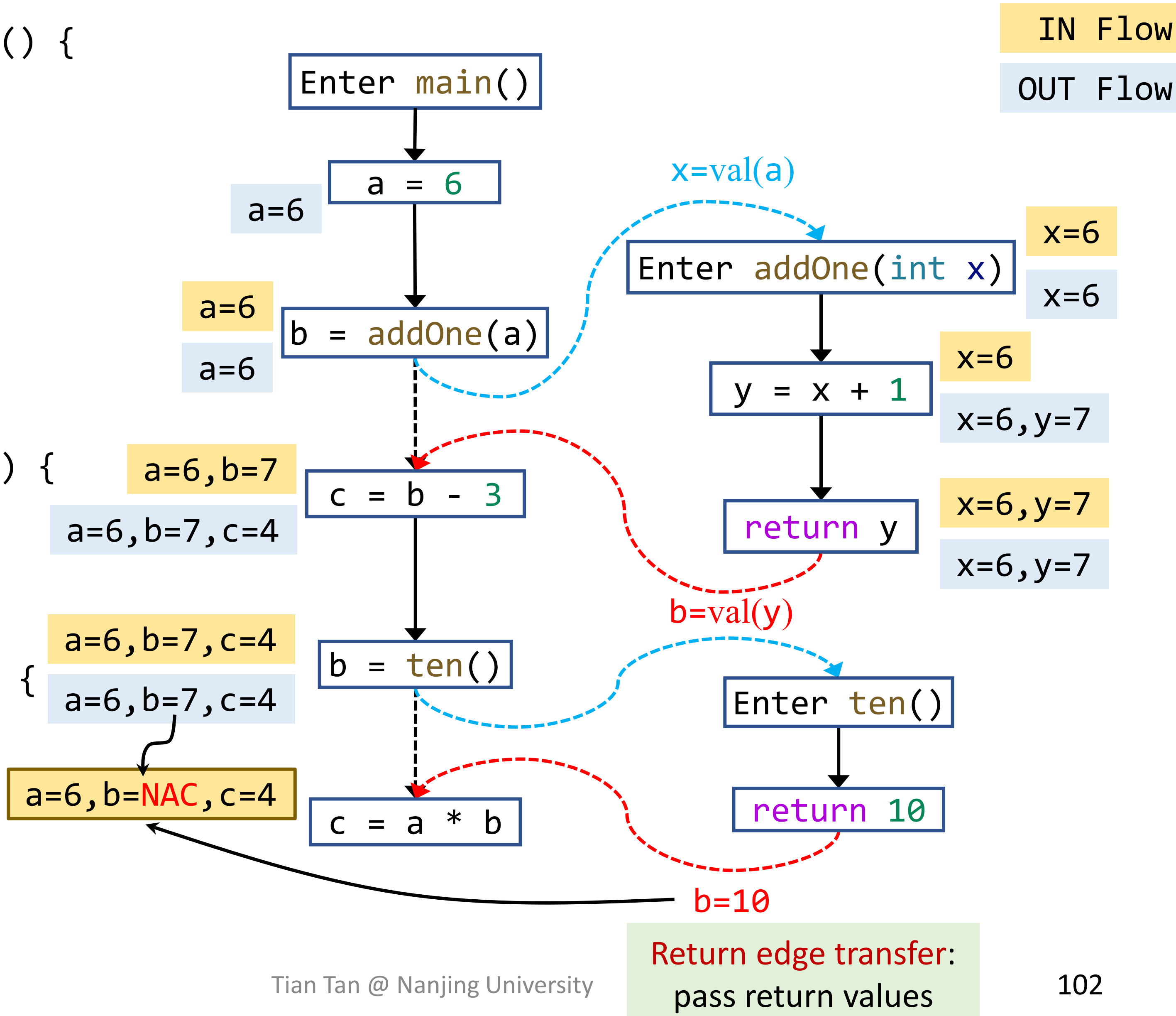
```
static void main() {
  int a, b, c;
  a = 6;
  b = addOne(a);
  c = b - 3;
  b = ten();
  c = a * b;
}

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

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



Return edge transfer: pass return values

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;
}
```

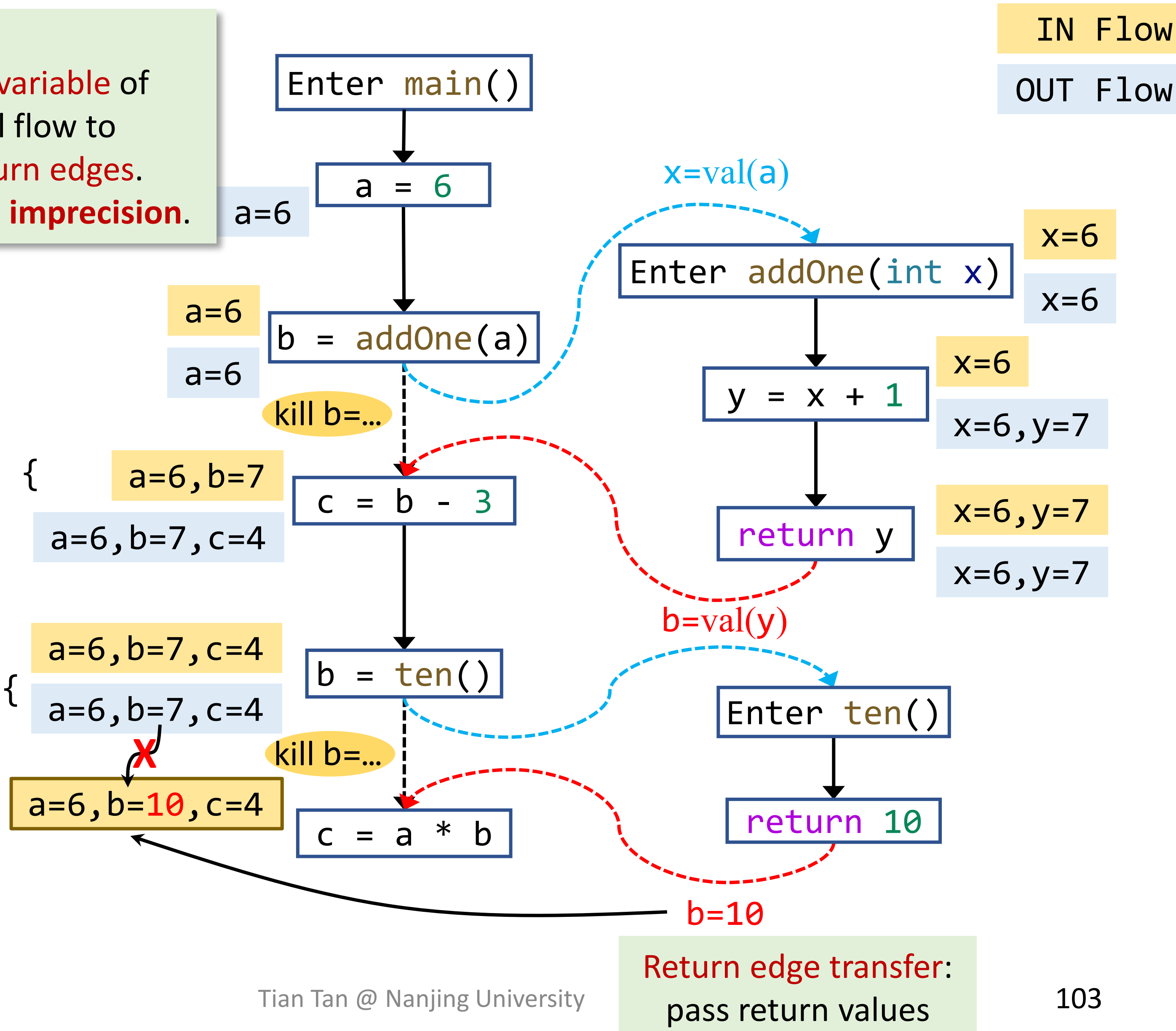


Imprecise!

Interprocedural Constant Propagation: An Example

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

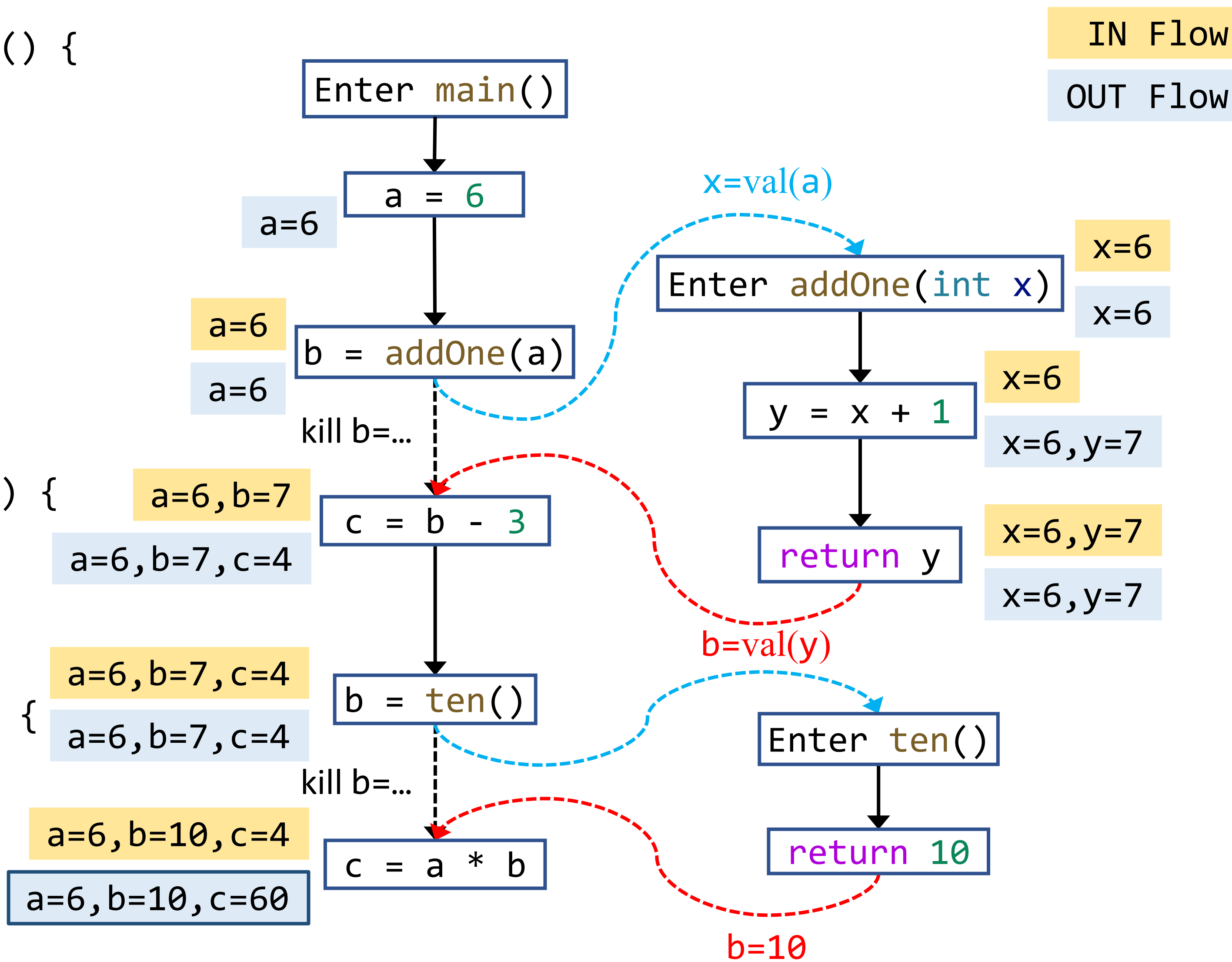
```
    b = ten();
    c = a * b;
}

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

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



Precise!

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;
}
```

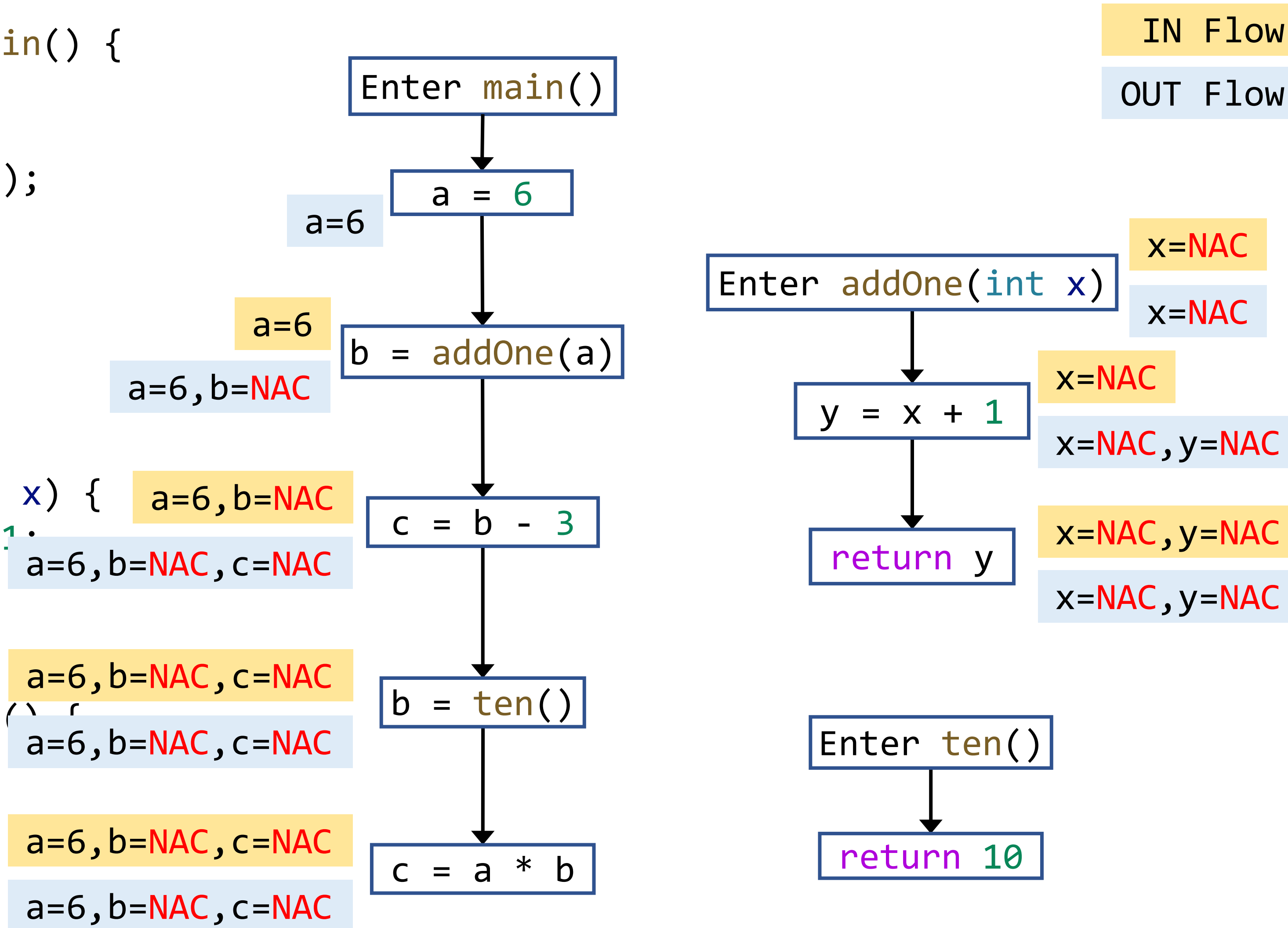


Interprocedural Constant Propagation, In Summary

- Node transfer
  - Call nodes: identity
  - Other nodes: same as intraprocedural constant propagation
- Edge transfer
  - Normal edges: identity
  - Call-to-return edges: kill the value of LHS variable of the call site, propagate values of other local variables
  - Call edges: pass argument values
  - Return edges: pass return values

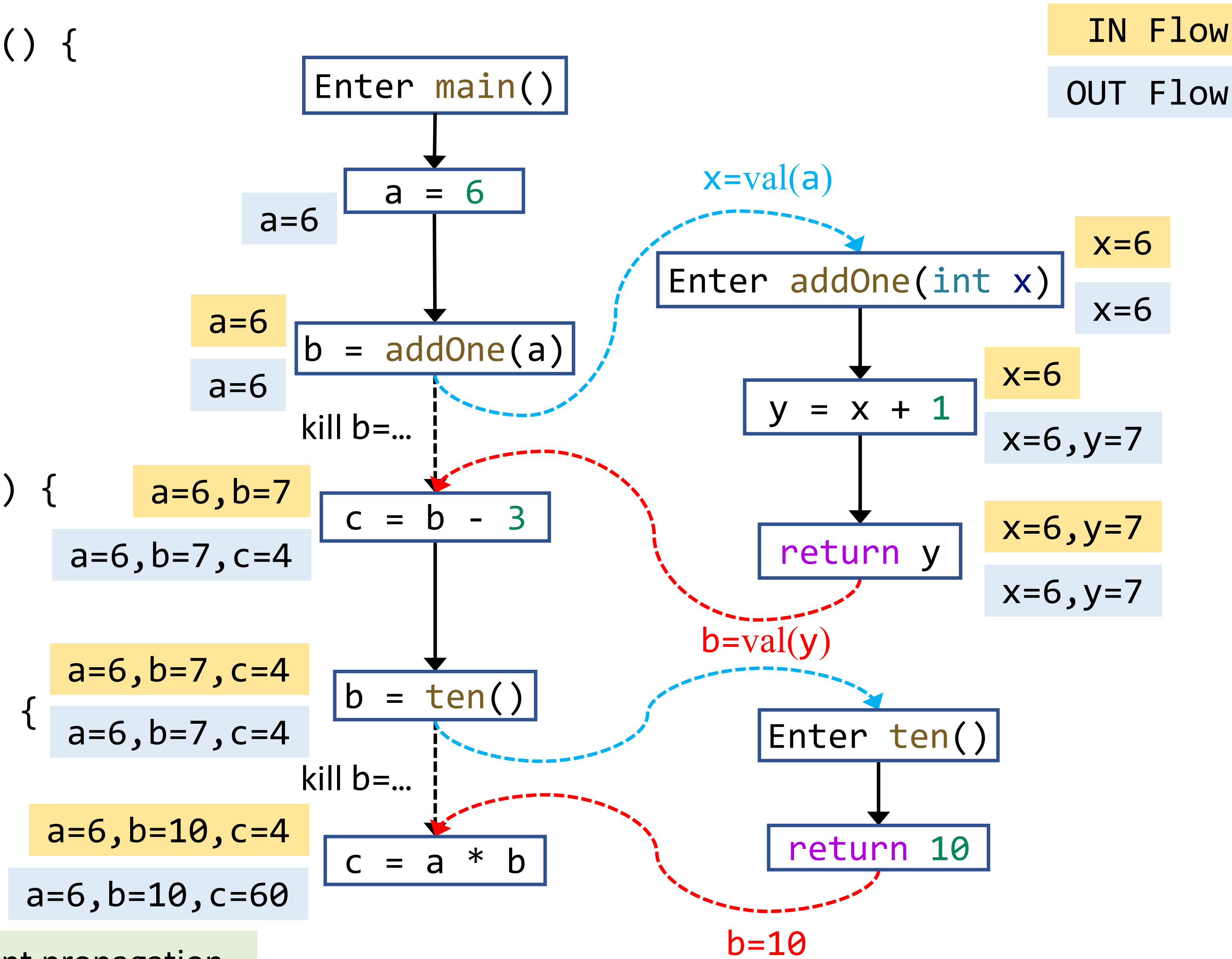
Intraprocedural Constant Propagation: An Example

```
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;
}
```



> ***Inter***procedural 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