

Static Program Analysis

Yue Li and Tian Tan



2020 Spring

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

Contents

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?

```
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

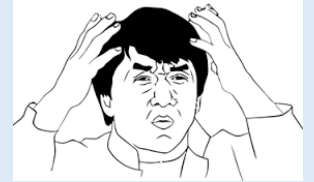
Constant Propagation

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

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

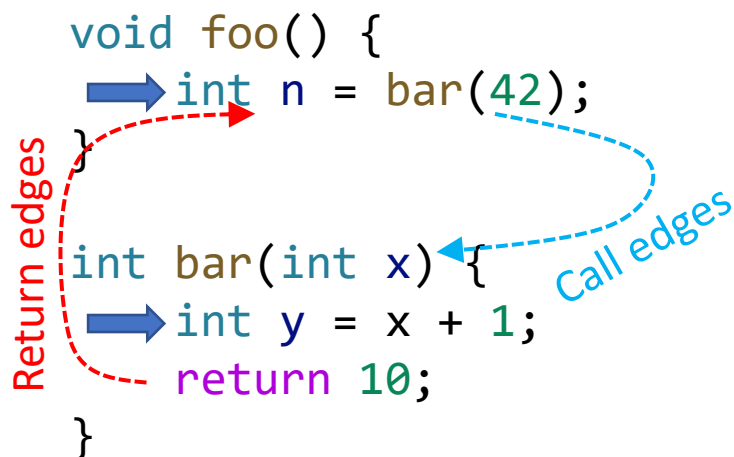
So far, all analyses we learnt are **intraprocedural**.
How to deal with method calls?

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



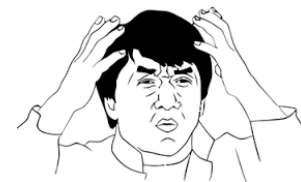
Motivation of Interprocedural Analysis

Constant Propagation



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

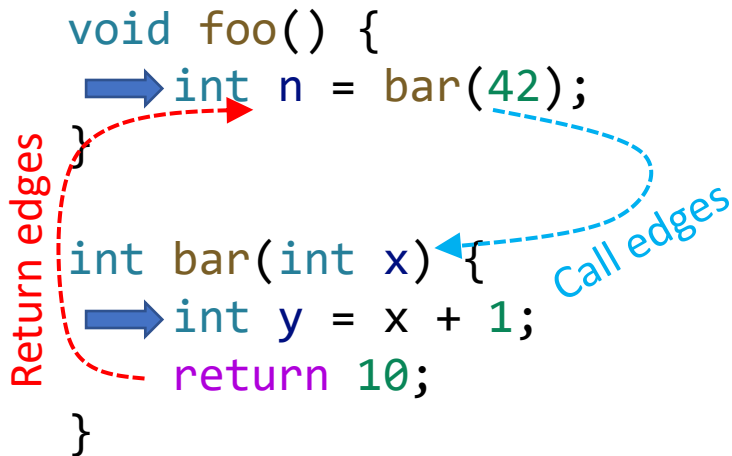
- Make the **most conservative assumption** for method calls, for safe-approximation
- Source of **imprecision**
 - $x = \text{NAC}, y = \text{NAC}$
 - $n = \text{NAC}$



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

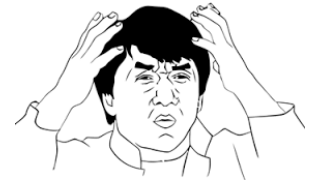
Motivation of Interprocedural Analysis

Constant Propagation



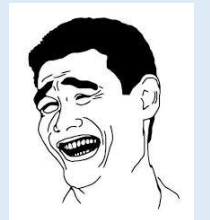
So far, all analyses we learnt are **intraprocedural**. How to deal with method calls?

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



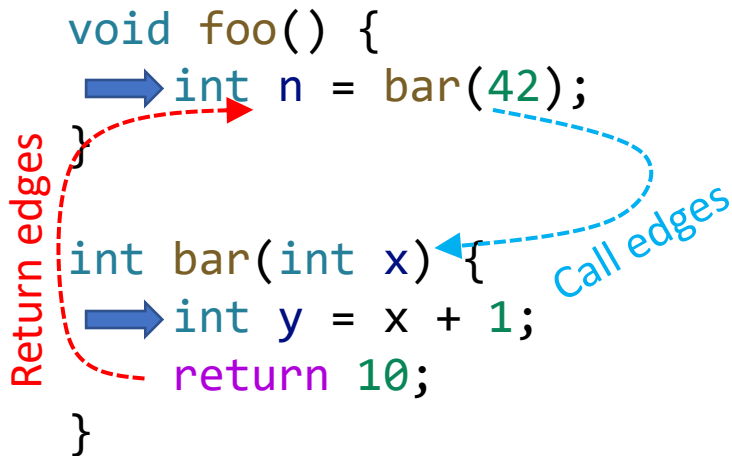
For better precision, we need **Interprocedural analysis**: propagate data-flow information along **interprocedural control-flow edges** (i.e., call and return edges)

- $x = 42, y = 43$
- $n = 10$



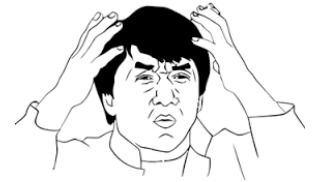
Motivation of Interprocedural Analysis

Constant Propagation



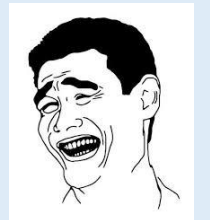
So far, all analyses we learnt are **intraprocedural**. How to deal with method calls?

- 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

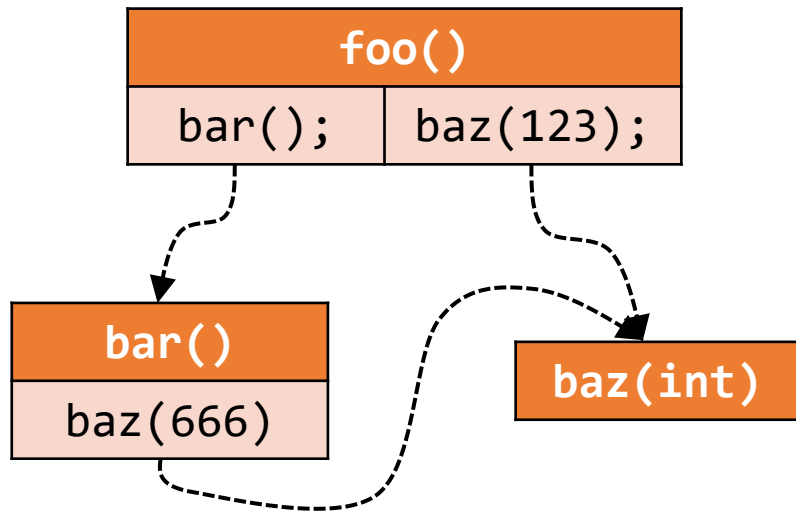
- Essentially, a call graph is a **set of call edges** from call-sites to their target methods (callees)

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)

```
void foo() {  
    bar();  
    baz(123);  
}  
  
void bar(int x) {  
    baz(666);  
}  
  
void baz() { }
```



Applications of Call Graph

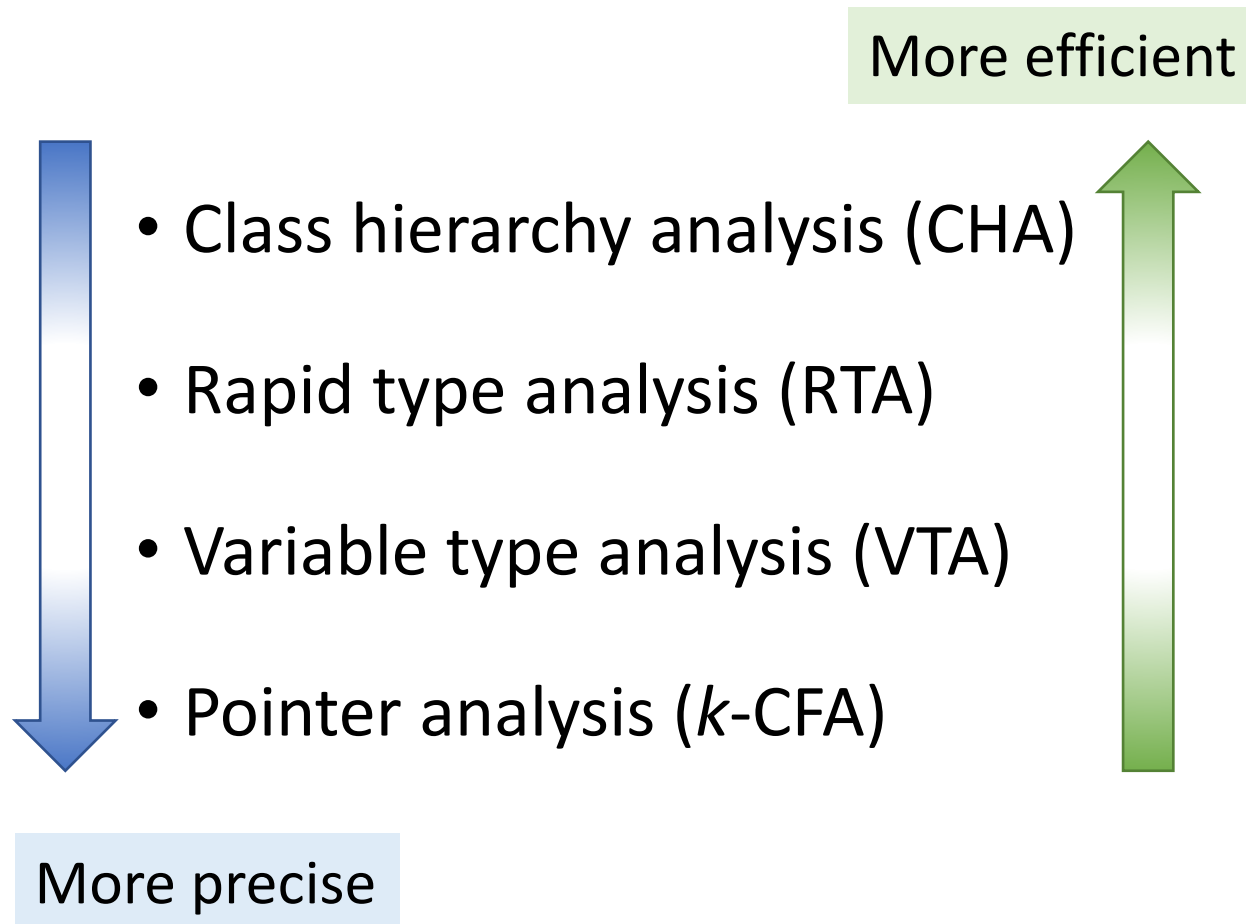
- Foundation of all interprocedural analyses
- Program optimization
- Program understanding
- Program debugging
- Program testing
- And many more ...

Call graph is **VERY important**
program information

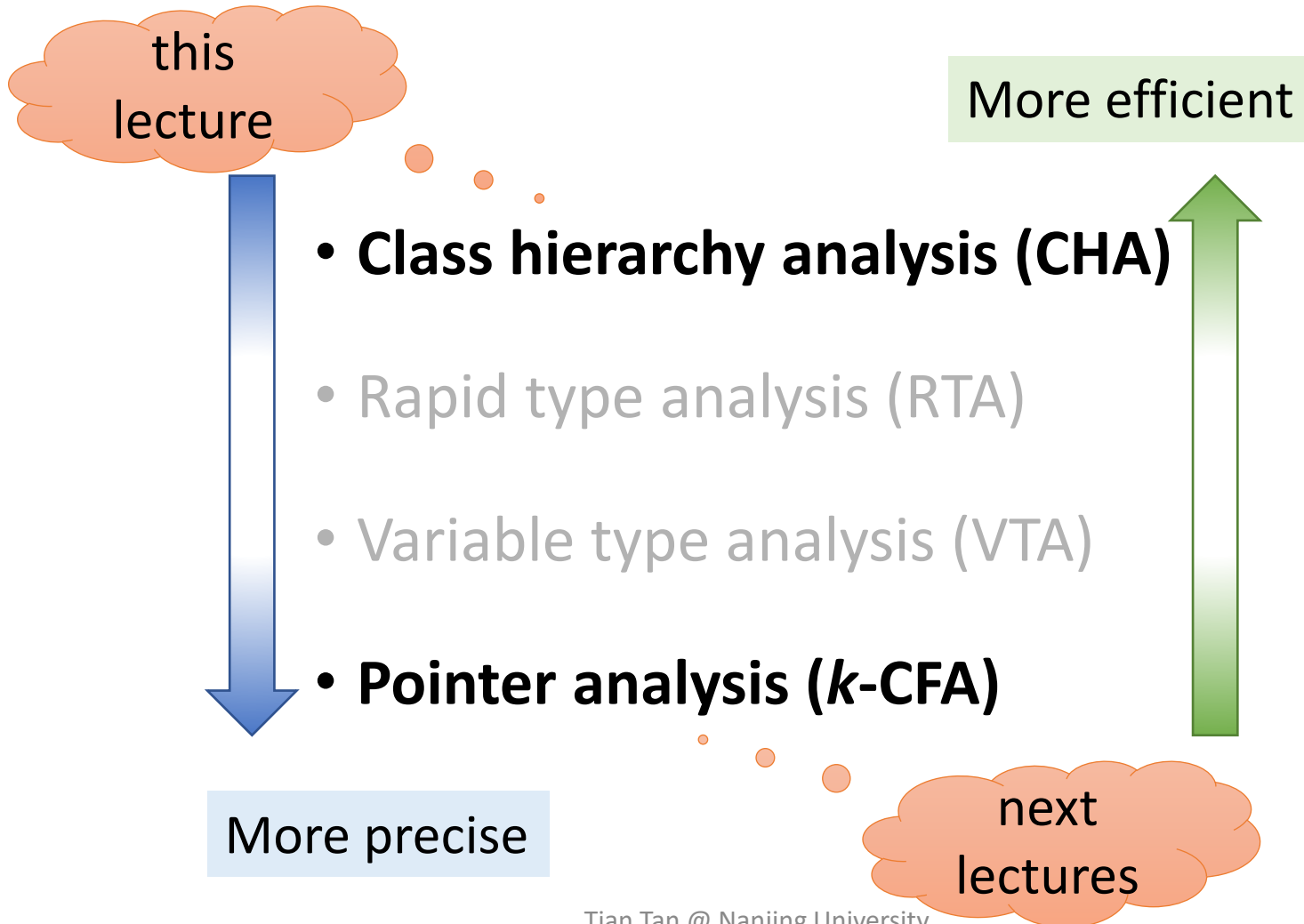
Call Graph Construction for OOPs (focus on Java)

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

Call Graph Construction for OOPs (focus on Java)



Call Graph Construction for OOPs (focus on Java)



Method Calls (Invocations) in Java

	Static call	Special call	Virtual call
Instruction	<code>invokestatic</code>	<code>invokespecial</code>	<code>invokeinterface</code> <code>invokevirtual</code>

Method Calls (Invocations) in Java

	Static call	Special call	Virtual call
Instruction	<code>invokestatic</code>	<code>invokespecial</code>	<code>invokeinterface</code> <code>invokevirtual</code>
Receiver objects	×	✓	✓
Target methods	<ul style="list-style-type: none">• Static methods	<ul style="list-style-type: none">• Constructors• Private instance methods• Superclass instance methods	<ul style="list-style-type: none">• Other instance methods

Method Calls (Invocations) in Java

	Static call	Special call	Virtual call
Instruction	<code>invokestatic</code>	<code>invokespecial</code>	<code>invokeinterface</code> <code>invokevirtual</code>
Receiver objects	×	✓	✓
Target methods	<ul style="list-style-type: none">• Static methods	<ul style="list-style-type: none">• Constructors• Private instance methods• Superclass instance methods	<ul style="list-style-type: none">• 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	<code>invokestatic</code>	<code>invokespecial</code>	<code>invokeinterface</code> <code>invokevirtual</code>
Receiver objects	×	✓	✓
Target methods	<ul style="list-style-type: none">• Static methods	<ul style="list-style-type: none">• Constructors• Private instance methods• Superclass instance methods	<ul style="list-style-type: none">• Other instance methods
#Target methods	1	1	≥1 (polymorphism)
Determinacy	Compile-time	Compile-time	Run-time

Key to call graph construction for OOPs

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

$o^1 \cdot \text{foo}(\dots)^2;$

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

$o^1 \cdot \text{foo}(\dots)^2;$

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

$o^1.foo(\dots)^2;$

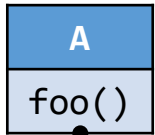
We define function **Dispatch**(c, m) to simulate the procedure of run-time method dispatch

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

where c' is superclass of c

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

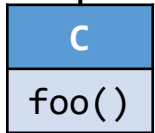
Dispatch: An Example



```
class A {  
    void foo() {...}  
}
```



```
class B extends A {  
}
```



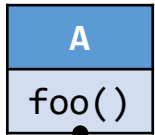
```
class C extends B {  
    void foo() {...}  
}
```

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

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

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

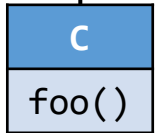
Dispatch: An Example



```
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();  
}
```

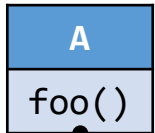
$\text{Dispatch}(c, m) = \begin{cases} m', & \text{if } c \text{ contains non-abstract method } m' \text{ that} \\ & \text{has the same name and descriptor as } m \\ \text{Dispatch}(c', m), & \text{otherwise} \end{cases}$

where c' is superclass of c

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

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

Dispatch: An Example



```
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();  
}
```

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

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

```
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 a 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

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 {  
    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 a 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*

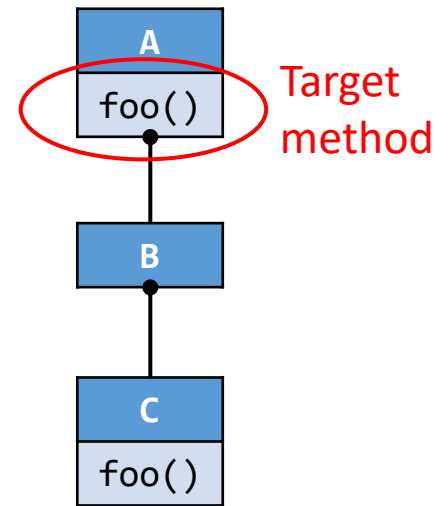
return *T*

```
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)>  
cm 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 a 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 a 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 a 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 {  
    T foo(P p, Q q) {...}  
}  
C c = ...  
c.foo(x, y); ←
```

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

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

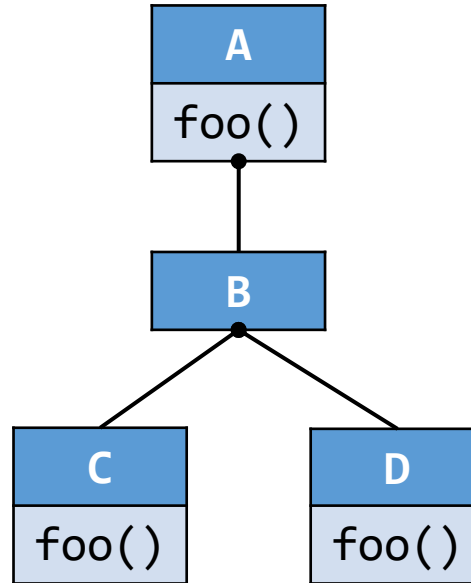
```
cs  c.foo(x, y);  
m   <C: T foo(P,Q)>  
c   C
```

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

CHA: An Example

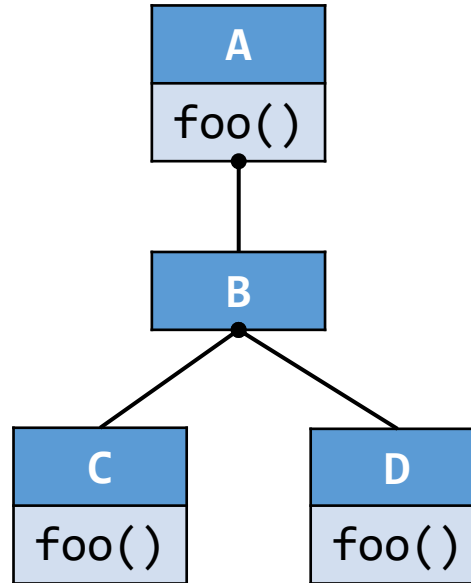
```
class A {  
    void foo() {...}  
}  
class B extends A {}  
  
class C extends B {  
    void foo() {...}  
}  
class D extends B {  
    void foo() {...}  
}
```

```
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() {...}  
}
```

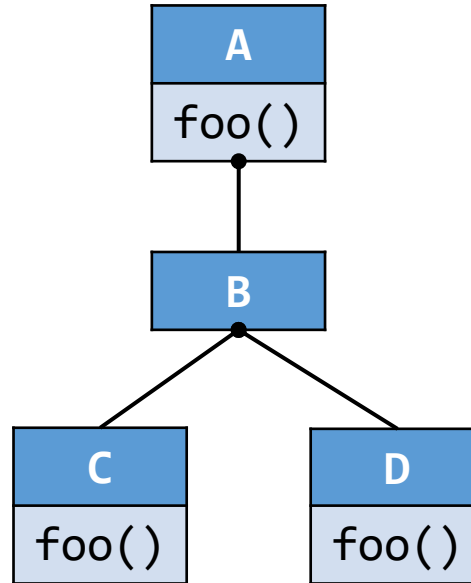


```
void resolve() {  
    C c = ...  
    c.foo();  
  
    A a = ...  
    a.foo();  
  
    B b = ...  
    b.foo();  
}
```

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() {...}  
}
```



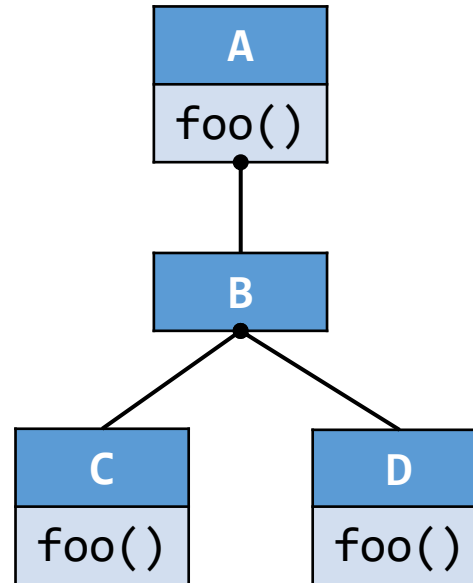
```
void resolve() {  
    C c = ...  
    c.foo();  
  
    A a = ...  
    a.foo();  
  
    B b = ...  
    b.foo();  
}
```

$\text{Resolve}(c.\text{foo}()) = \{C.\text{foo}()\}$

$\text{Resolve}(a.\text{foo}()) = ?$

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();  
}
```

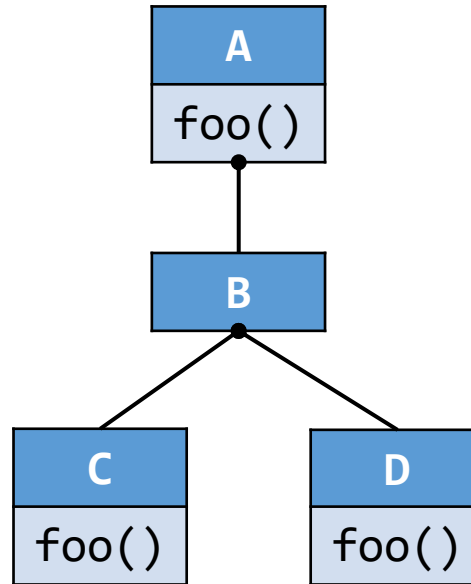
$\text{Resolve}(c.\text{foo}()) = \{C.\text{foo}()\}$

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

$\text{Resolve}(b.\text{foo}()) = ?$

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();  
}
```

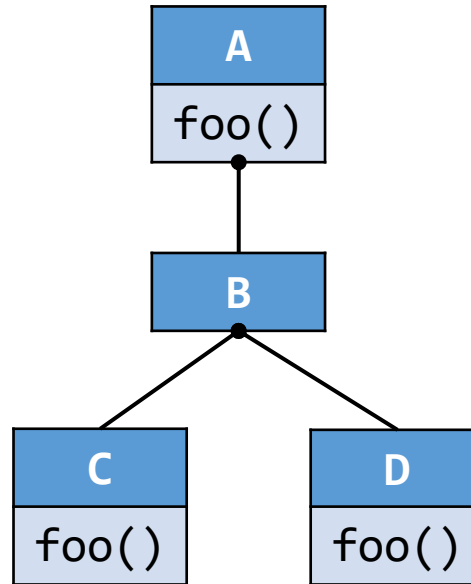
Resolve(c.foo()) = {C.foo()}

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();
```

$\text{Resolve}(c.foo()) = \{C.foo()\}$

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

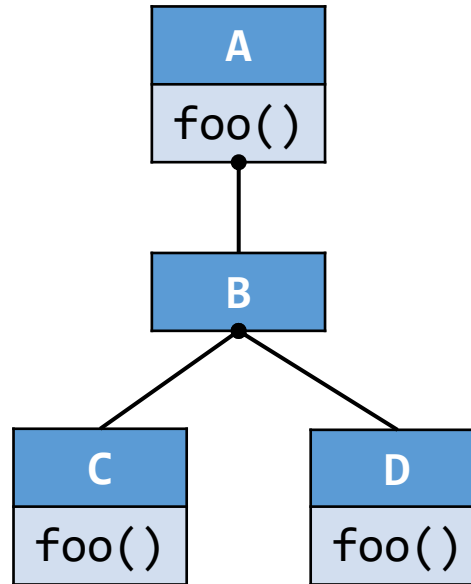
$\text{Resolve}(a.foo()) = \{A.foo(), C.foo(), D.foo()\}$

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

$\text{Resolve}(b.foo()) = ?$

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();
```

$\text{Resolve}(c.\text{foo}()) = \{C.\text{foo}()\}$

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

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

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

$\text{Resolve}(b.\text{foo}()) = \{A.\text{foo}(), \underline{C.\text{foo}(), D.\text{foo}()}\}$

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)

The screenshot displays the IntelliJ IDEA IDE interface. On the left, the 'TestCHA.java' file is open, showing the following code:

```
1 public class TestCHA {  
2     void test() {  
3         B b = new B();  
4         b.foo();  
5     }  
6 }  
7 class A {  
8     void foo() {}  
9 }  
10 class B extends A {}  
11 class C extends B {  
12     void foo() {}  
13 }  
14 class D extends B {  
15     void foo() {}  
16 }
```

The line containing `b.foo();` is highlighted in yellow. On the right, the 'Hierarchy: Callees of foo' tool window is open, showing a list of methods that are callees of the selected `foo()` method. The list includes:

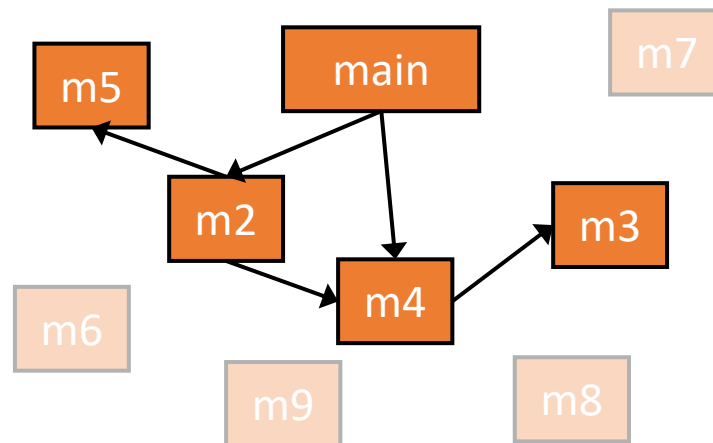
- * m ◦ A.foo() ()
- m ◦ C.foo() ()
- m ◦ D.foo() ()

The tool window also features a 'Scope: All' dropdown menu and several navigation icons.

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 (**Resolve**(cs))
- Repeat until no new method is discovered



Call Graph Construction: Algorithm

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

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

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^{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**

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

foreach target method m' in T **do**

add $cs \rightarrow m'$ to CG

add m' to WL

Resolve target methods via CHA

return CG

WL Work list, containing the methods to be processed

CG Call graph, a set of call edges

RM A set of reachable methods

Call Graph Construction: Algorithm

BuildCallGraph(m^{entry})

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

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**

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

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

return CG

WL Work list, containing the methods to be processed

CG Call graph, a set of call edges

RM A set of reachable methods

Call Graph Construction: Algorithm

BuildCallGraph(m^{entry})

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

while WL is not empty **do**

remove m from WL

if $m \notin RM$ **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

Resolve target methods via CHA

Add call edges to call graph

May discover new method,
add it to work list

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() {}
}
```

A.main()
A.foo();

Initialization with main method

$WL = [A.main()]$

A.foo()
a.bar();

A.bar()
c.bar();

B.Bar()

C.bar()
A.foo();

C.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();

A.bar()
c.bar();

B.Bar()

C.bar()
A.foo();

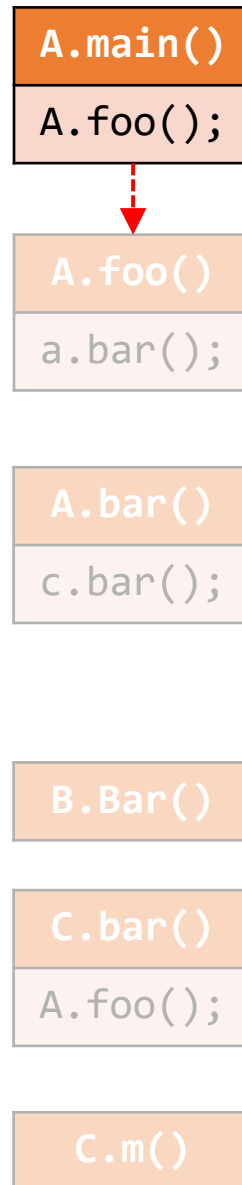
C.m()

$WL = []$

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() {}  
}
```

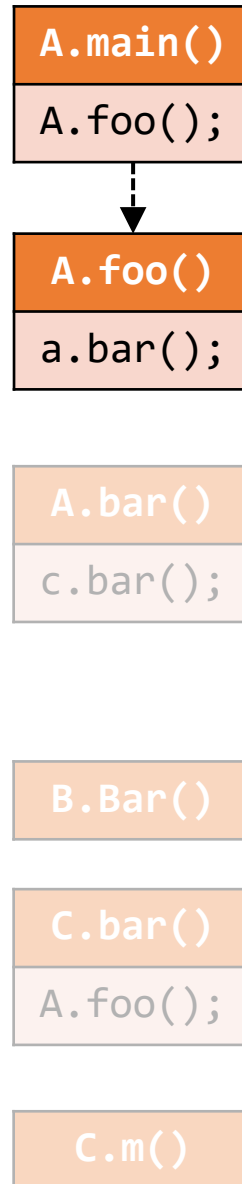


$WL = [A.foo()]$

$\text{Resolve}(A.foo()) = \{ A.foo() \}$

Call Graph Construction: An Example

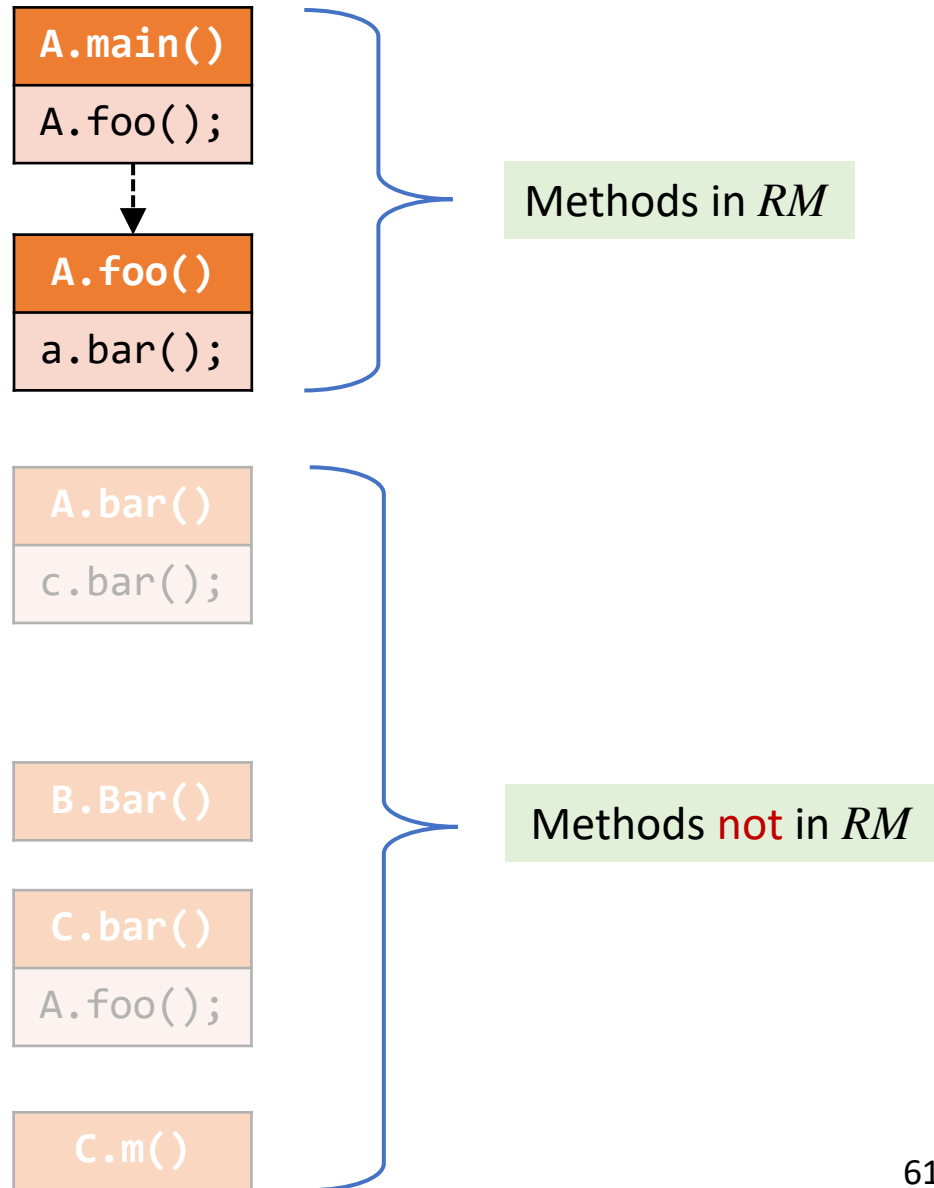
```
class A {
    static void main() {
        A.foo();
    }
    static void foo() {
        A a = new A();
        a.bar();
    }
    void bar() {
        C c = new C();
        c.bar();
    }
}
class B extends A {
    void bar() {} }
class C extends A {
    void bar() {
        if (...) A.foo();
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



$WL = []$

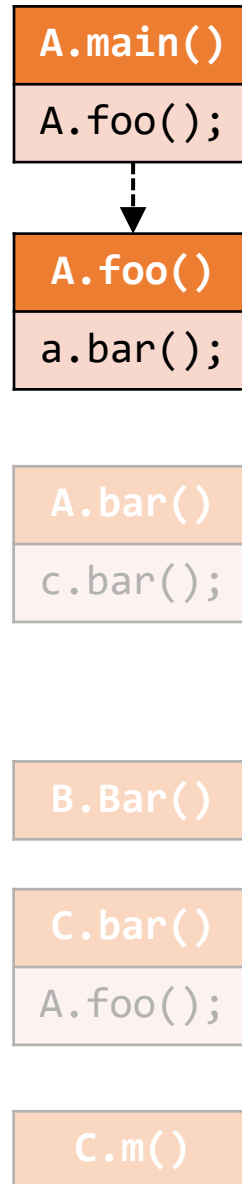
Call Graph Construction: An Example

```
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() {}
}
```

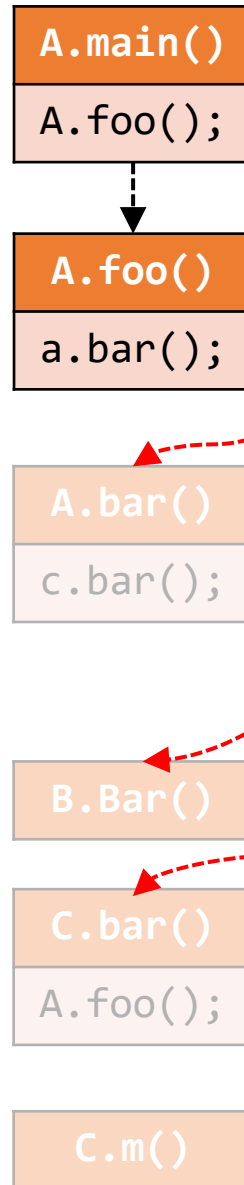


$WL = []$

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() {}  
}
```

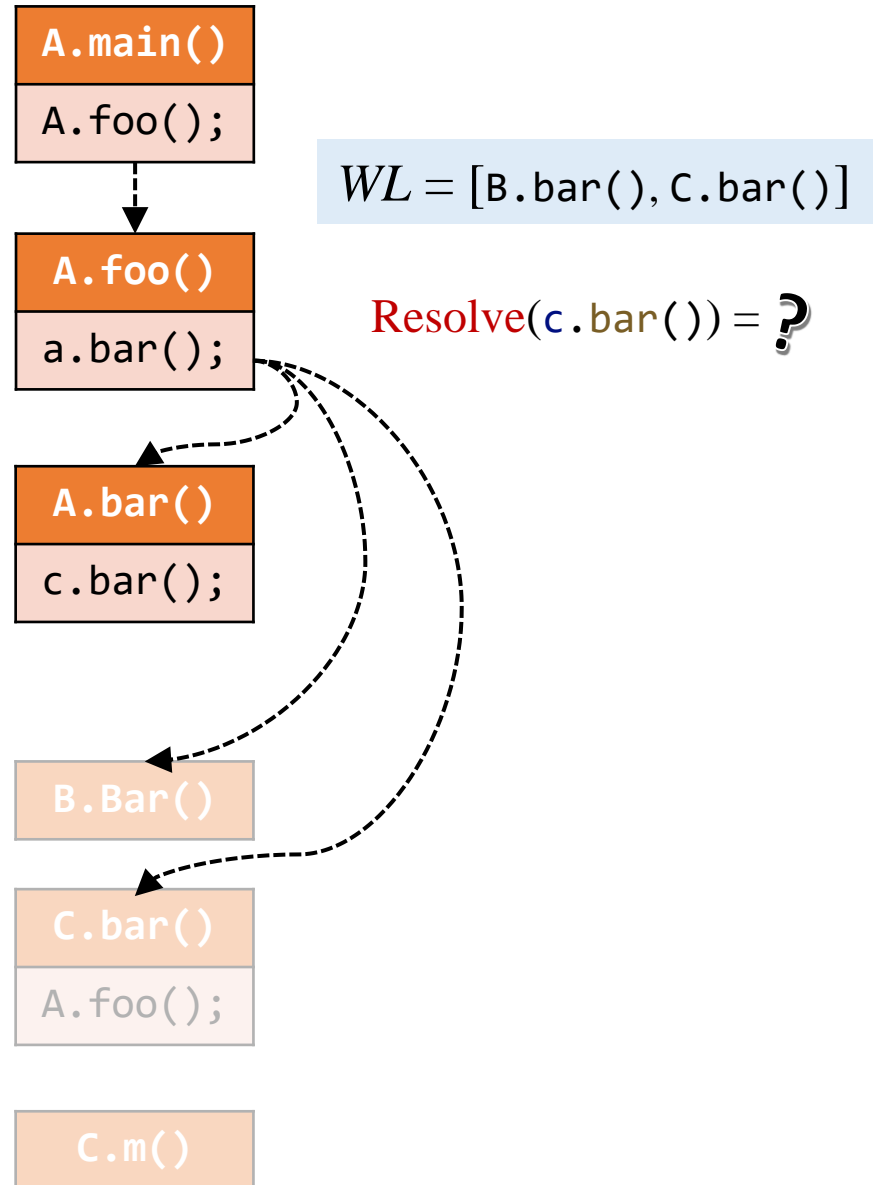


$WL = [A.bar(), B.bar(), C.bar()]$

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

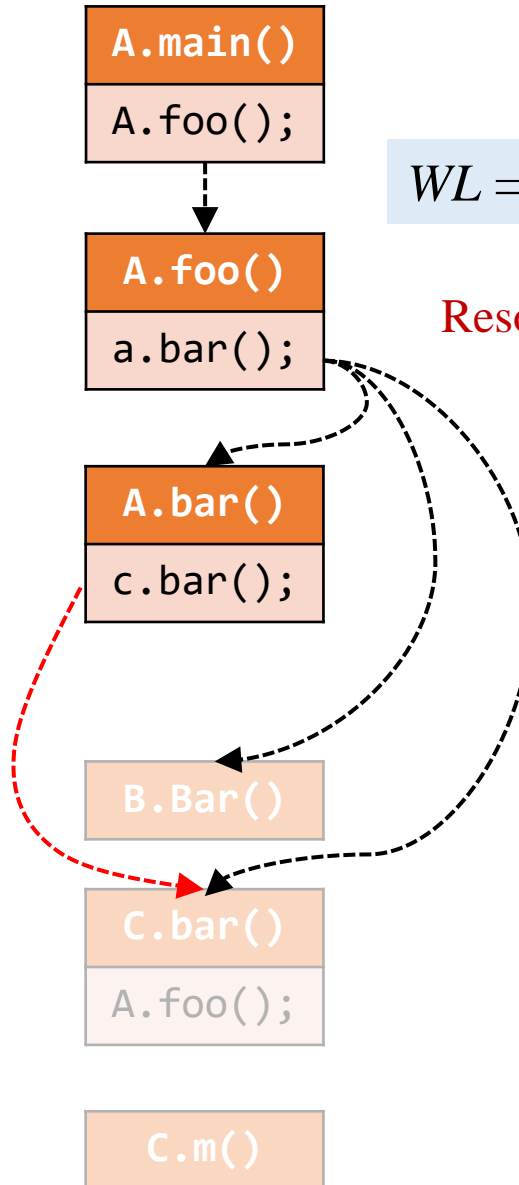
Call Graph Construction: An Example

```
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() {}
}
```



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() {}
}
```

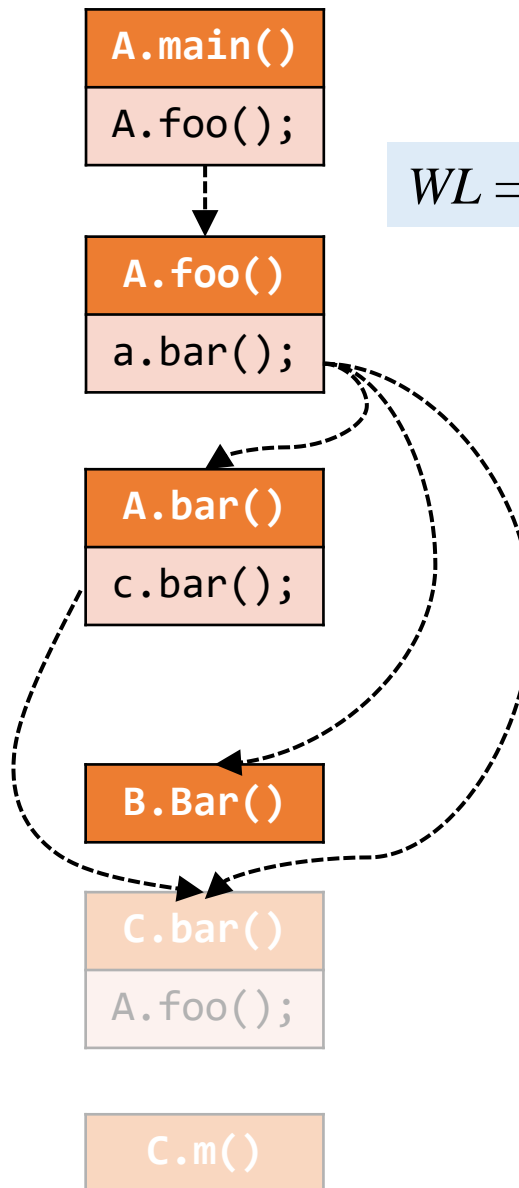


$WL = [B.bar(), C.bar(), C.bar()]$

$Resolve(c.bar()) = \{ C.bar() \}$

Call Graph Construction: An Example

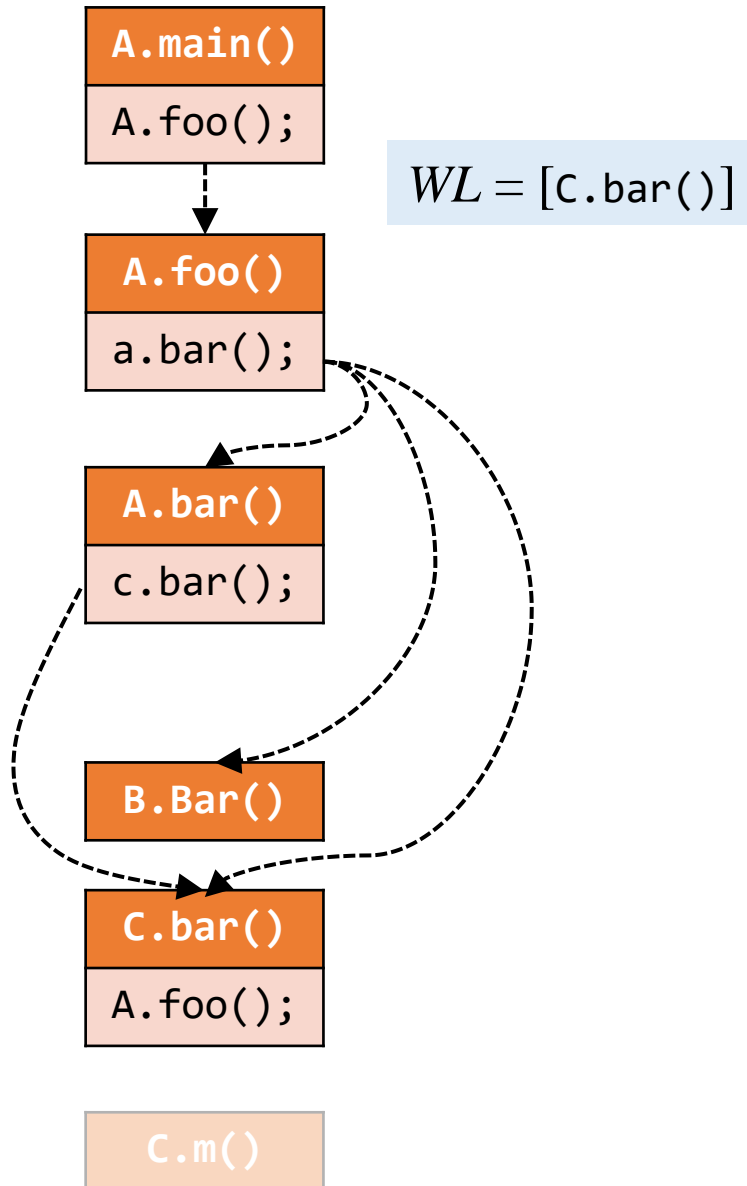
```
class A {
  static void main() {
    A.foo();
  }
  static void foo() {
    A a = new A();
    a.bar();
  }
  void bar() {
    C c = new C();
    c.bar();
  }
}
class B extends A {
  void bar() {}
}
class C extends A {
  void bar() {
    if (...) A.foo();
  }
  void m() {}
}
```



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

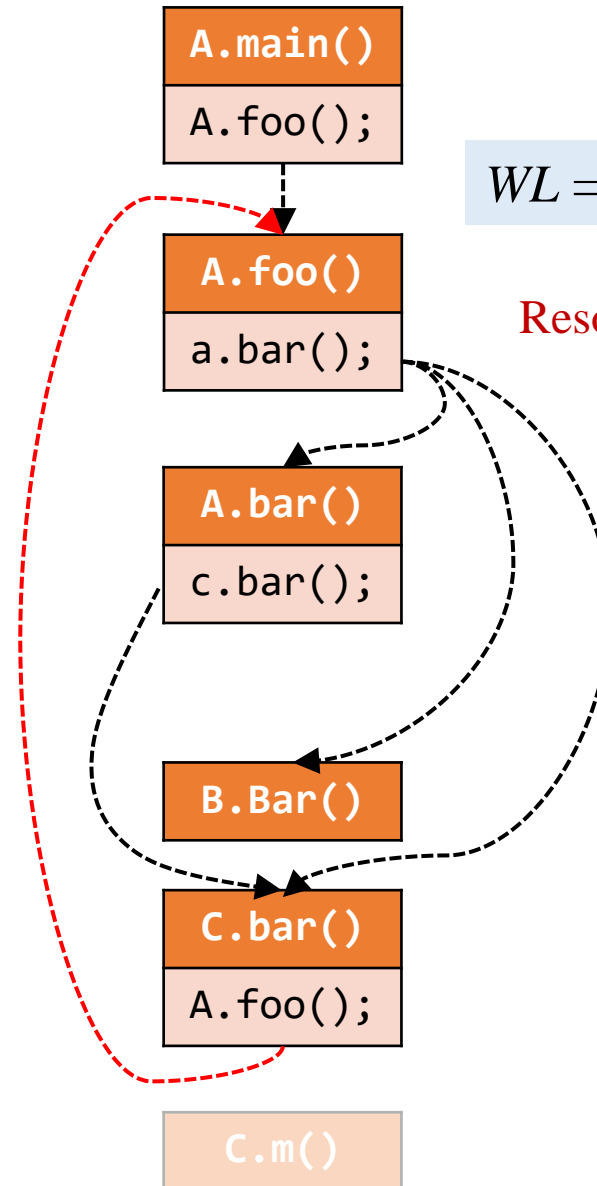
Call Graph Construction: An Example

```
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() {}
}
```



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() {}
}
```

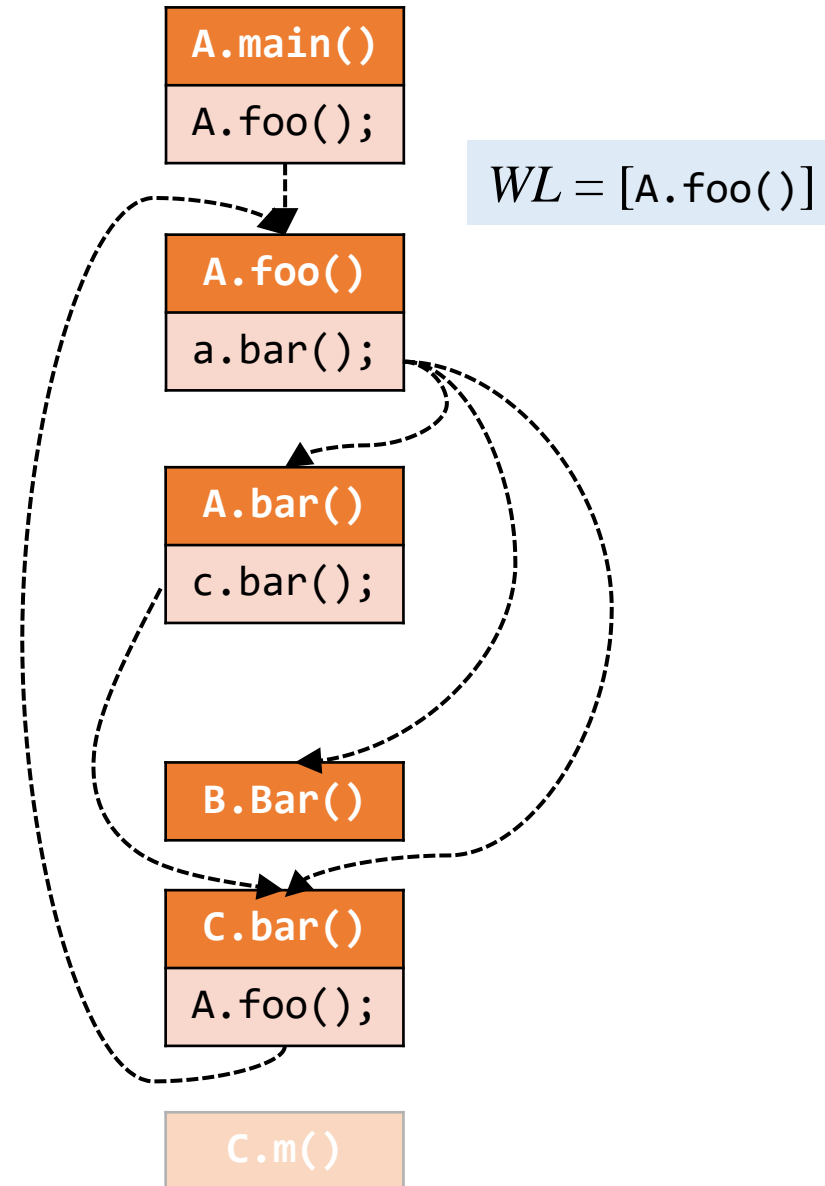


$WL = [C.bar(), A.foo()]$

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

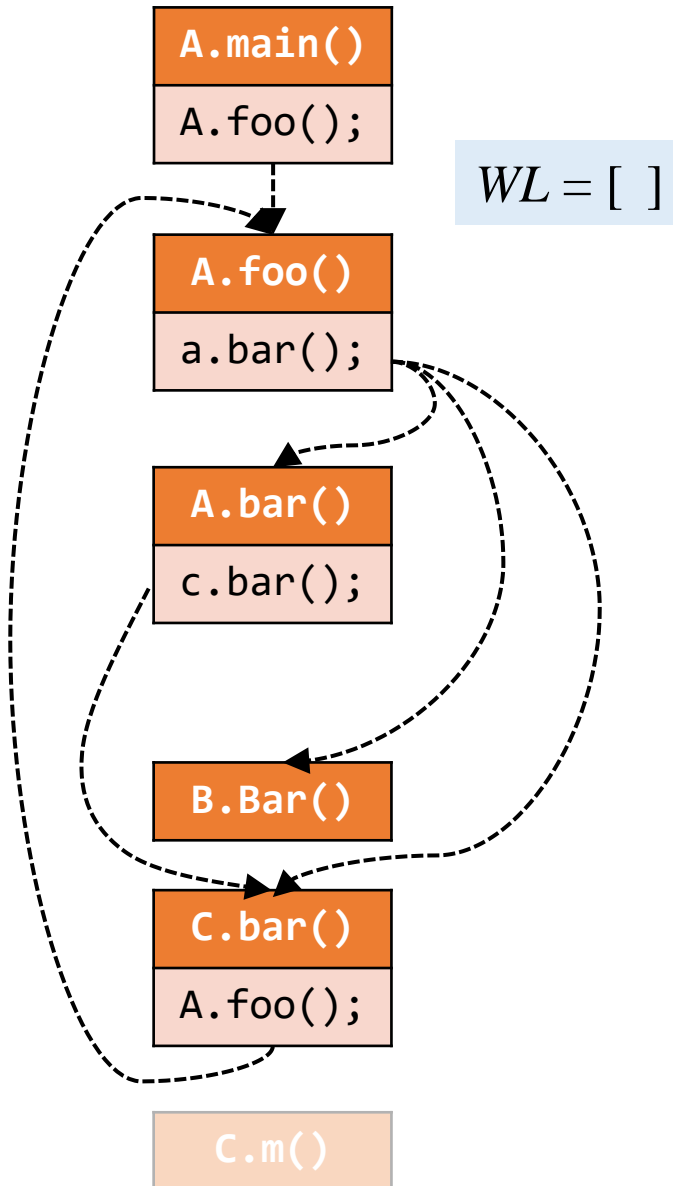
Call Graph Construction: An Example

```
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() {}
}
```



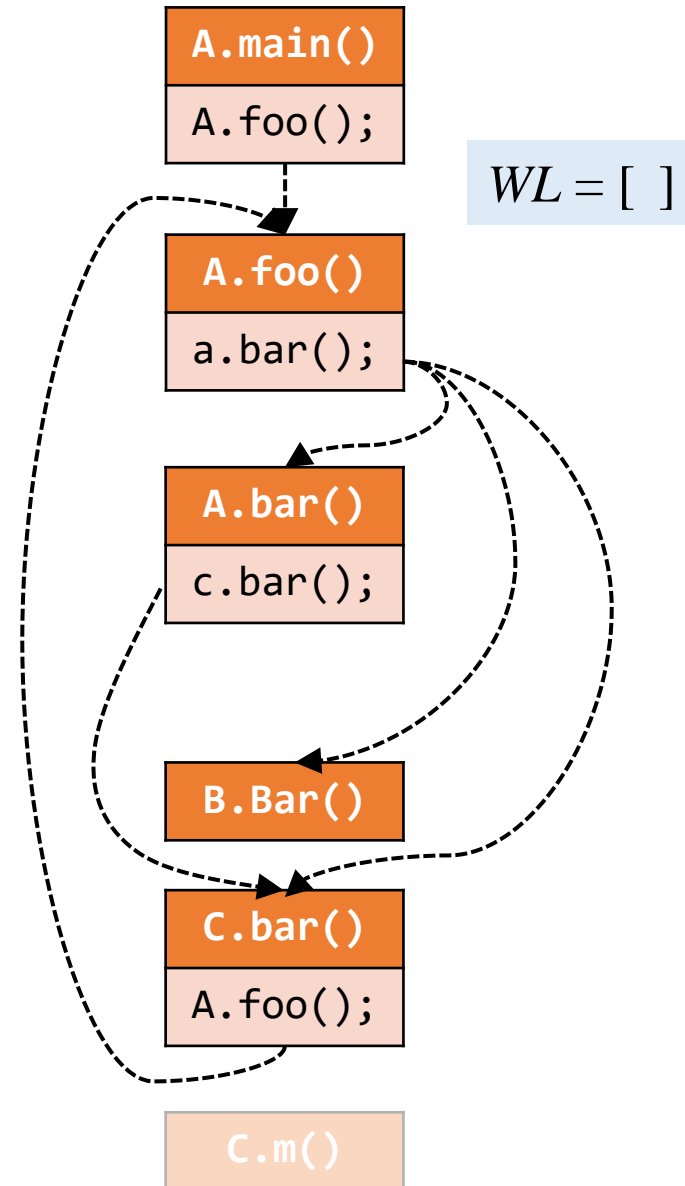
Call Graph Construction: An Example

```
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() {}
}
```



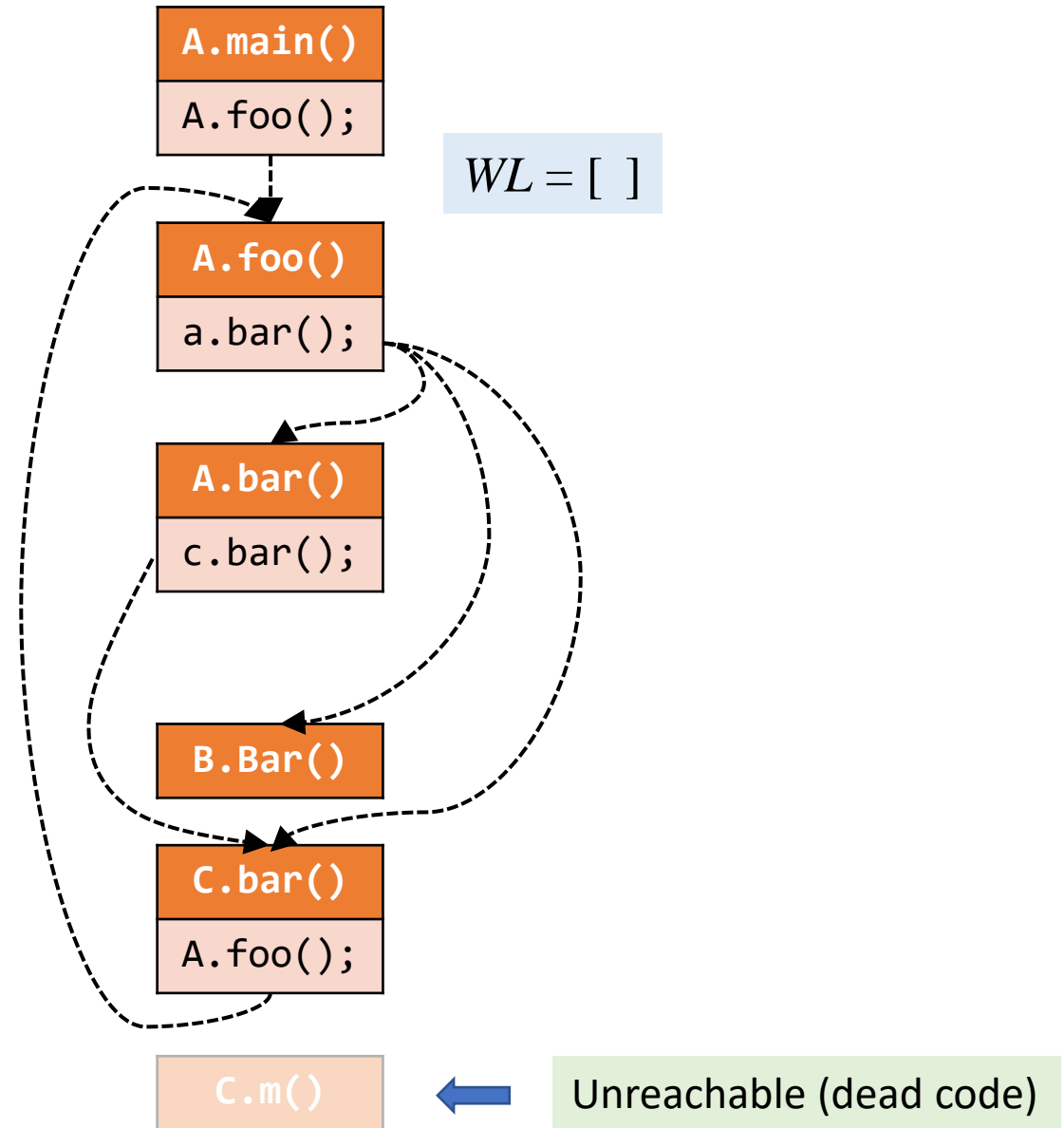
Call Graph Construction: An Example

```
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() {}
}
```



Call Graph Construction: An Example

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



Contents

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

Interprocedural Control-Flow Graph

- CFG represents structure of an individual method
- ICFG represents structure of the whole program
 - With ICFG, we can perform interprocedural analysis

Interprocedural Control-Flow Graph

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

Interprocedural Control-Flow Graph

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

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

Interprocedural Control-Flow Graph

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

ICFG = CFGs + call & return edges

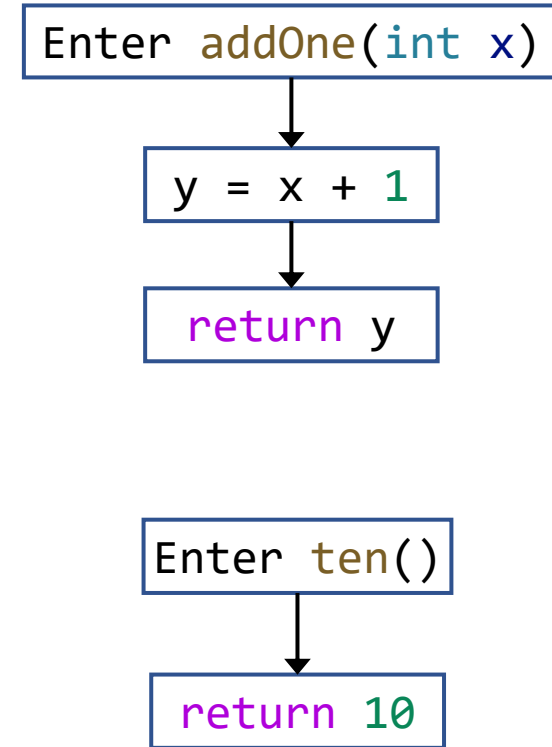
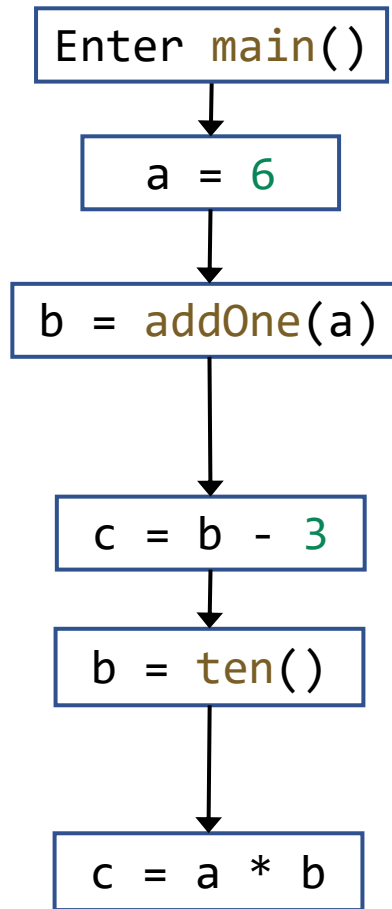
The information for connecting these two kinds of edges comes from **call graph**

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;  
}
```



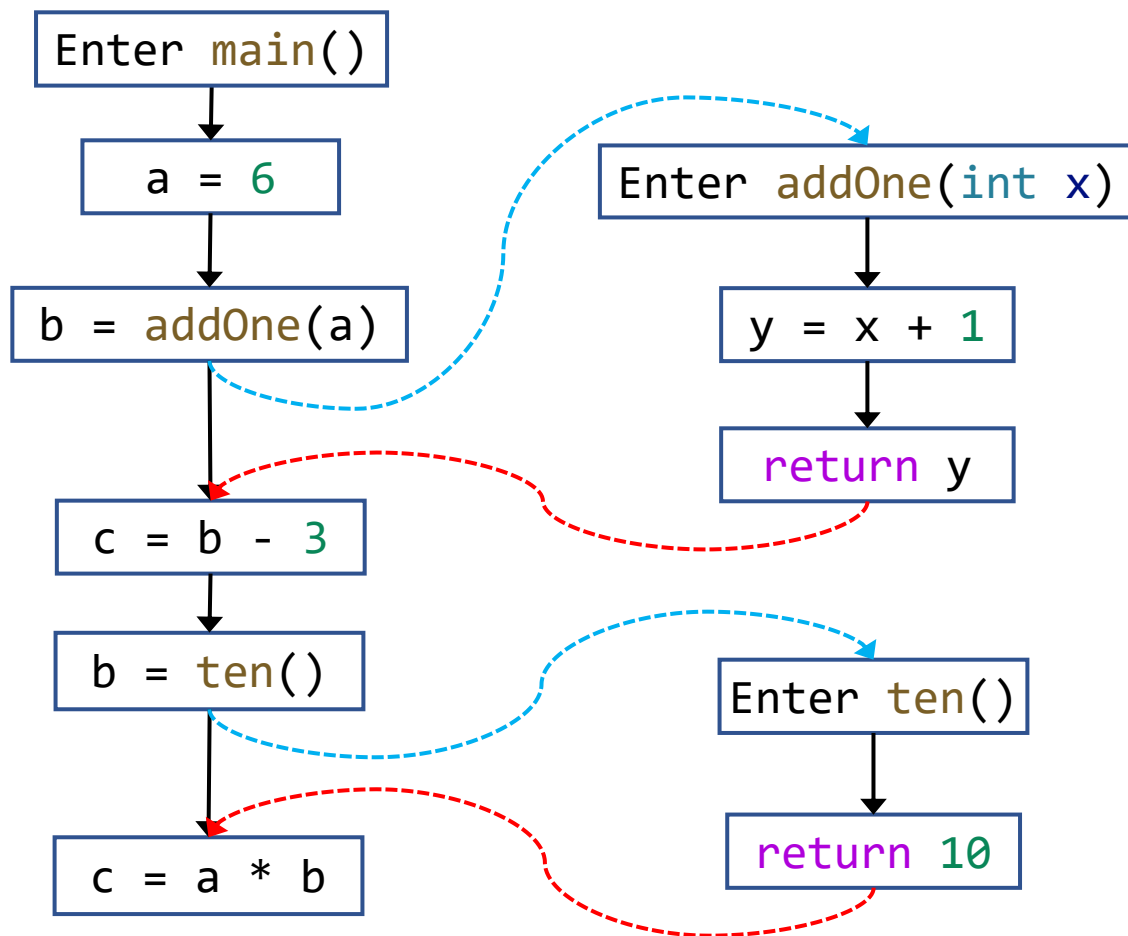
→ CFG edges

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

- CFG edges
- .-> Call edges
- .-> Return edges

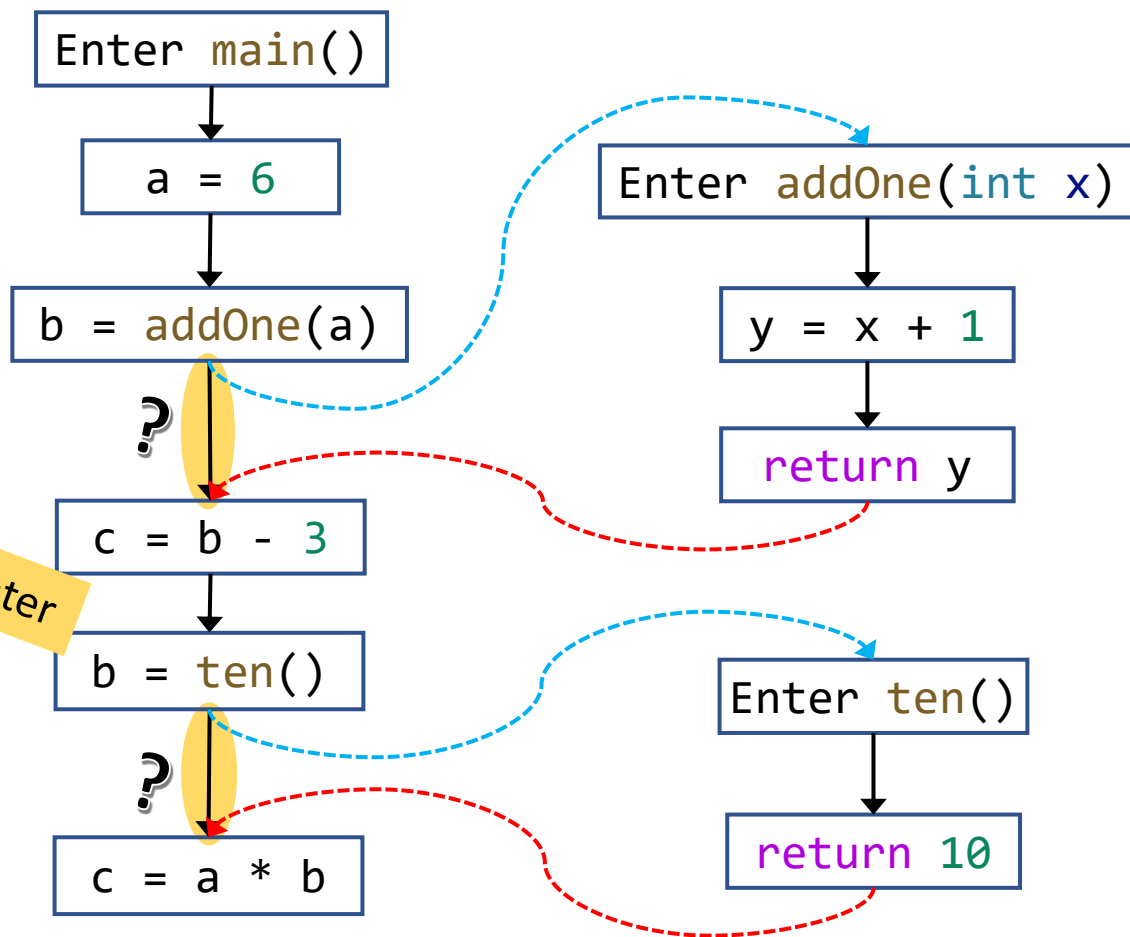
ICFG: An Example

```
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;  
}
```

Will be explained later



ICFG = CFGs + call & return edges

- CFG edges
- - - - - Call edges
- - - - - Return edges

Contents

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

Interprocedural Data-Flow Analysis

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

	<i>Intra</i> procedural	<i>Inter</i> procedural
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)

	<i>Intra</i> procedural	<i>Inter</i> procedural
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)

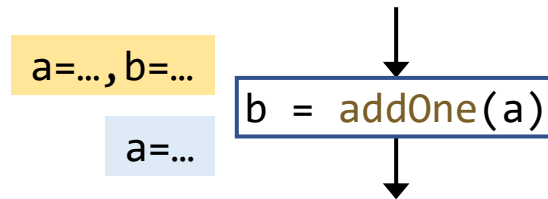
	<i>Intra</i> procedural	<i>Inter</i> procedural
Program representation	CFG	ICFG = CFGs + call & return edges
Transfer functions	Node transfer	Node transfer + edge transfer

Edge transfer

- **Call edge transfer**: transfer data flow from call 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

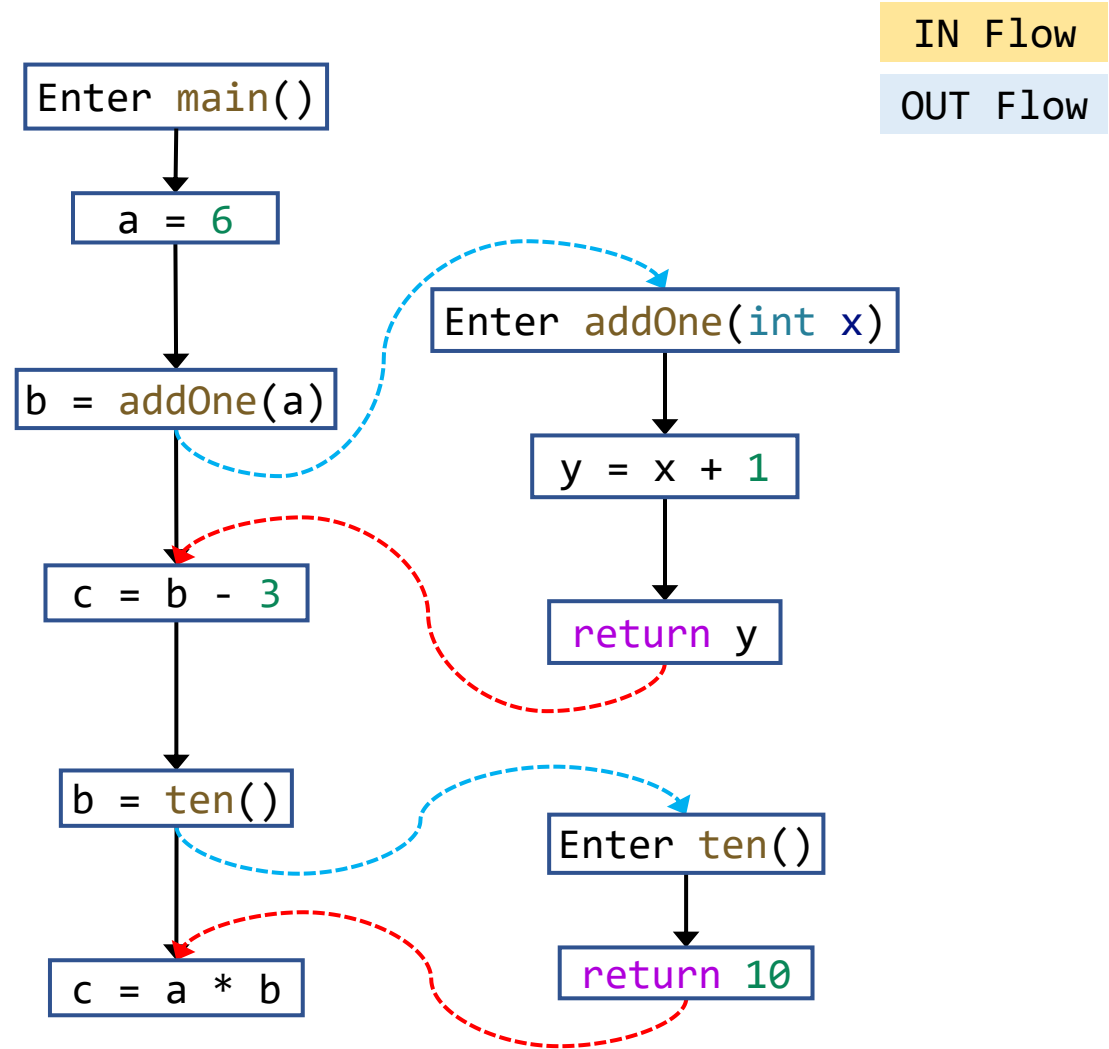


Interprocedural Constant Propagation: An Example

```
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;  
}
```

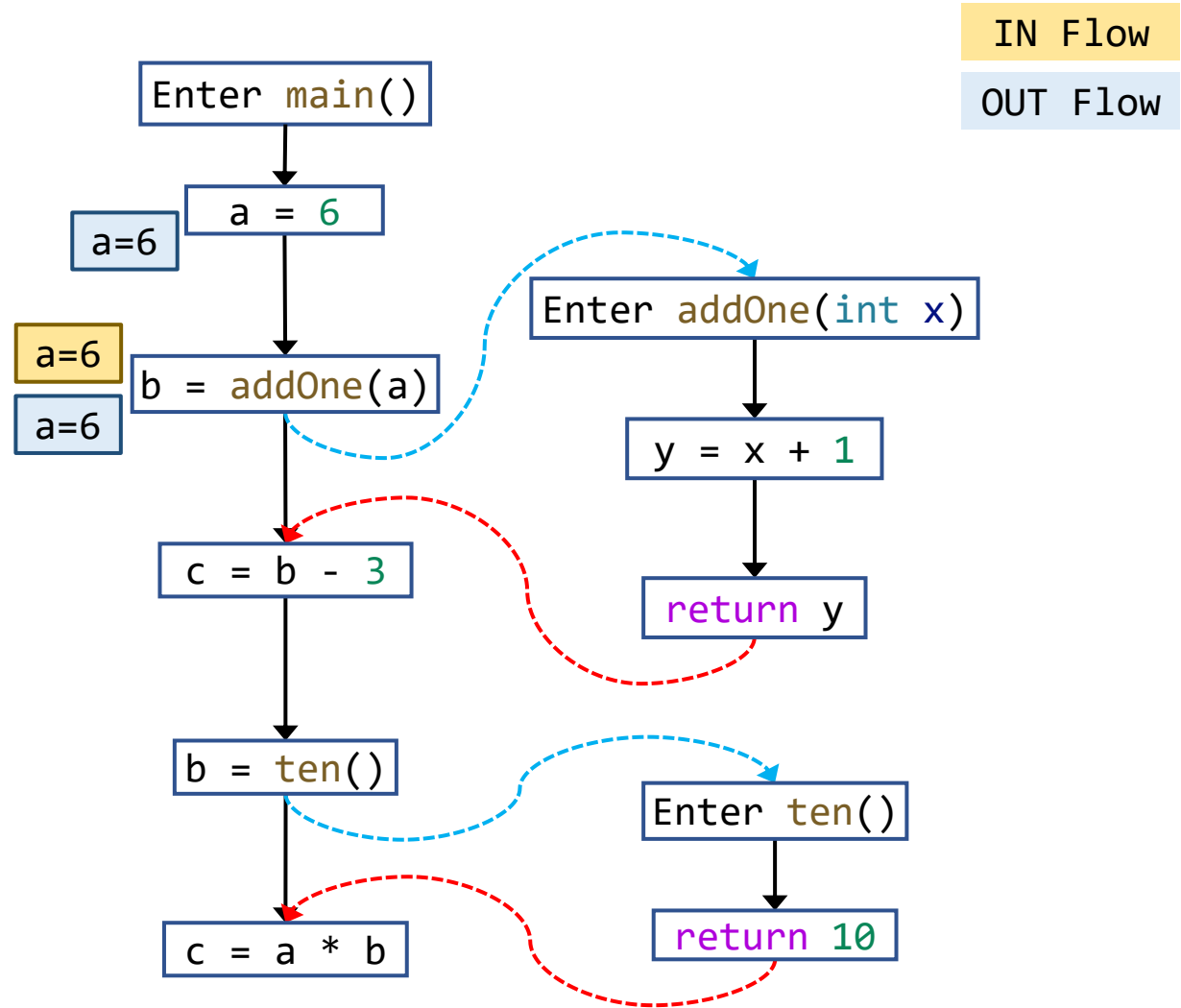


Interprocedural Constant Propagation: An Example

```
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;  
}
```

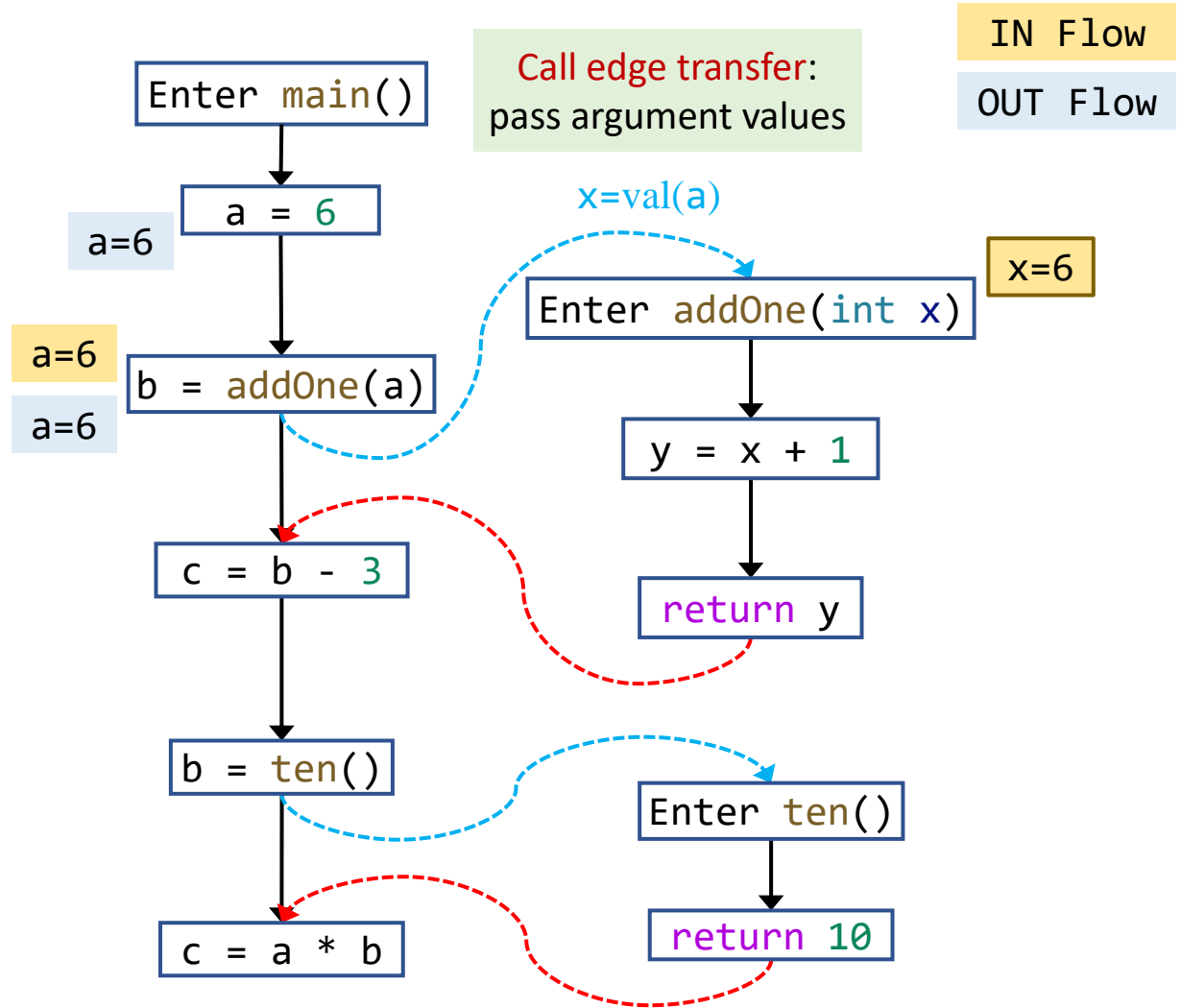


Interprocedural Constant Propagation: An Example

```
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;  
}
```

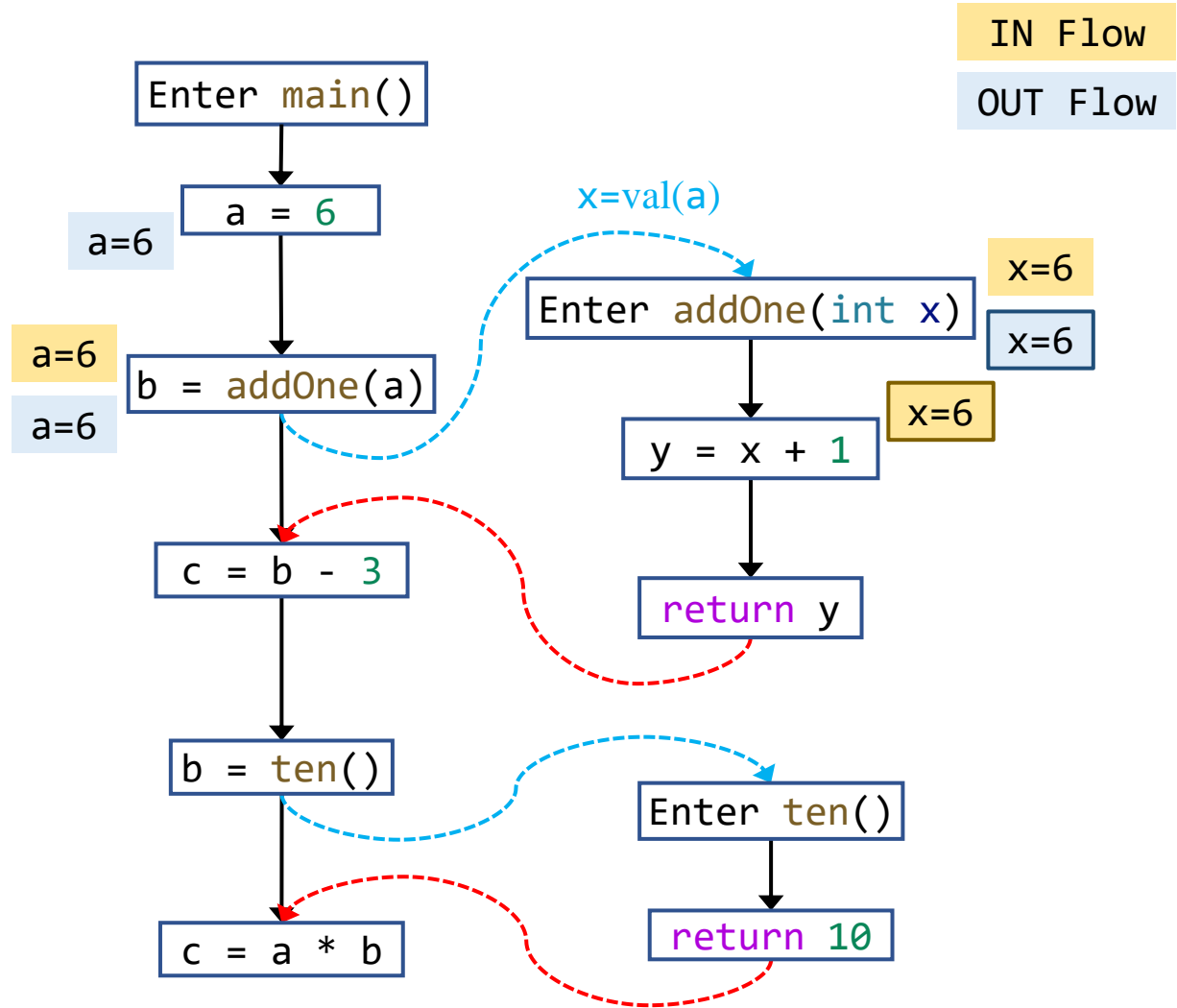


Interprocedural Constant Propagation: An Example

```
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;  
}
```

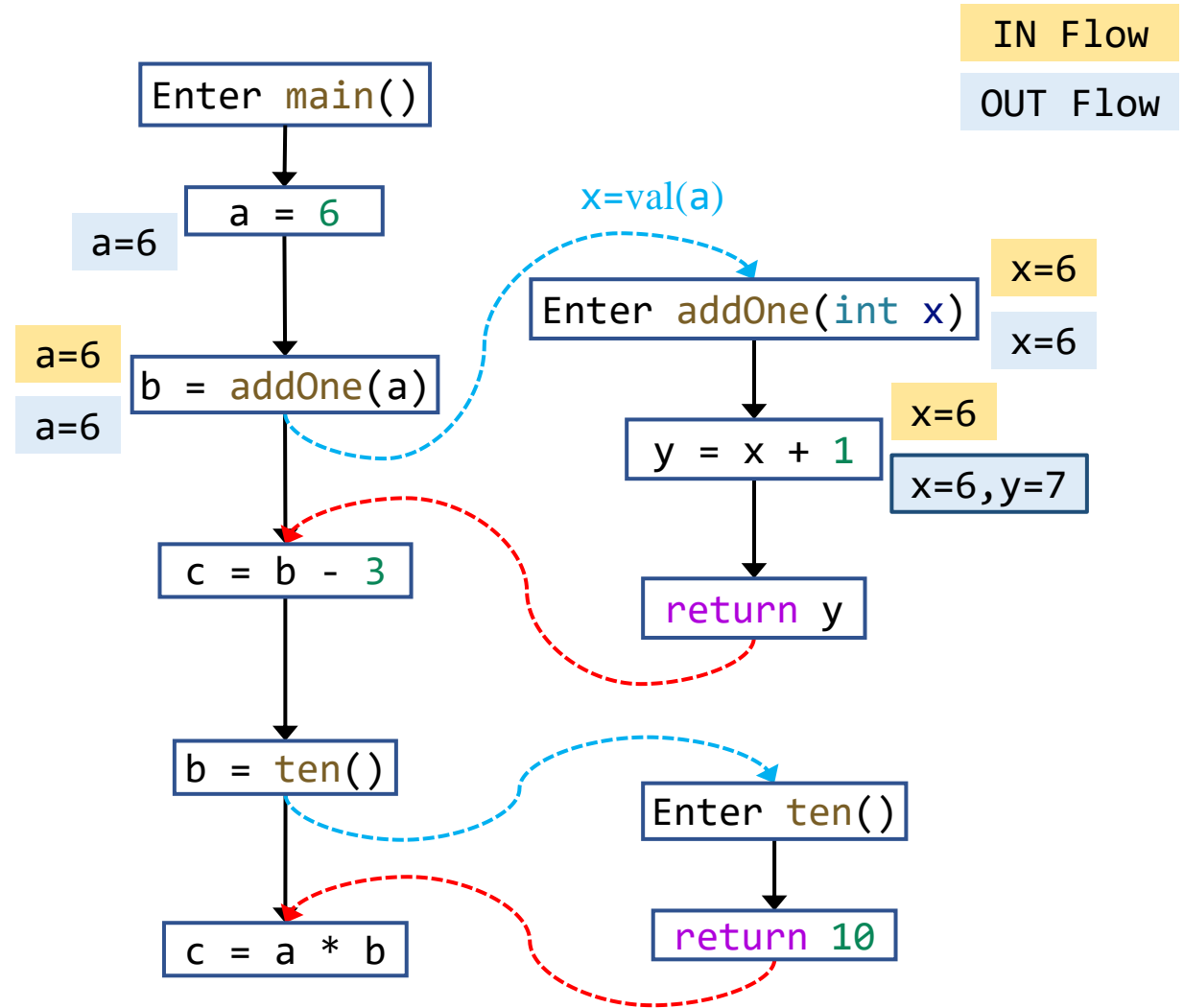


Interprocedural Constant Propagation: An Example

```
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;  
}
```

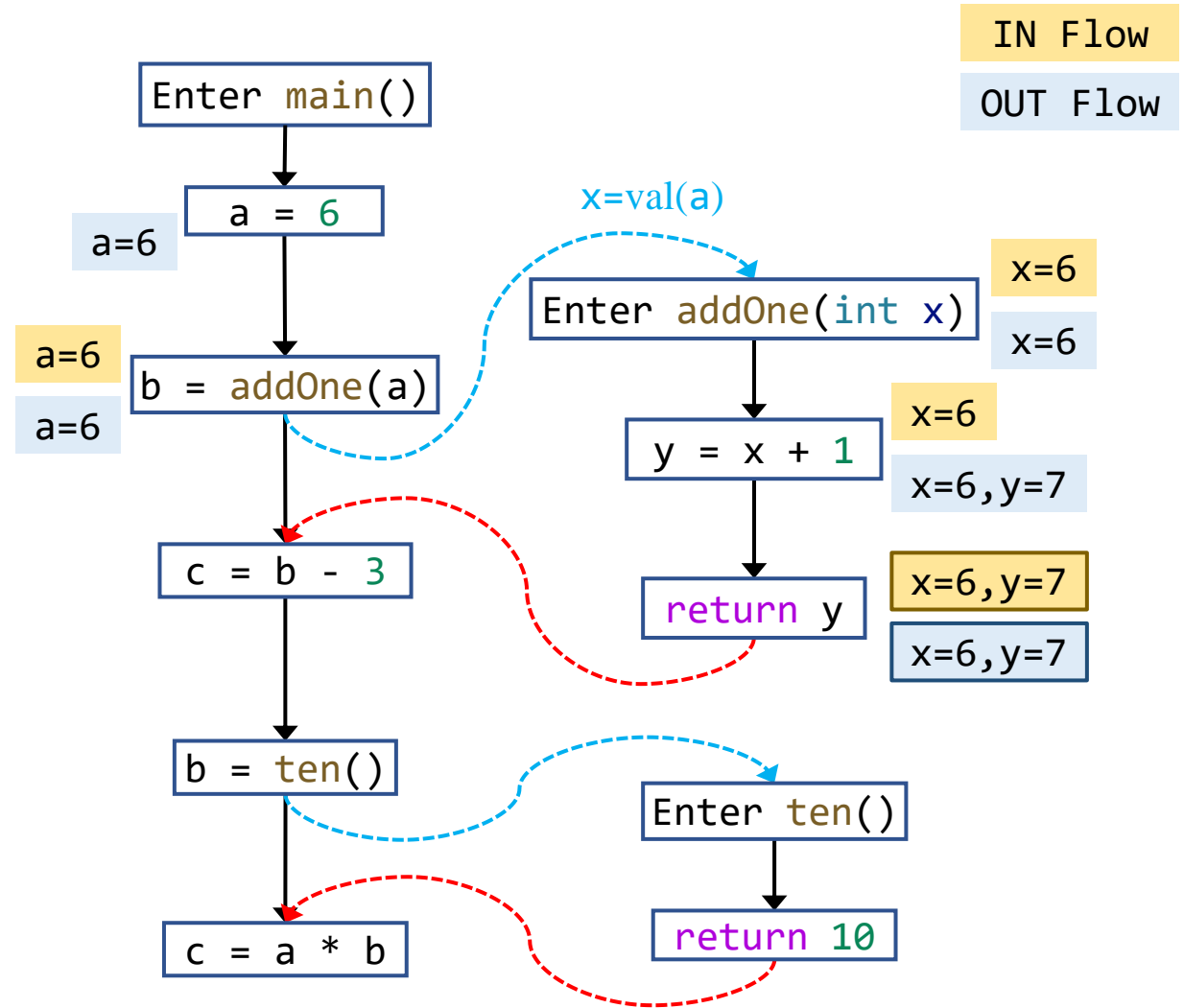


Interprocedural Constant Propagation: An Example

```
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;  
}
```

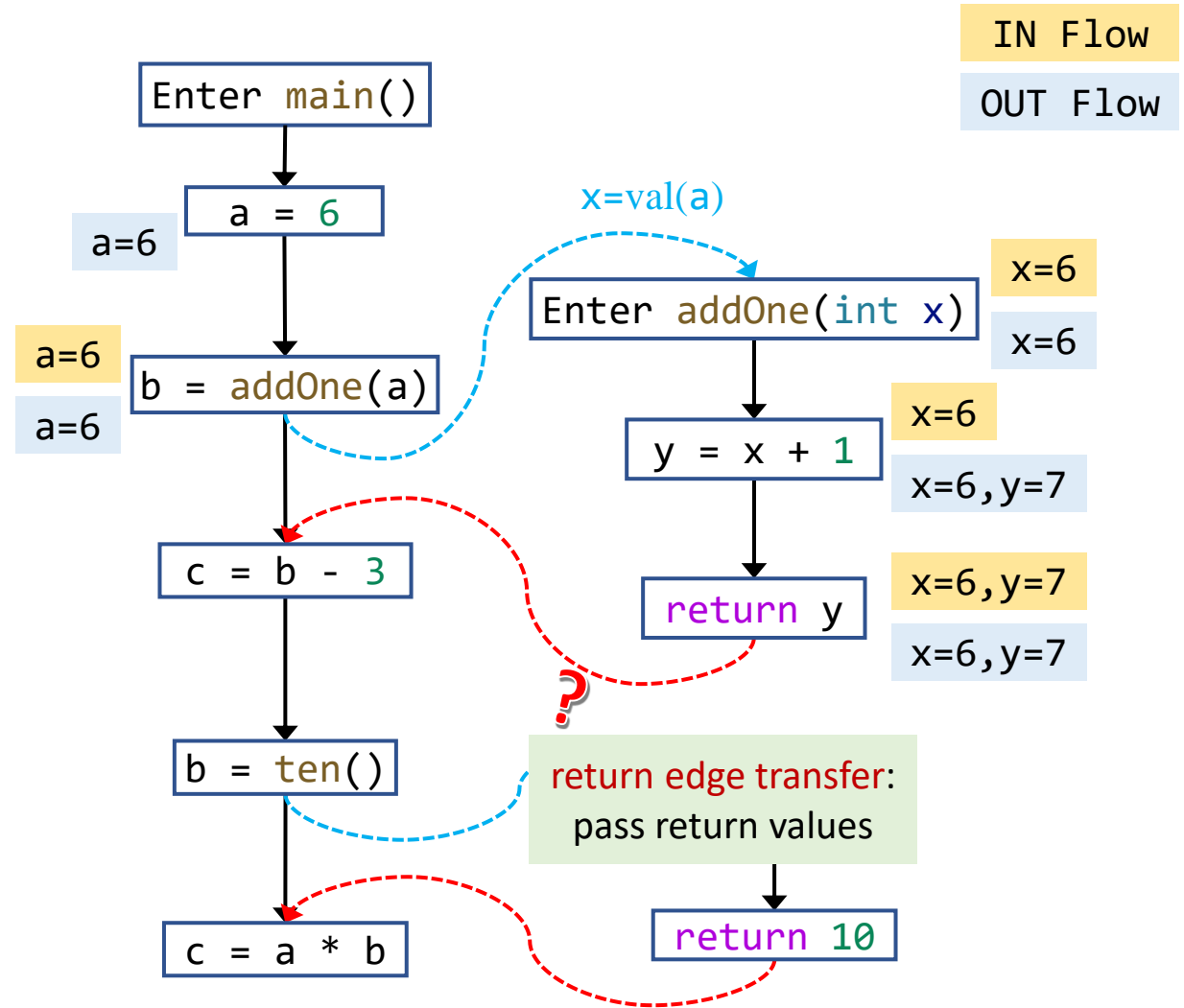


Interprocedural Constant Propagation: An Example

```
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;  
}
```

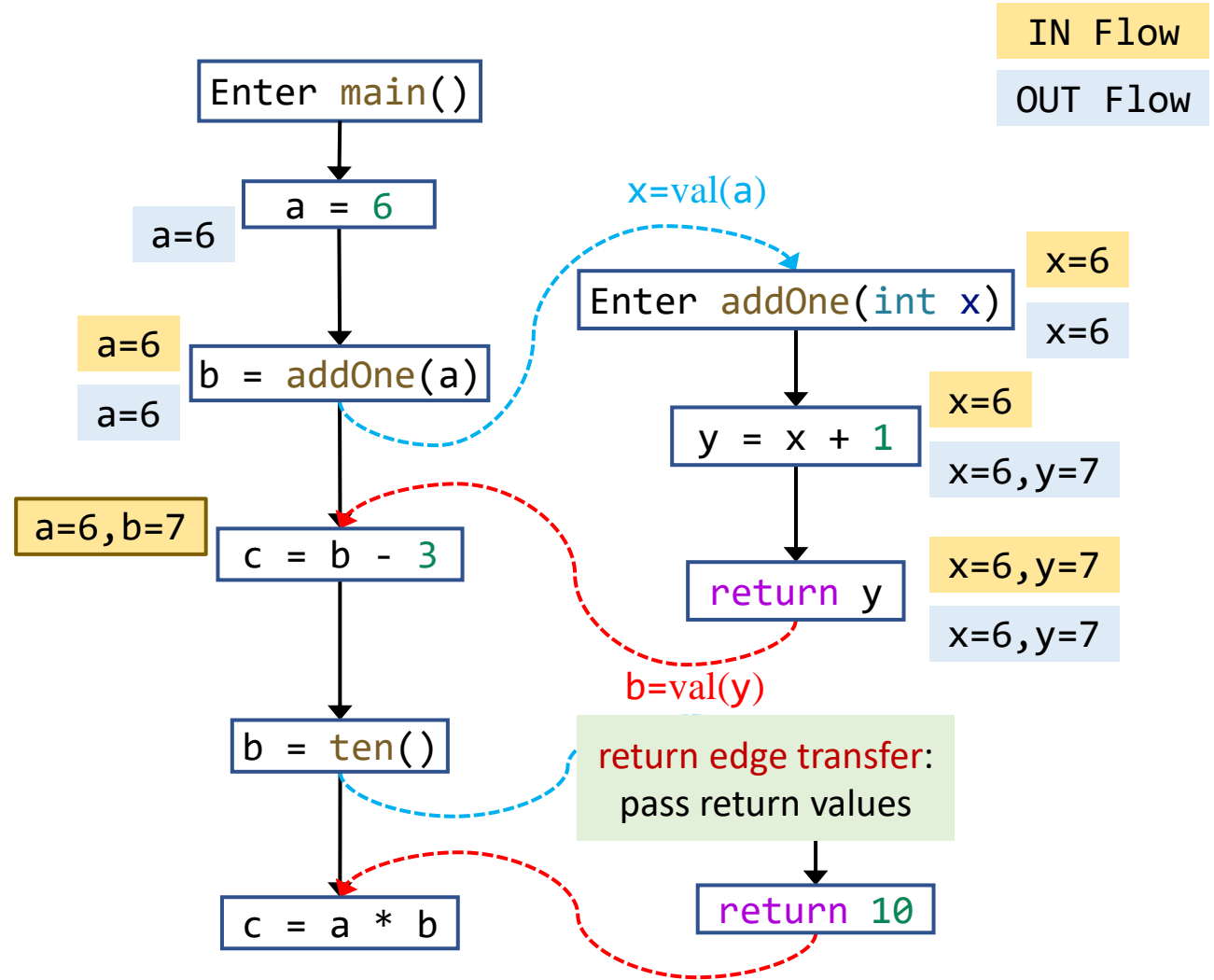


Interprocedural Constant Propagation: An Example

```
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;  
}
```

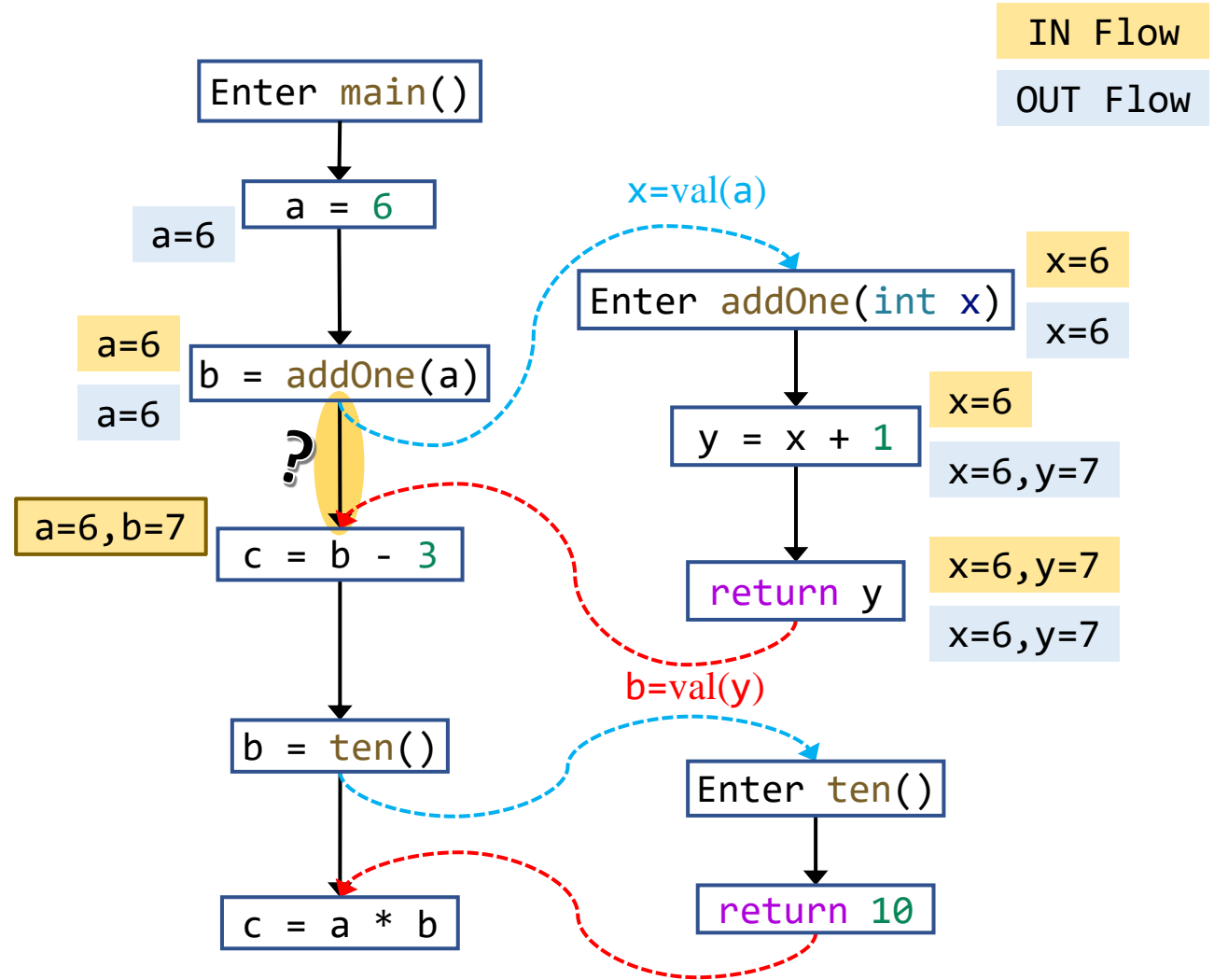


Interprocedural Constant Propagation: An Example

```
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;  
}
```

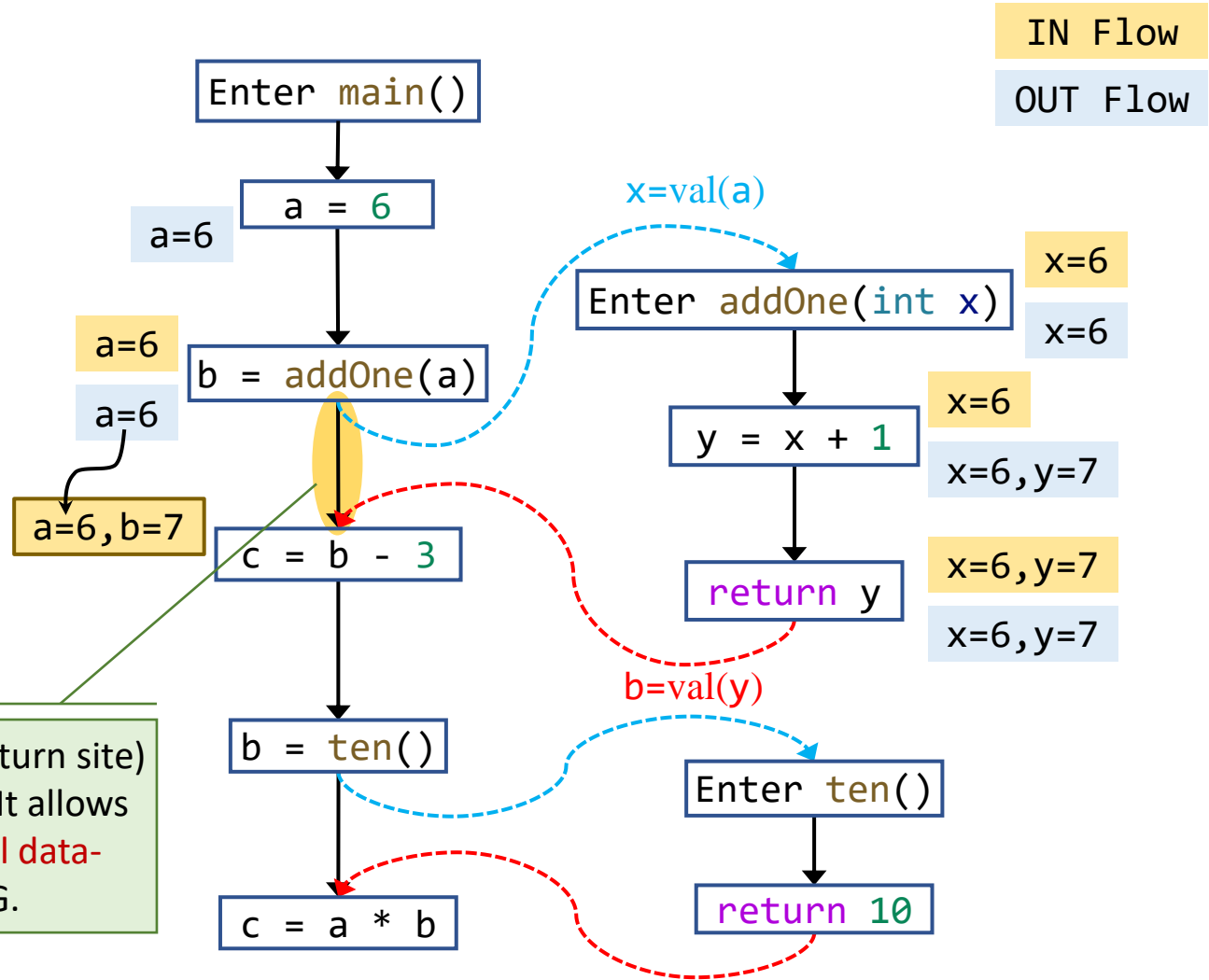


Interprocedural Constant Propagation: An Example

```
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;  
}
```

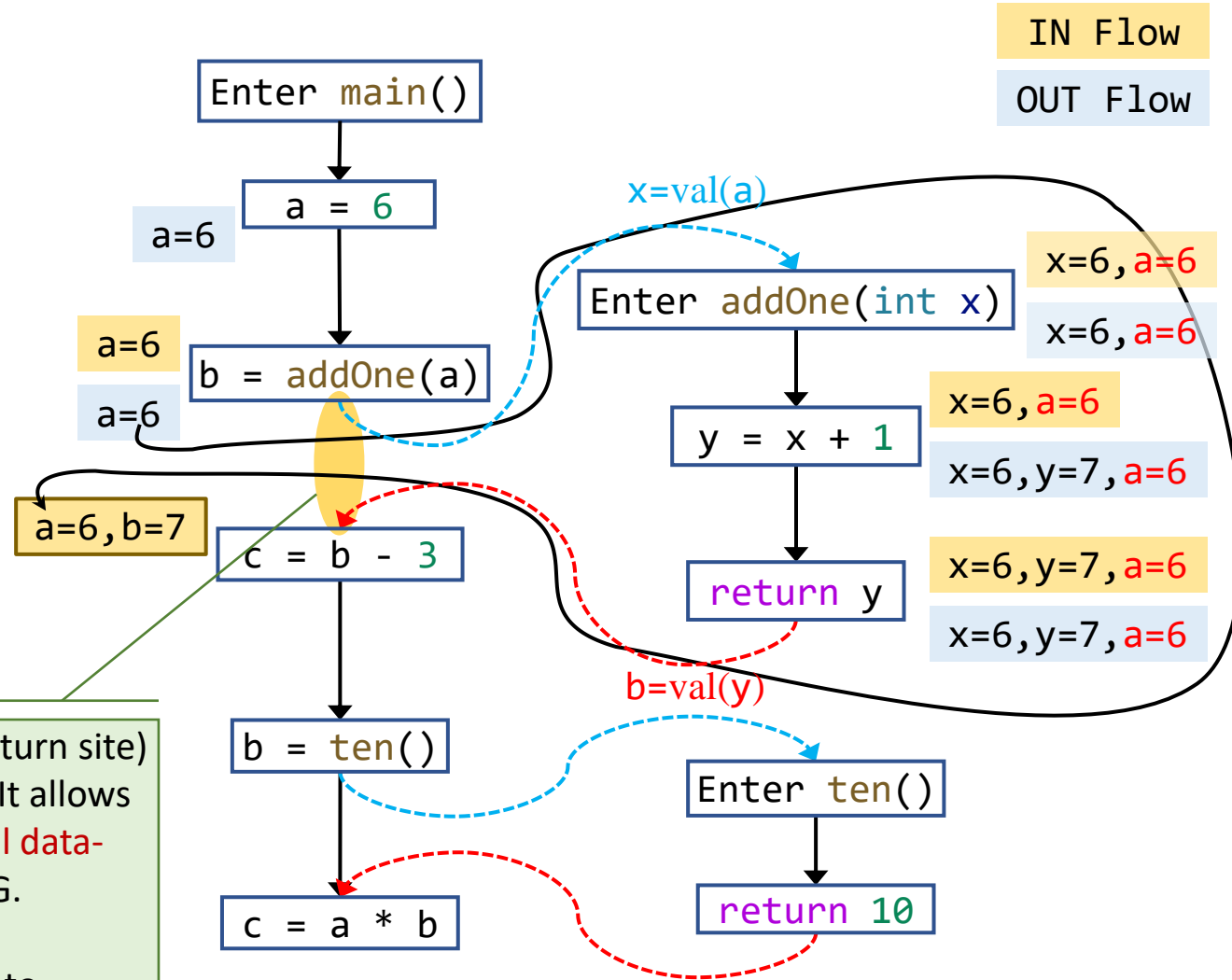
Such edge (from call site to return site) is named *call-to-return edge*. It allows the analysis to propagate **local data-flow** (a=6 in this case) on ICFG.



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

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

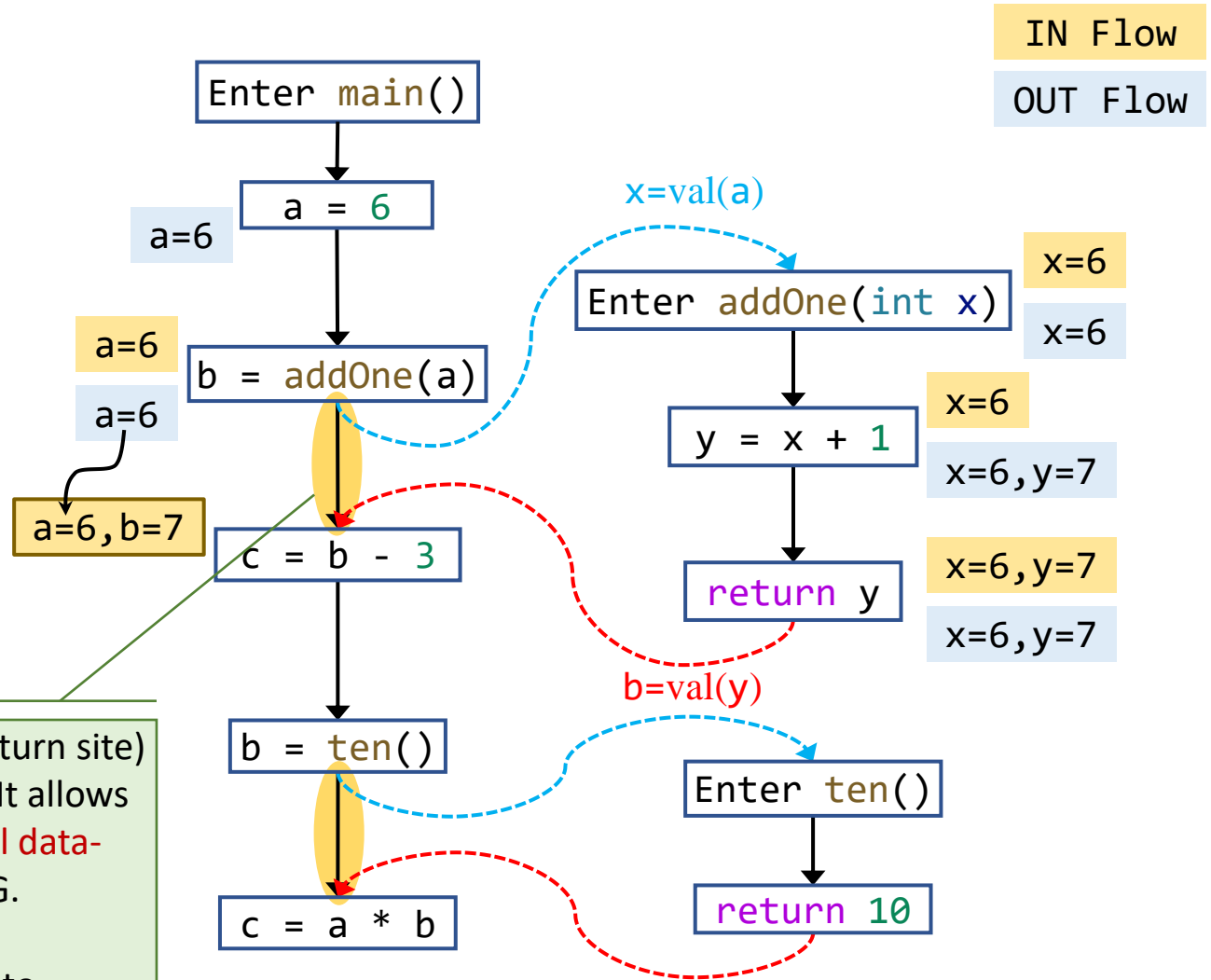
Interprocedural Constant Propagation: An Example

```
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**.

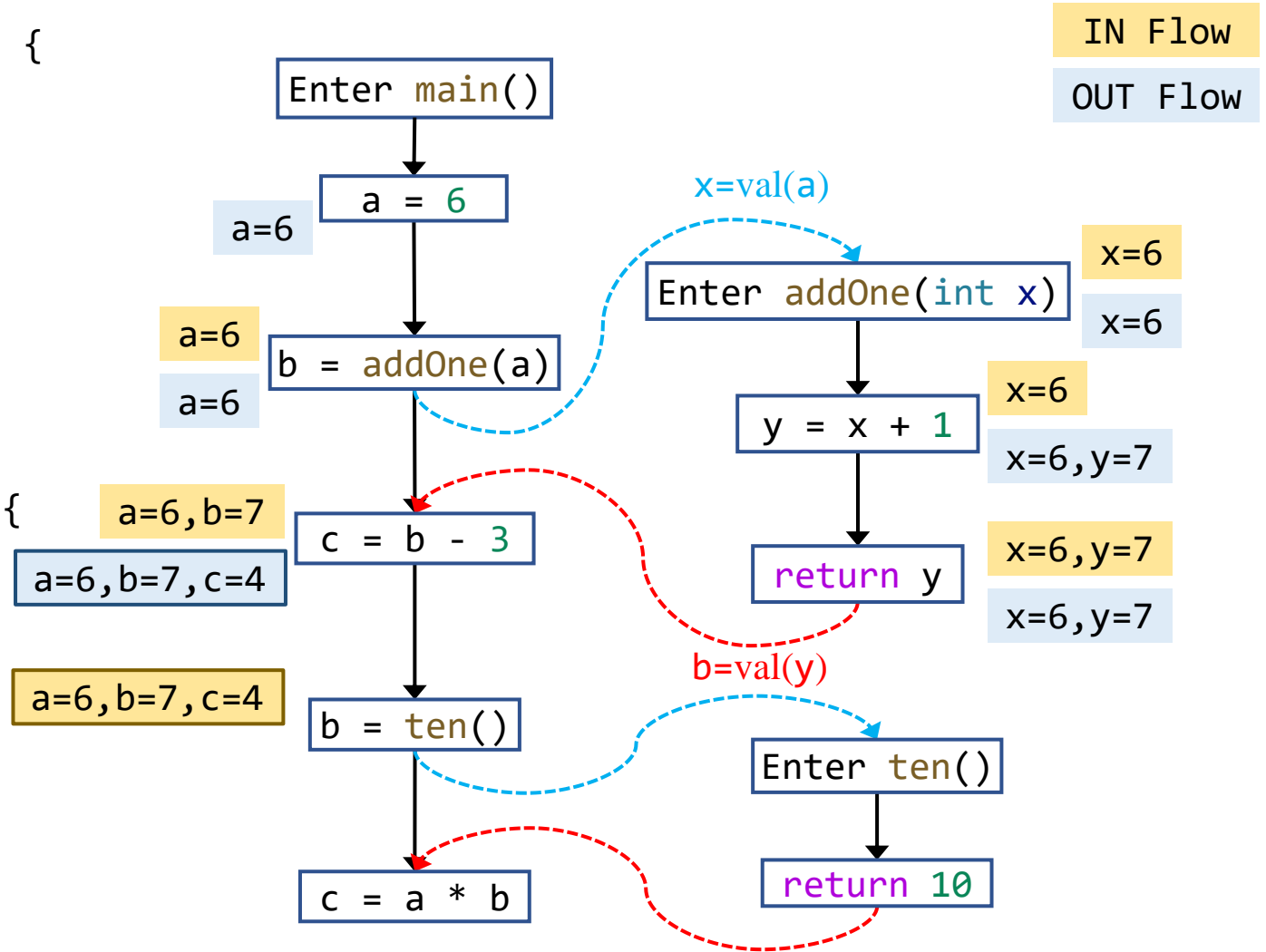


Interprocedural Constant Propagation: An Example

```
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;
}
```

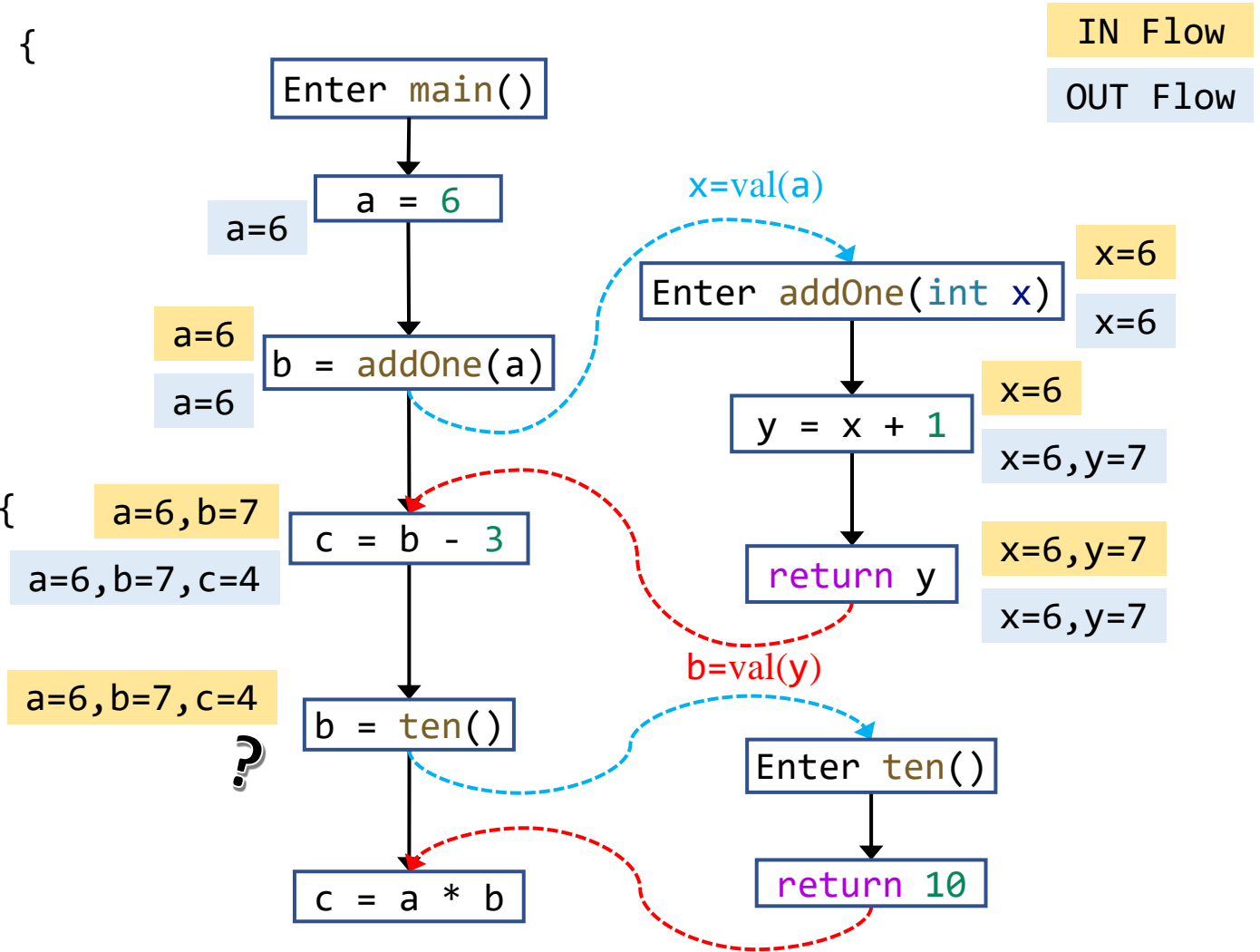


Interprocedural Constant Propagation: An Example

```
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;
}
```

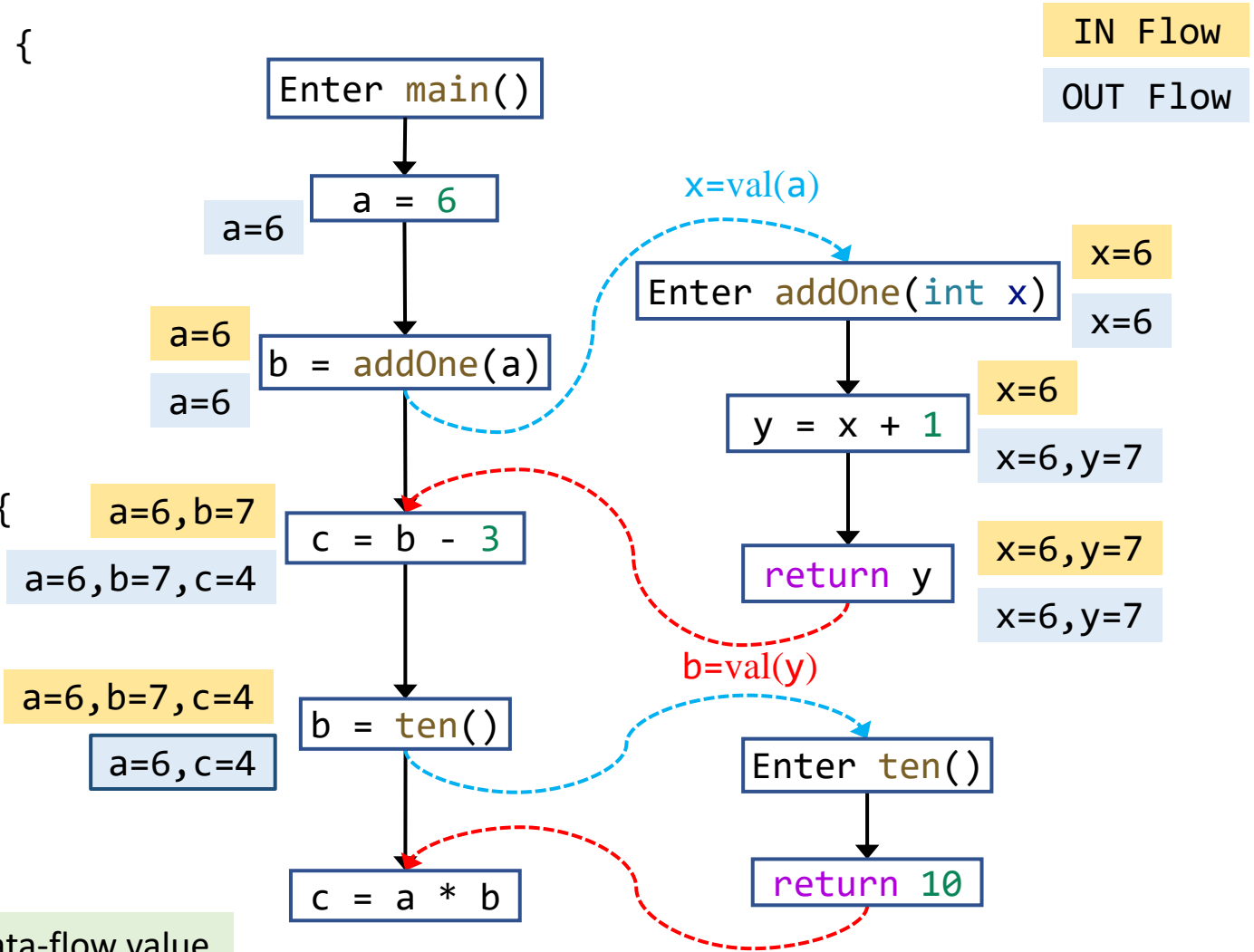


Interprocedural Constant Propagation: An Example

```
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;
}
```



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

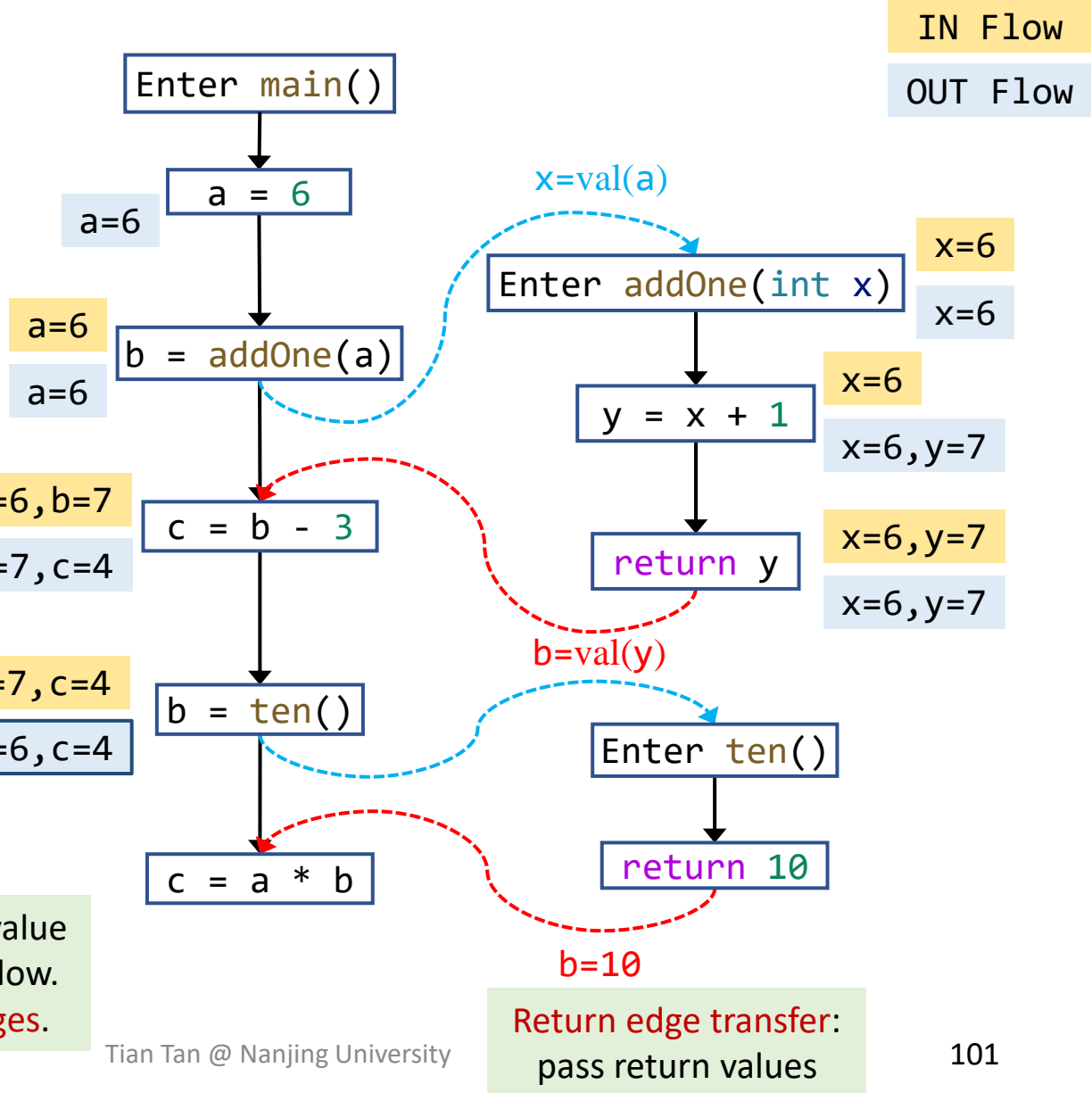
Interprocedural Constant Propagation: An Example

```
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**.

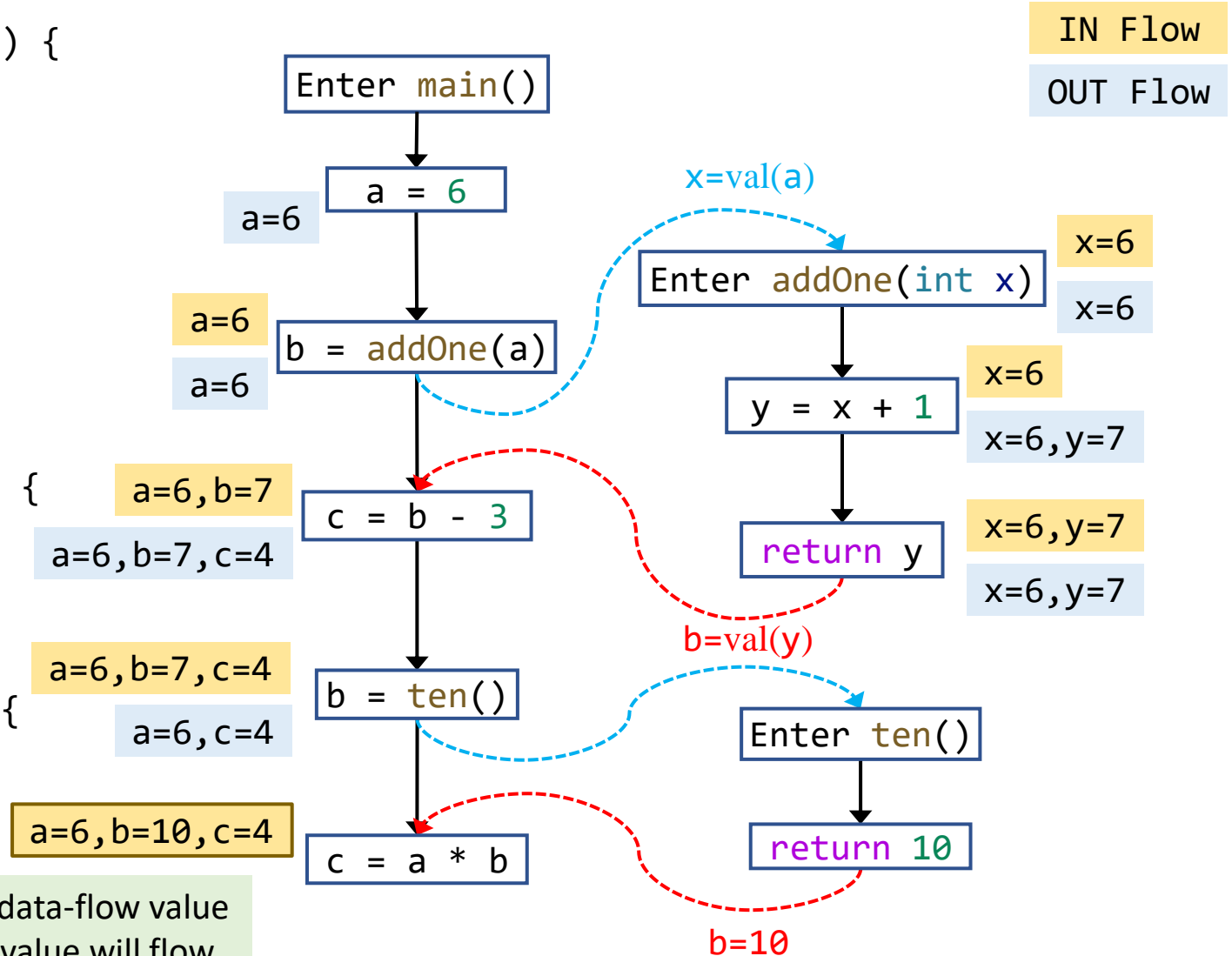


Interprocedural Constant Propagation: An Example

```
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.

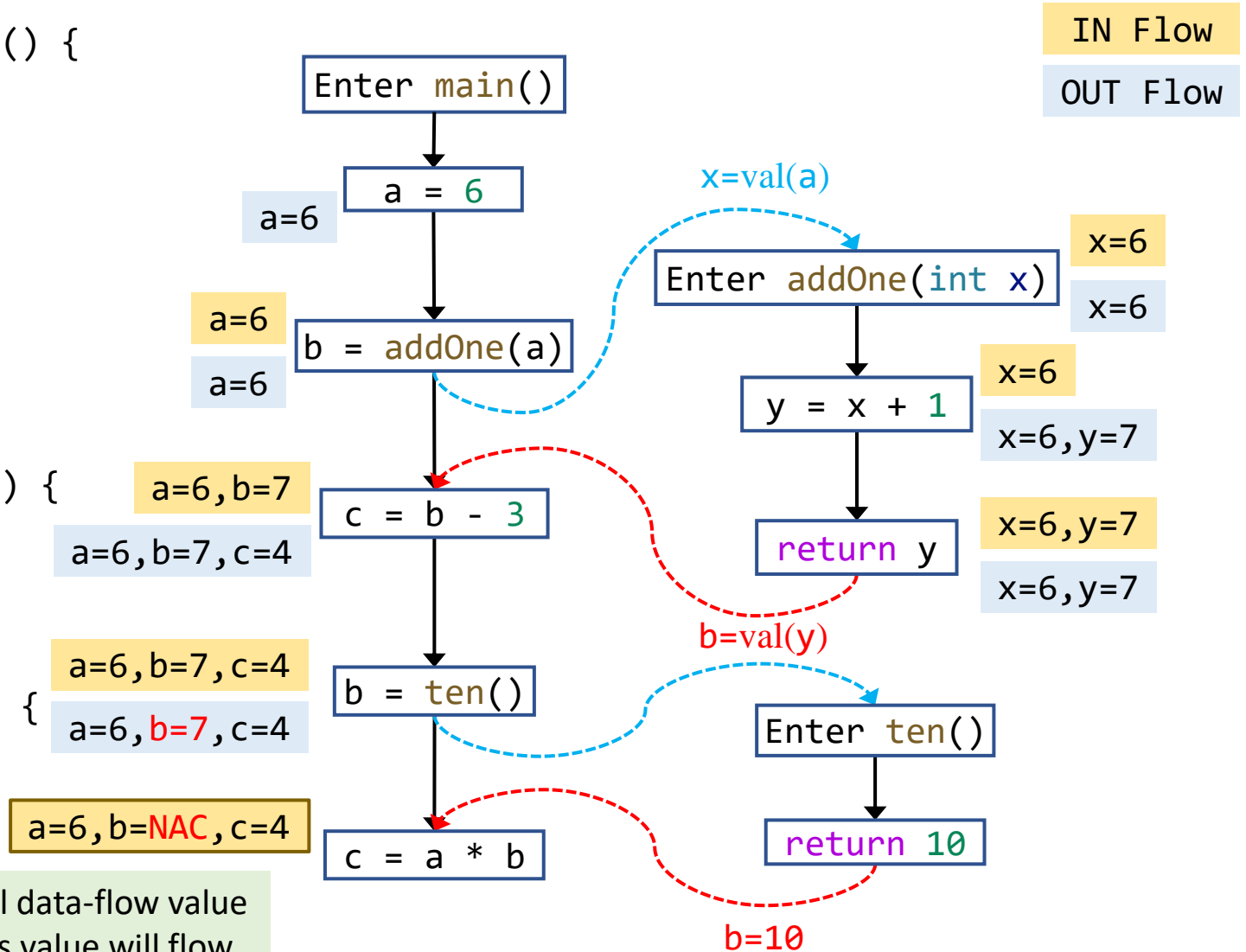
Return edge transfer: pass return values

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;
}
```



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**.

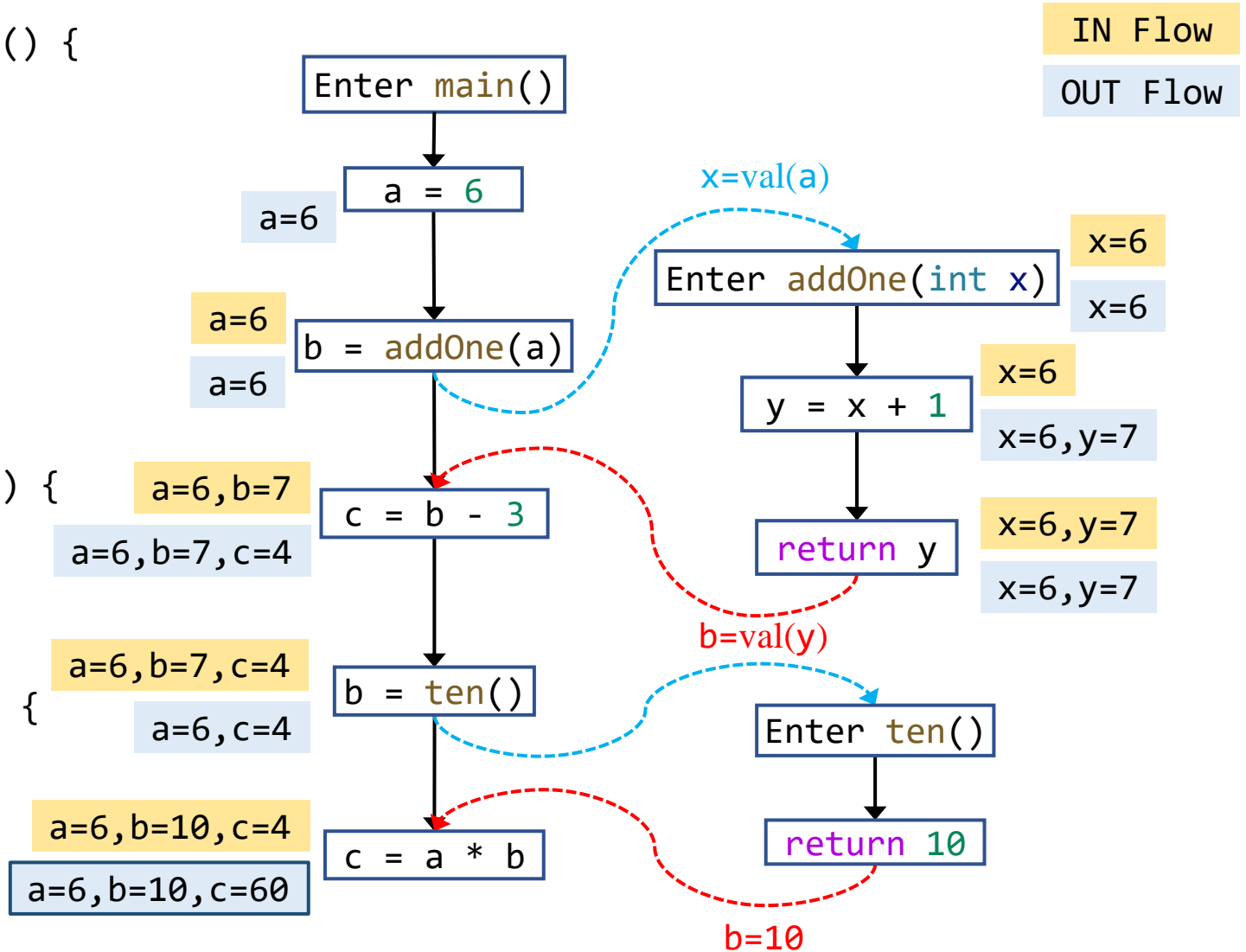
Return edge transfer: pass return values

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;
}
```

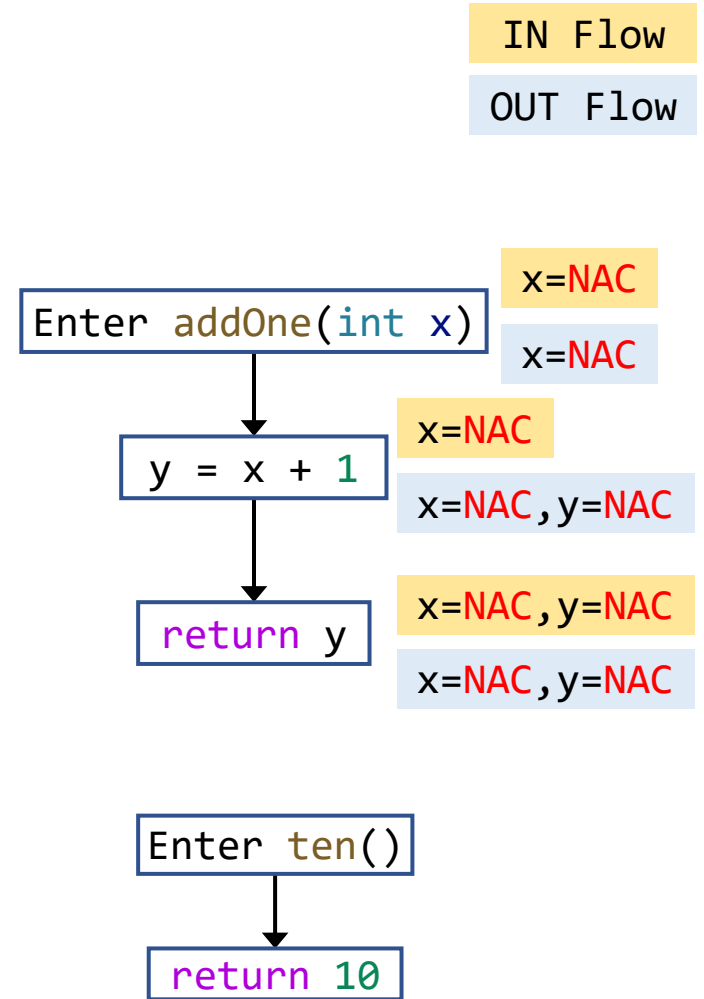
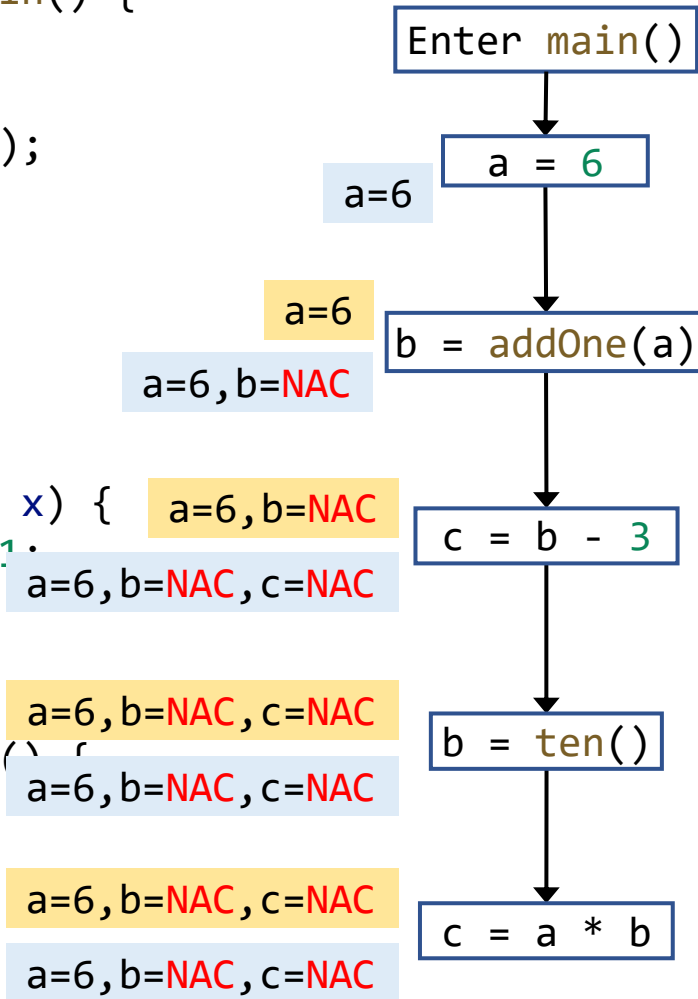


Intraprocedural Constant Propagation: An Example

```
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;
}
```

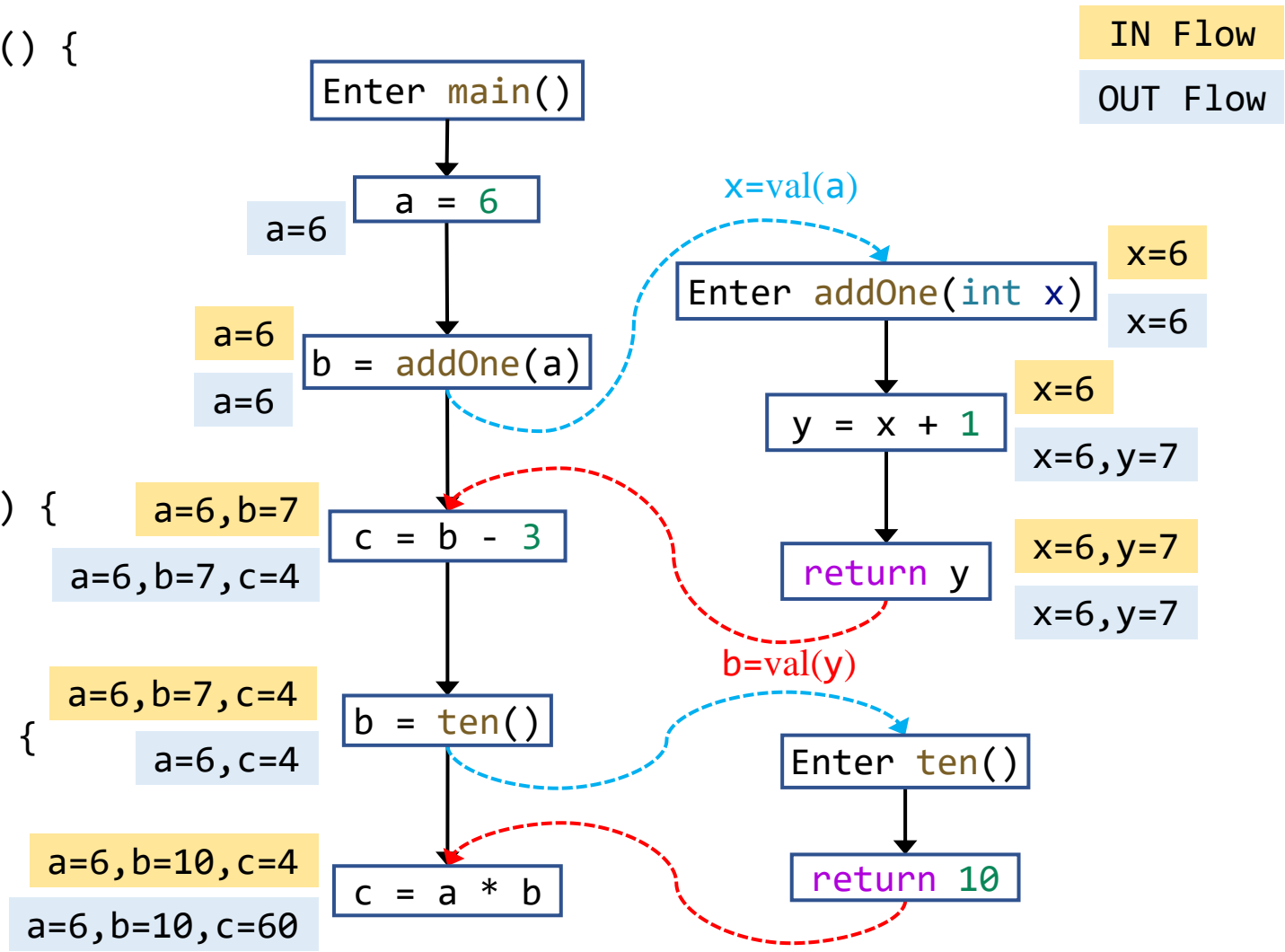


Interprocedural Constant Propagation: An Example

```
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

Interprocedural constant propagation is **more precise** than **Intraprocedural** constant propagation

The X You Need To Understand in This Lecture

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

注意注意!
划重点了!



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

李棣
谭添