

# Introduction to Program Analysis

## 9. Static Analysis for Advanced Programming Features

Kihong Heo



# Advanced Programming Features

- So far, we have learned **design and implementation** of static analysis
  - Domains, semantics, abstractions, algorithms
  - For basic programming features: assignments, conditionals, loops
- From now, we extend SmaLLVM to support more **advanced features**
  - In this lecture, pointers with dynamic allocations, and functions

# The ThriLLVM Language

- Extension 1: Pointers

$L ::= l$	program label
$E ::= n$	integer
$x$	variable
$E \oplus E$	arithmetic operation
$E \otimes E$	comparison operation
$\&x$	location of a variable
$\text{malloc}$	location of a newly allocated memory
$*E$	dereference of a memory location
$\oplus ::= +   -   \times   /$	
$\otimes ::= <   \leq   >   \geq   ==   !=$	
$C ::= x := E$	assignment
$\text{br } E L L$	conditional jump
$\text{goto } L$	unconditional jump
$x := \text{input}()$	input
$\text{print}(x)$	print
$*E := E$	indirect assignment



# Example

x = malloc;	$\{x \mapsto a\}$
y = &x;	$\{x \mapsto a, y \mapsto x\}$
z = x;	$\{x \mapsto a, y \mapsto x, z \mapsto a\}$
*x = 5;	$\{x \mapsto a, y \mapsto x, z \mapsto a, a \mapsto 5\}$
*y = *x;	$\{x \mapsto 5, y \mapsto x, z \mapsto a, a \mapsto 5\}$

Memory addresses (variables or heap addresses)  
also can be values

# Concrete Domains

- Similar as before except for the followings:
  - A memory location is either a variable or a heap address
  - A value is either an integer or a memory location

$S = L \times M$	states
$M = A \rightarrow V$	memories
$A = X \cup H$	addresses (locations)
$V = Z \cup A$	values
$X$	set of variables
$H$	set of allocated heap addresses
$L$	set of statement labels

# Domain of Heap Addresses

$$\mathbb{H} = \mathbb{N}_{site} \times \mathbb{N}$$

- A heap address is represented as a pair of an allocation-site and a counter
  - Given a program, every malloc expression is assumed to have a unique number  $\mu$ , writing `malloc $\mu$`
  - For example, the addresses of fresh locations from a malloc-site  $\mu$  are  $(\mu, 0), (\mu, 1), (\mu, 2), \dots$

```
while(*) {  
   $\mu$ : p = malloc(size);  
}
```

# State Transition

- The semantics is specified by a transition system  $(\mathbb{S}, \rightarrow)$ 
  - $\mathbb{S} = \mathbb{L} \times \mathbb{M}$  : the set of states  $\langle l, m \rangle$
  - $(\rightarrow) \subseteq \mathbb{S} \times \mathbb{S}$  : the transition relation that describes computation steps
- New rule: the transition for indirect assignments
$$*E_1 := E_2 : \langle l, m \rangle \rightarrow \langle \text{next}(l), \text{update}(m, \text{eval}_{E_1}(m), \text{eval}_{E_2}(m)) \rangle$$

# Semantic Operators

- The memory read/write operations

$$fetch : \mathbb{M} \times \mathbb{V} \rightarrow \mathbb{V}$$

$$fetch(m, v) = m(v)$$

$$update : \mathbb{M} \times \mathbb{V} \times \mathbb{V} \rightarrow \mathbb{M}$$

$$update(m, v_1, v_2) = m\{v_1 \mapsto v_2\}$$

- The expression-evaluation operation

$$eval_E : \mathbb{M} \rightarrow \mathbb{V}$$

$$eval_x(m) = fetch(m, x)$$

$$eval_{\&x}(m) = x$$

variable as a location &x

$$eval_{\text{malloc}}_\mu(m) = (\mu, z)$$

new number z for malloc site  $\mu$

$$eval_{*E}(m) = fetch(m, eval_E(m))$$

dereference of a location

- The memory filter operation

$$filter_E : \mathbb{M} \rightarrow \mathbb{M}$$

$$filter_E(m) = m \quad \text{if } eval_E(m) = \text{true}$$

# Concrete Semantics

- Concrete domain:  $\mathbb{D} = \wp(\mathbb{S})$  where  $\mathbb{S} = \mathbb{L} \times \mathbb{M}$
- Concrete semantic function:

$$F : \wp(\mathbb{S}) \rightarrow \wp(\mathbb{S})$$

$$F(X) = I \cup Step(X)$$

where  $I$  is the set of initial states and  $Step$  is the powerset-lifted version of  $\hookrightarrow$

$$Step : \wp(\mathbb{S}) \rightarrow \wp(\mathbb{S})$$

$$Step(X) = \{s' \mid s \hookrightarrow s', s \in X\}$$

- Concrete semantic (i.e., reachable states):  $\text{lfp } F$

# Abstract Domains

- Point-wise abstractions of underlying abstractions (addresses and values)

$S^\#$	$=$	$\mathbb{L} \times M^\#$	abstract states
$M^\#$	$=$	$(X \cup H^\#) \rightarrow V^\#$	abstract memories
$A^\#$	$=$	$\wp(X \cup H^\#)$	abstract addresses (locations)
$V^\#$	$=$	$Z^\# \times A^\#$	abstract values
$X$			set of variables
$H^\#$			set of abstract heap addresses
$L$			set of statement labels

# Abstraction of Values

- Kind-wise abstractions of a set of values (integers and locations)

$$\wp(\mathbb{V} = \mathbb{Z} \cup \mathbb{A}) \xrightleftharpoons[\alpha_{\mathbb{V}}]{\gamma_{\mathbb{V}}} \mathbb{V}^{\sharp} = \mathbb{Z}^{\sharp} \times \mathbb{A}^{\sharp}$$

$$\alpha_{\mathbb{V}}(V) = (\alpha_{\mathbb{Z}}(V \cap \mathbb{Z}), \alpha_{\mathbb{A}}(V \cap \mathbb{A}))$$

$$\gamma_{\mathbb{V}}(z^{\sharp}, a^{\sharp}) = \gamma_{\mathbb{Z}}(z^{\sharp}) \cup \gamma_{\mathbb{A}}(a^{\sharp})$$

- All the operators (order, join, meet, widening, narrowing, etc) are defined as kind-wise liftings

# Abstraction of Locations

- Abstract all heap locations allocated at a malloc site  $\mu$  into a single abstract address (so called allocation-site-based abstraction)

$$\wp(\mathbb{H} = \mathbb{N}_{site} \times \mathbb{N}) \xleftrightarrow[\alpha_{\mathbb{H}}]{\gamma_{\mathbb{H}}} \mathbb{H}^\sharp = \wp(\mathbb{N}_{site})$$

$$\alpha_{\mathbb{H}}(H) = \{\mu \mid \langle \mu, - \rangle \in H\}$$

$$\gamma_{\mathbb{H}}(h^\sharp) = \{\langle \mu, n \rangle \mid \mu \in h^\sharp, n \in \mathbb{N}\}$$

- The abstract domain is the power set of variables and abstract heap locations
  - It is a finite set: the set of variables and malloc sites are finite for a program

$$\wp(\mathbb{A} = \mathbb{X} \cup \mathbb{H}) \xleftrightarrow[\alpha_{\mathbb{A}}]{\gamma_{\mathbb{A}}} \mathbb{A}^\sharp = \wp(\mathbb{X} \cup \mathbb{H}^\sharp)$$

# Abstract State Transition

- The abstract semantics is defined using a transition system  $(\mathbb{S}^\sharp, \rightarrow^\sharp)$ 
  - $\mathbb{S}^\sharp = \mathbb{L} \times \mathbb{M}^\sharp$ : the set of states  $\langle l, m^\sharp \rangle$
  - $(\rightarrow^\sharp) \subseteq \mathbb{S}^\sharp \times \mathbb{S}^\sharp$ : the transition relation that describes computation steps
- The abstract transition for indirect assignments

$$*E_1 := E_2 : \langle l, m^\sharp \rangle \rightarrow^\sharp \langle \text{next}(l), \text{update}^\sharp(m^\sharp, \text{eval}_{E_1}^\sharp(m^\sharp), \text{eval}_{E_2}^\sharp(m^\sharp)) \rangle$$

# Abstract Semantic Operators

- The abstract expression-evaluation operation:

$$\begin{aligned} eval_E^\sharp &: \mathbb{M}^\sharp \rightarrow \mathbb{V}^\sharp \\ eval_x^\sharp(m^\sharp) &= fetch^\sharp(m^\sharp, x) \\ eval_{\&x}^\sharp(m^\sharp) &= \{x\} \\ eval_{\text{malloc}_\mu}^\sharp(m^\sharp) &= \{\mu\} \\ eval_{*E}^\sharp(m^\sharp) &= fetch^\sharp(m^\sharp, eval_E^\sharp(m^\sharp)) \end{aligned}$$

- The abstract memory filter operation:

$$\begin{aligned} filter_E^\sharp &: \mathbb{M}^\sharp \rightarrow \mathbb{M}^\sharp \\ filter_E^\sharp(m^\sharp) &= \alpha_{\mathbb{M}}(\{m \in \gamma_{\mathbb{M}}(m^\sharp) \mid eval_E(m) = \text{true}\}) \end{aligned}$$

# Safe Memory Read Operation

- The abstract memory **read** operation  $fetch^\# : \mathbb{M}^\# \times \mathbb{A}^\# \rightarrow \mathbb{V}^\#$  looks up the abstract memory entry at the abstract location
- Since the **abstract** location is a **set** of variables and malloc-sites, the result is the **join** of all the entries:

$$fetch^\#(m^\#, a^\#) = \bigsqcup_{a \in a^\#} m^\#(a)$$

# Safe Memory Write Operation

- The abstract memory **write** operation:  $update^\# : \mathbb{M}^\# \times \mathbb{A}^\# \times \mathbb{V}^\# \rightarrow \mathbb{M}^\#$ 
  - **Overwrite** the memory entry (called **strong update**) when the target abstract location means a single concrete location
  - Otherwise, every entry that constitutes the target abstract location must be **joined** with the value to store (called **weak update**)

$$update^\#(m^\#, a^\#, v^\#) = \begin{cases} m^\#[x \mapsto v^\#] & \text{if } \gamma_{\mathbb{A}}(a^\#) = \{x\} \\ \bigsqcup_{a \in a^\#} m^\#[a \mapsto m^\#(a) \sqcup v^\#] & \text{otherwise} \end{cases}$$

# Example

```
x = 0;  
y = 1;  
if (*) {  
    p = &x;      {x ↦ [0, 0], y ↦ [1, 1], p ↦ {x}}  
} else {  
    p = &y;      {x ↦ [0, 0], y ↦ [1, 1], p ↦ {y}}  
}  
z = *p;      {x ↦ [0, 0], y ↦ [1, 1], p ↦ {x, y}, z ↦ [0, 1]}  
*p = 2;      {x ↦ [0, 2], y ↦ [1, 2], p ↦ {x, y}, z ↦ [0, 1]}
```

# Abstract Semantics

- Abstract domain:  $\mathbb{D}^\sharp = \mathbb{L} \rightarrow \mathbb{M}^\sharp$

- The abstract semantic function:

$$F^\sharp : (\mathbb{L} \rightarrow \mathbb{M}^\sharp) \rightarrow (\mathbb{L} \rightarrow \mathbb{M}^\sharp)$$

$$F^\sharp(X) = \alpha(I) \sqcup Step^\sharp(X)$$

- The abstract semantic functions for transition and partition:

$$Step^\sharp : (\mathbb{L} \rightarrow \mathbb{M}^\sharp) \rightarrow (\mathbb{L} \rightarrow \mathbb{M}^\sharp)$$

$$Step^\sharp(X) = \wp(id, \sqcup) \circ \pi \circ \wp(\hookrightarrow^\sharp)(X)$$

$$\pi : \wp(\mathbb{S}^\sharp) \rightarrow (\mathbb{L} \rightarrow \wp(\mathbb{M}^\sharp))$$

$$\pi(X) = \lambda l. \{ m^\sharp \in \mathbb{M}^\sharp \mid \langle l, m^\sharp \rangle \in X \}$$

- Soundness:  $\text{lfp } F \subseteq \gamma(\bigsqcup_{i \geq 0} F^\sharp(\perp))$

# Soundness Proof

$\text{lfp } F \subseteq \gamma(\bigsqcup_{i \geq 0} F^\sharp(\perp))$  if  $F^\sharp \circ \gamma \subseteq \gamma \circ F^\sharp$  by the fixpoint transfer theorem

It is enough to prove the following soundnesses for all semantic operator:

$$\wp(eval_E) \circ \gamma_{\mathbb{M}} \subseteq \gamma_{\mathbb{M}} \circ eval_E^\sharp$$

$$\wp(update_E) \circ \times \circ (\gamma_{\mathbb{M}}, \gamma_{\mathbb{A}}, \gamma_{\mathbb{V}}) \subseteq \gamma_{\mathbb{M}} \circ update_E^\sharp$$

$$\wp(fetch) \circ \times \circ (\gamma_{\mathbb{M}}, \gamma_{\mathbb{V}}) \subseteq \gamma_{\mathbb{V}} \circ fetch_E^\sharp$$

$$\wp(filter_E) \circ \gamma_{\mathbb{M}} \subseteq \gamma_{\mathbb{M}} \circ filter_E^\sharp$$

# The ThriLLVM Language

- Extension 2: Functions

$E ::= \dots$	expression, as before
$f$	function name
$C ::= \dots$	statement, as before
$E(E)$	function call
$\text{return}$	return from call
$F ::= f(x) = C^+$	
$P ::= F^+C^+$	program



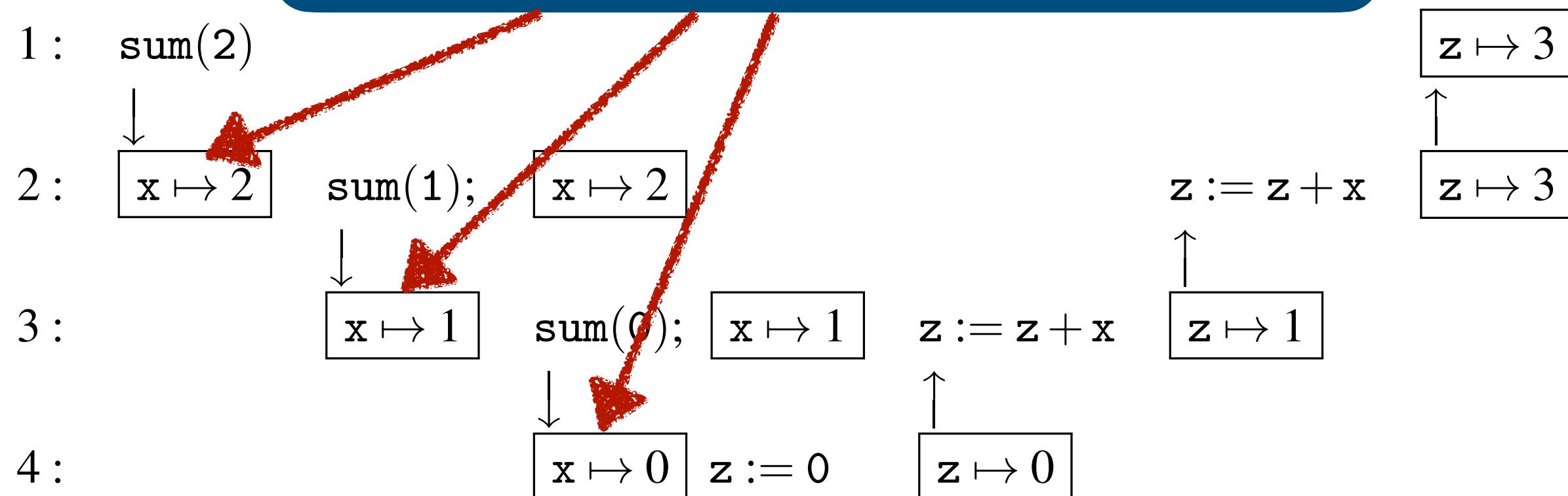
## Assume:

1. Only flat function definitions (no nested function definitions)
2. Variables other than parameters are all global
3. Function names are first-class values
4. Every function has one parameter with a unique name

# Example

```
void sum(int x) {  
    if (x == 0) {  
        z = 0;  
        return;  
    } else {  
        sum(x - 1);  
        z = z + x;  
        return;  
    }  
}  
  
sum(2);
```

Multiple instances of parameters  
(What is the current instance of x?)



What components are needed to support functions?

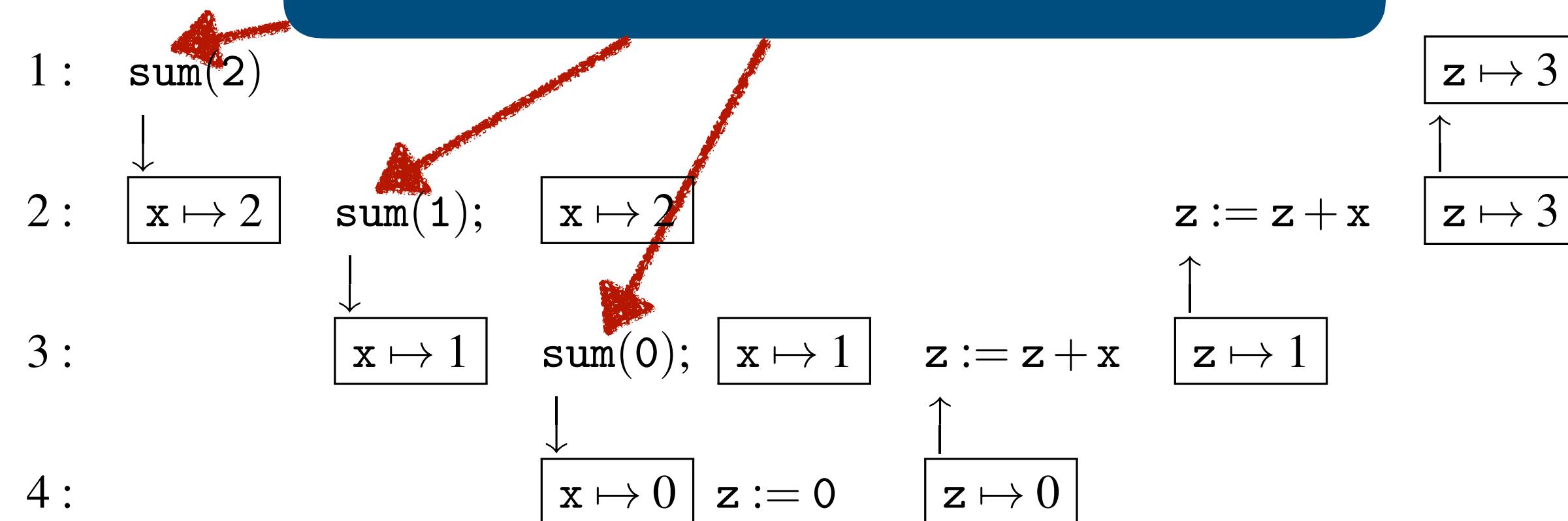
1. **Environment:** determine the current instance of the parameter

# Example

```
void sum(int x) {  
    if (x == 0) {  
        z = 0;  
        return;  
    } else {  
        sum(x - 1);  
        z = z + x;  
        return;  
    }  
}
```

```
sum(2);
```

Multiple calls to the same function  
(Where to return after function calls?)



What components are needed to support functions?

- 1. Environment:** determine the current instance of the parameter
- 2. Continuation:** remember the return context for each function call

# Concrete Domains

$\langle l, m, \sigma, \kappa, \phi \rangle \in$	$S$	$=$	$L \times M \times E \times K \times I$	
$m \in$	$M$	$=$	$A \rightarrow V$	memories
$\sigma \in$	$E$	$=$	$X \rightarrow I$	environments
$\kappa \in$	$K$	$=$	$(L \times E)^*$	continuations
$\phi \in$	$I$			instances
$a \in$	$A$	$=$	$X \times I$	addresses
$v \in$	$V$	$=$	$Z \cup F$	values
	$X$			set of variables and parameters
	$L$			set of statement labels
	$F$			set of function names

# Example

```
1: void sum(int x) {  
2:     if (x == 0) {  
3:         z = 0;  
4:         return;  
5:     } else {  
6:         sum (x - 1);  
7:         z = z + x;  
8:         return;  
9:     }  
10: }  
11: sum(2);
```

Memory		Environment		Continuation	Instance
Addr	Val	Var	Instance		
		z	0		0

# Example

```
1: void sum(int x) {  
2:     if (x == 0) {  
3:         z = 0;  
4:         return;  
5:     } else {  
6:         sum (x - 1);  
7:         z = z + x;  
8:         return;  
9:     }  
10: }  
11: sum(2);
```

Memory

Addr	Val
<x, 1>	2

Environment

Var	Instance
z	0
x	1

Continuation

11
----

Instance

1

# Example

```
1: void sum(int x) {  
2:     if (x == 0) {  
3:         z = 0;  
4:         return;  
5:     } else {  
6:         sum (x - 1); ←  
7:         z = z + x;  
8:         return;  
9:     }  
10: }  
11: sum(2);
```

Memory		Environment		Continuation	Instance
Addr	Val	Var	Instance		
<x, 1>	2	z	0		1
		x	1		
				11	

# Example

```
1: void sum(int x) {  
2:     if (x == 0) {  
3:         z = 0;  
4:         return;  
5:     } else {  
6:         sum (x - 1);  
7:         z = z + x;  
8:         return;  
9:     }  
10: }  
11: sum(2);
```

Memory		Environment		Continuation	Instance
Addr	Val	Var	Instance		
<x, 1>	2	z	0		
<x, 2>	1	x	2		
				6	2
				11	

# Example

```
1: void sum(int x) {  
2:     if (x == 0) {  
3:         z = 0;  
4:         return;  
5:     } else {  
6:         sum (x - 1); ←  
7:         z = z + x;  
8:         return;  
9:     }  
10: }  
11: sum(2);
```

Memory		Environment		Continuation	Instance
Addr	Val	Var	Instance		
<x, 1>	2	z	0		
<x, 2>	1	x	2		
				6 11	2

# Example

```
1: void sum(int x) {  
2:     if (x == 0) {  
3:         z = 0;  
4:         return;  
5:     } else {  
6:         sum (x - 1);  
7:         z = z + x;  
8:         return;  
9:     }  
10: }  
11: sum(2);
```

Memory		Environment		Continuation	Instance
Addr	Val	Var	Instance		
<x, 1>	2	z	0		
<x, 2>	1	x	3		
<x, 3>	0				

# Example

```
1: void sum(int x) {  
2:     if (x == 0) {  
3:         z = 0; ←  
4:         return;  
5:     } else {  
6:         sum (x - 1);  
7:         z = z + x;  
8:         return;  
9:     }  
10: }  
11: sum(2);
```

Memory		Environment		Continuation	Instance
Addr	Val	Var	Instance		
<x, 1>	2	z	0		
<x, 2>	1	x	3		
<x, 3>	0				
<z, 0>	0				

# Example

```
1: void sum(int x) {  
2:     if (x == 0) {  
3:         z = 0;  
4:         return; -----  
5:     } else {  
6:         sum (x - 1);  
7:         z = z + x; -----  
8:         return;  
9:     }  
10: }  
11: sum(2);
```

Memory		Environment		Continuation	Instance
Addr	Val	Var	Instance		
<x, 1>	2	z	0		
<x, 2>	1	x	2		
<x, 3>	0				
<z, 0>	1				

# Example

```
1: void sum(int x) {  
2:     if (x == 0) {  
3:         z = 0;  
4:         return;  
5:     } else {  
6:         sum (x - 1);  
7:         z = z + x;   
8:         return;   
9:     }  
10: }  
11: sum(2);
```

Memory		Environment		Continuation	Instance
Addr	Val	Var	Instance		
<x, 1>	2	z	0		
<x, 2>	1	x	1		
<x, 3>	0				
<z, 0>	3			11	3

# Example

```
1: void sum(int x) {  
2:     if (x == 0) {  
3:         z = 0;  
4:         return;  
5:     } else {  
6:         sum (x - 1);  
7:         z = z + x;  
8:         return;  
9:     }  
10: }  
11: sum(2);
```

Memory		Environment		Continuation	Instance
Addr	Val	Var	Instance		
<x, 1>	2				
<x, 2>	1				
<x, 3>	0				
<z, 0>	3				

# State Transitions

- The transition for function calls:

$$\begin{aligned} E_0(E_1) & : \langle l, m, \sigma, \kappa, \phi \rangle \\ & \rightarrow \langle \text{body}(f), \\ & \quad \text{bind}_x(m, \phi', v), \\ & \quad \text{new-env}_x(\sigma, \phi'), \\ & \quad \text{push-context}(\kappa, l, \sigma), \\ & \quad \phi' \rangle \end{aligned}$$

where  $f = eval_{E_0}(m, \sigma)$ ,  $x = param(f)$ ,  $v = eval_{E_1}(m, \sigma)$ ,  $\phi' = tick(\phi)$

- The transition for returns:

$$\text{return} : \langle l, m, \sigma, \kappa, \phi \rangle \hookrightarrow \langle \text{next}(l'), m, \sigma', \kappa', \phi \rangle$$

where  $\langle l', \sigma', \kappa' \rangle = pop-context(\kappa)$

# Concrete Semantic Operators

- The first label of the function body:  $body : \mathbb{F} \rightarrow \mathbb{L}$
- Semantic operators for function calls:

$$bind_x : \mathbb{M} \times \mathbb{I} \times \mathbb{V} \rightarrow \mathbb{M}$$
$$bind_x(m, \phi, v) = m[\langle x, \phi \rangle \mapsto v]$$

$$new-env_x : \mathbb{E} \times \mathbb{I} \rightarrow \mathbb{E}$$
$$new-env_x(\sigma, \phi) = \sigma[x \mapsto \phi]$$

$$push-context : \mathbb{K} \times \mathbb{L} \times \mathbb{E} \rightarrow \mathbb{K}$$
$$push-context(\kappa, l, \sigma) = \langle l, \sigma \rangle \cdot \kappa$$

$$tick : \mathbb{I} \rightarrow \mathbb{I}$$
$$tick(\phi) = \phi' \quad (\text{new } \phi')$$

- Semantic operators for returns:

$$pop-context : \mathbb{K} \rightarrow \mathbb{L} \times \mathbb{E} \times \mathbb{K}$$
$$pop-context(\langle l, \sigma \rangle \cdot \kappa) = \langle l, \sigma, \kappa \rangle$$

# Concrete Semantics

- Concrete semantic function:

$$F : \wp(\mathbb{S}) \rightarrow \wp(\mathbb{S})$$

$$F(X) = I \cup Step(X)$$

where  $I$  is the set of initial states and  $Step$  is the powerset-lifted version of  $\hookrightarrow$

$$Step : \wp(\mathbb{S}) \rightarrow \wp(\mathbb{S})$$

$$Step(X) = \{s' \mid s \hookrightarrow s', s \in X\}$$

- Concrete semantic (i.e., reachable states):  $\text{lfp } F$

# Abstract Domains

- The abstract domain  $\mathbb{D}^\sharp = \mathbb{L} \rightarrow \mathbb{M}^\sharp \times \mathbb{E}^\sharp \times \mathbb{K}^\sharp \times \mathbb{I}^\sharp$
- Element-wise Galois connection

$$\wp(\mathbb{L} \times \mathbb{M} \times \mathbb{E} \times \mathbb{K} \times \mathbb{I}) \xrightleftharpoons[\alpha]{\gamma} \mathbb{L} \rightarrow \mathbb{M}^\sharp \times \mathbb{E}^\sharp \times \mathbb{K}^\sharp \times \mathbb{I}^\sharp$$

where

$$\wp(\mathbb{M}) \xrightleftharpoons[\alpha_{\mathbb{M}}]{\gamma_{\mathbb{M}}} \mathbb{M}^\sharp \qquad \wp(\mathbb{E}) \xrightleftharpoons[\alpha_{\mathbb{E}}]{\gamma_{\mathbb{E}}} \mathbb{E}^\sharp$$

$$\wp(\mathbb{K}) \xrightleftharpoons[\alpha_{\mathbb{K}}]{\gamma_{\mathbb{K}}} \mathbb{K}^\sharp \qquad \wp(\mathbb{I}) \xrightleftharpoons[\alpha_{\mathbb{I}}]{\gamma_{\mathbb{I}}} \mathbb{I}^\sharp$$

# Abstract Domains (Cont'd)

- An example of a memory abstract domain:

$$M^\# = (\mathbb{X} \times \mathbb{I}^\#) \rightarrow V^\# \quad \text{where} \quad \wp(M) \xrightleftharpoons[\alpha_M]{\gamma_M} M^\# \quad \text{and} \quad M = (\mathbb{X} \times \mathbb{I}) \rightarrow V$$

- An example of an environment abstract domain:

$$E^\# = \mathbb{X} \rightarrow \mathbb{I}^\# \quad \text{where} \quad \wp(E) \xrightleftharpoons[\alpha_E]{\gamma_E} E^\# \quad \text{and} \quad E = \mathbb{X} \rightarrow \mathbb{I}$$

- An example of a value abstract domain:

$$V^\# = Z^\# \times F^\# \quad \text{where} \quad \wp(V) \xrightleftharpoons[\alpha_V]{\gamma_V} V^\# \quad \wp(Z) \xrightleftharpoons[\alpha_Z]{\gamma_Z} Z^\# \quad \wp(F) \xrightleftharpoons[\alpha_F]{\gamma_F} F^\#$$

- An example of a location abstract domain:

$$L^\# \quad \text{where} \quad \wp(L) \xrightleftharpoons[\alpha_L]{\gamma_L} L^\#$$

Abstractions of instances and continuations are parameters

# Abstract Semantics

- The abstract semantic function:

$$F^\sharp : (\mathbb{L} \rightarrow \mathbb{M}^\sharp \times \mathbb{E}^\sharp \times \mathbb{K}^\sharp \times \mathbb{I}^\sharp) \rightarrow (\mathbb{L} \rightarrow \mathbb{M}^\sharp \times \mathbb{E}^\sharp \times \mathbb{K}^\sharp \times \mathbb{I}^\sharp)$$

$$F^\sharp(X) = \alpha(I) \sqcup Step^\sharp(X)$$

- The abstract semantic functions for transition and partition:

$$Step^\sharp : (\mathbb{L} \rightarrow \mathbb{M}^\sharp \times \mathbb{E}^\sharp \times \mathbb{K}^\sharp \times \mathbb{I}^\sharp) \rightarrow (\mathbb{L} \rightarrow \mathbb{M}^\sharp \times \mathbb{E}^\sharp \times \mathbb{K}^\sharp \times \mathbb{I}^\sharp)$$

$$Step^\sharp(X) = \wp(id, \sqcup) \circ \pi \circ \wp(\hookrightarrow^\sharp)(X)$$

$$\pi : \wp(\mathbb{S}^\sharp) \rightarrow (\mathbb{L} \rightarrow \wp(\mathbb{M}^\sharp \times \mathbb{E}^\sharp \times \mathbb{K}^\sharp \times \mathbb{I}^\sharp))$$

$$\pi(X) = \lambda l. \{ \langle m^\sharp, \sigma^\sharp, \kappa^\sharp, \phi^\sharp \rangle \mid \langle l, m^\sharp, \sigma^\sharp, \kappa^\sharp, \phi^\sharp \rangle \in X \}$$

- Soundness:  $\text{lfp } F \subseteq \gamma(\bigsqcup_{i \geq 0} F^\sharp(\perp))$

# Abstract State Transition

- The transition for function calls:

$$E_0(E_1) : \langle l, m^\#, \sigma^\#, \kappa^\#, \phi^\# \rangle \xrightarrow{\#} \langle body(f),$$

How to abstract different machine states at each call?

How to abstract different continuations at each call?

$$\begin{aligned} & bind_x^\#(m^\#, \phi'^\#, v^\#), \\ & new-env_x^\#(\sigma^\#, \phi'^\#), \\ & push-context^\#(\kappa^\#, l, \sigma^\#), \\ & \phi'^\# \rangle \end{aligned}$$

where  $f \in eval_{E_0}^\#(m^\#, \sigma^\#)$ ,  $x = param(f)$ ,  $v^\# = eval_{E_1}^\#(m^\#, \sigma^\#)$ ,  $\phi'^\# = tick^\#(\phi^\#)$

- The transition for returns:

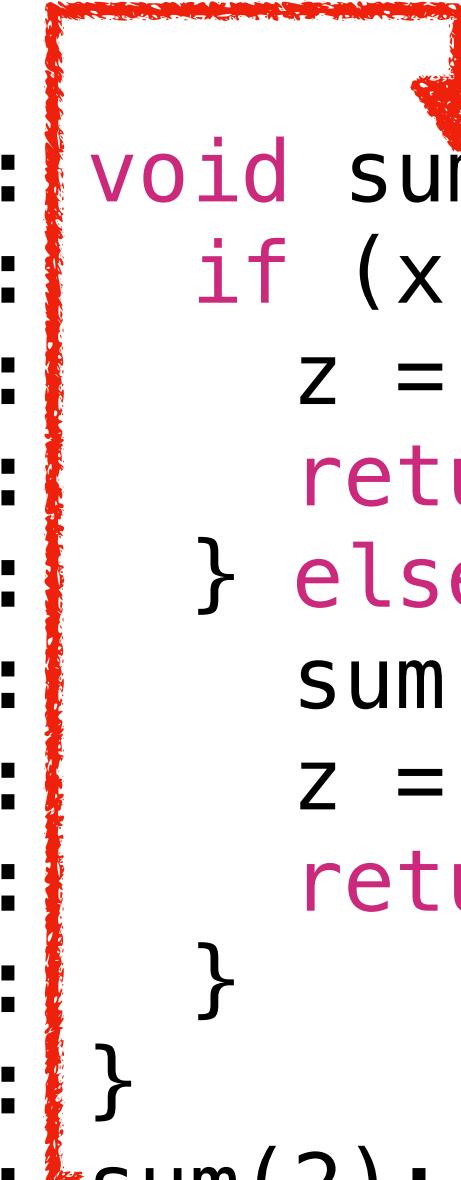
$$\text{return} : \langle l, m^\#, \sigma^\#, \kappa^\#, \phi^\# \rangle \xrightarrow{\#} \langle \text{next}(l'), m^\#, \sigma'^\#, \kappa'^\#, \phi^\# \rangle$$

where  $\langle l'^\#, \sigma'^\#, \kappa'^\# \rangle = pop-context^\#(\kappa)$  and  $l' \in l^\#$

How to abstract different continuations at each call?

# Context-sensitivity

- How to abstract different instances of machine states at each call?
- How to abstract different continuations at each call?



Memory			Continuation
Label	x	z	
1			
3			
4			
6			
7			
8			
12			

?

# Abstract Domains for Context-insensitive Analysis

- Context-insensitive: abstract all possible contexts into a single abstract one
- An example of an abstract instance domain:

$$\wp(\mathbb{I}) \xrightleftharpoons[\alpha_{\mathbb{I}}]{\gamma_{\mathbb{I}}} \mathbb{I}^{\sharp} \quad \text{where} \quad \mathbb{I}^{\sharp} = \{\cdot\}$$

Aggregate all **instances** from different contexts into a single abstract element

- An example of an abstract continuation domain:

$$\wp(\mathbb{K}) \xrightleftharpoons[\alpha_{\mathbb{K}}]{\gamma_{\mathbb{K}}} \mathbb{K}^{\sharp} \quad \text{where} \quad \mathbb{K} = (\mathbb{L} \times \mathbb{E})^* \quad \text{and} \quad \mathbb{K}^{\sharp} = \{\cdot\}$$

Aggregate all **continuations** from different contexts into a single abstract element

# Context-insensitive Analysis

```
1: void sum(int x) {  
2:     if (x == 0) {  
3:         z = 0;  
4:         return;  
5:     } else {  
6:         sum(x - 1);  
7:         z = z + x;  
8:         return;  
9:     }  
10:  
11: sum(2);
```

Label	x	z	Continuation
1	[2,2]		
3			•
4			
6			
7			
8			
12			

\*Assume a proper widening is used

# Context-insensitive Analysis

```
1: void sum(int x) {  
2:   if (x == 0) {  
3:     z = 0;  
4:     return;  
5:   } else {  
6:     sum(x - 1); ←  
7:     z = z + x;  
8:     return;  
9:   }  
10: }  
11: sum(2);
```

Label	x	z	Continuation
1	[2,2]		
3			•
4			
6	[2,2]		
7			
8			
12			

\*Assume a proper widening is used

# Context-insensitive Analysis

```
1: void sum(int x) {  
2:     if (x == 0) {  
3:         z = 0;  
4:         return;  
5:     } else {  
6:         sum(x - 1);  
7:         z = z + x;  
8:         return;  
9:     }  
10: }  
11: sum(2);
```

Label	x	z	Continuation
1	[1,2]		
3			•
4			
6	[2,2]		
7			
8			
12			

\*Assume a proper widening is used

# Context-insensitive Analysis

```
1: void sum(int x) {  
2:   if (x == 0) {  
3:     z = 0;  
4:     return;  
5:   } else {  
6:     sum(x - 1); ←  
7:     z = z + x;  
8:     return;  
9:   }  
10: }  
11: sum(2);
```

Label	x	z	Continuation
1	[1,2]		
3			•
4			
6	[1,2]		
7			
8			
12			

\*Assume a proper widening is used

# Context-insensitive Analysis

```
1: void sum(int x) {  
2:     if (x == 0) {  
3:         z = 0;  
4:         return;  
5:     } else {  
6:         sum(x - 1);  
7:         z = z + x;  
8:         return;  
9:     }  
10: }  
11: sum(2);
```

Label	x	z	Continuation
1	[0,2]		
3			•
4			
6	[1,2]		
7			
8			
12			

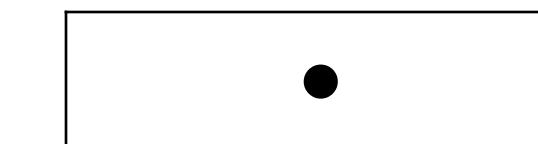
\*Assume a proper widening is used

# Context-insensitive Analysis

```
1: void sum(int x) { [ ]  
2:   if (x == 0) {  
3:     z = 0; ← [ ]  
4:     return;  
5:   } else {  
6:     sum(x - 1);  
7:     z = z + x;  
8:     return;  
9:   }  
10: }  
11: sum(2);
```

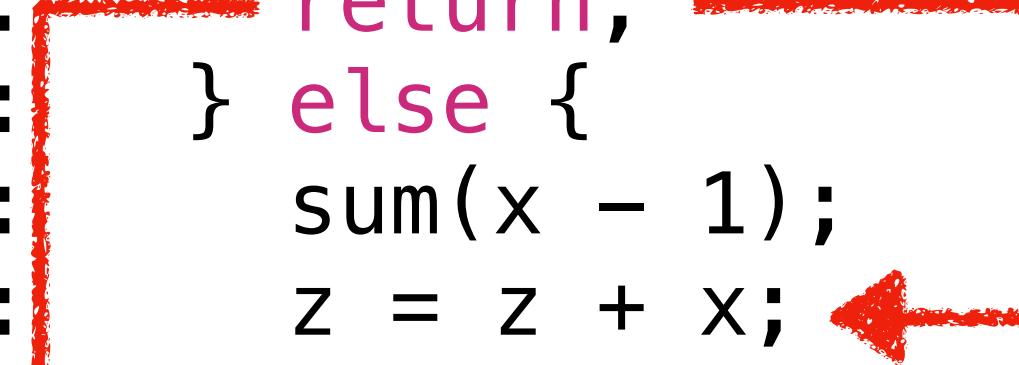
Label	x	z
1	[0,2]	
3	[0,2]	
4	[0,2]	[0,0]
6	[1,2]	
7		
8		
12		

Continuation



\*Assume a proper widening is used

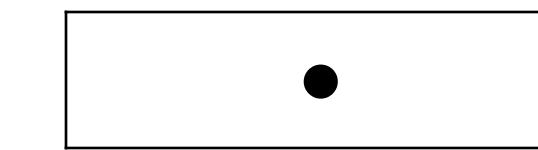
# Context-insensitive Analysis

```
1: void sum(int x) {  
2:     if (x == 0) {  
3:         z = 0;  
4:         return;   
5:     } else {  
6:         sum(x - 1);  
7:         z = z + x;   
8:         return;  
9:     }  
10:  
11: sum(2); 
```

Memory

Label	x	z
1	[0,2]	
3	[0,2]	
4	[0,2]	[0,0]
6	[1,2]	
7	[0,2]	[0,0]
8	[0,2]	[0,2]
12	[0,0]	[0,0]

Continuation



\*Assume a proper widening is used

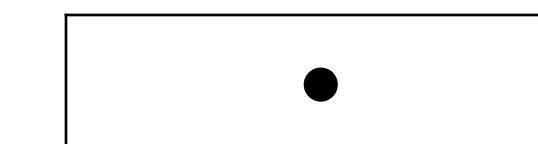
# Context-insensitive Analysis

```
1: void sum(int x) {  
2:     if (x == 0) {  
3:         z = 0;  
4:         return;  
5:     } else {  
6:         sum(x - 1);  
7:         z = z + x;   
8:         return;   
9:     }   
10: }   
11: sum(2); 
```

Memory

Label	x	z
1	[0,2]	
3	[0,2]	
4	[0,2]	[0,0]
6	[1,2]	
7	[0,2]	[0,4] 
8	[0,2]	[0,6]
12	[0,2]	[0,4]

Continuation

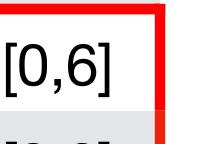
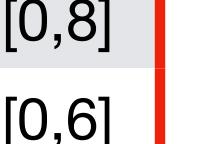


\*Assume a proper widening is used

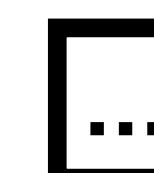
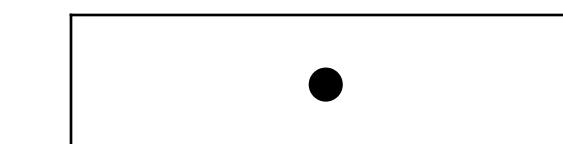
# Context-insensitive Analysis

```
1: void sum(int x) {  
2:   if (x == 0) {  
3:     z = 0;  
4:     return;  
5:   } else {  
6:     sum(x - 1);  
7:     z = z + x;   
8:     return;   
9:   }  
10: }  
11: sum(2); 
```

Memory

Label	x	z
1	[0,2]	
3	[0,2]	
4	[0,2]	[0,0]
6	[1,2]	
7	[0,2]	[0,6] 
8	[0,2]	[0,8]
12	[0,2]	[0,6] 

Continuation



\*Assume a proper widening is used

# Context-insensitive Analysis

```
1: void sum(int x) {  
2:   if (x == 0) {  
3:     z = 0;  
4:     return;  
5:   } else {  
6:     sum(x - 1);  
7:     z = z + x;   
8:   return;   
9: }   
10:  
11: sum(2); 
```

Label	x	z	Continuation
1	[0,2]		
3	[0,2]		
4	[0,2]	[0,0]	
6	[1,2]		
7	[0,2]	[0,+\infty]	
8	[0,2]	[0,+\infty]	
12	[0,2]	[0,+\infty]	

**Fixed point!**

\*Assume a proper widening is used

# Abstract Semantics for Context-insensitive Analysis

- The abstract semantic operators for function calls:

$$\begin{aligned} bind_x^\# : \mathbb{M}^\# \times \mathbb{I}^\# \times \mathbb{V}^\# &\rightarrow \mathbb{M}^\# \\ bind_x^\#(m^\#, \phi^\#, v^\#) &= m^\# \{ \langle x, \bullet \rangle \mapsto v^\# \} \end{aligned}$$

$$\begin{aligned} new-env_x^\# : \mathbb{E}^\# \times \mathbb{I}^\# &\rightarrow \mathbb{E}^\# \\ new-env_x^\#(\sigma^\#, \phi^\#) &= \sigma^\# \{ x \mapsto \bullet \} \end{aligned}$$

$$\begin{aligned} push-context^\# : \mathbb{K}^\# \times \mathbb{L} \times \mathbb{E}^\# &\rightarrow \mathbb{K}^\# \\ push-context^\#(\kappa^\#, l, \sigma^\#) &= \bullet \end{aligned}$$

Abstract all continuations

$$\begin{aligned} tick^\# : \mathbb{I}^\# &\rightarrow \mathbb{I}^\# \\ tick^\#(\phi^\#) &= \bullet \end{aligned}$$

Abstract all instances

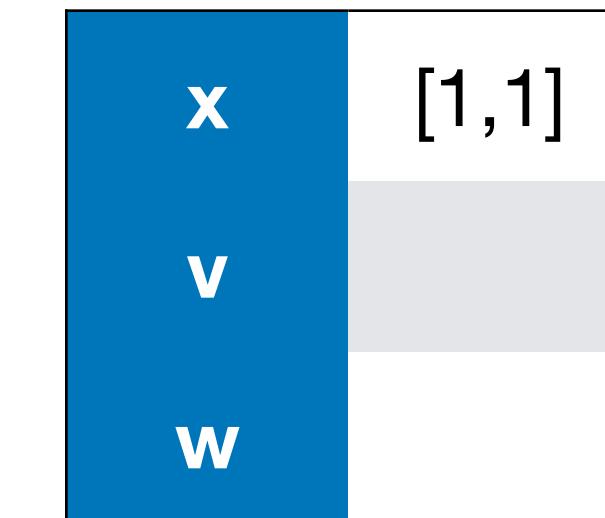
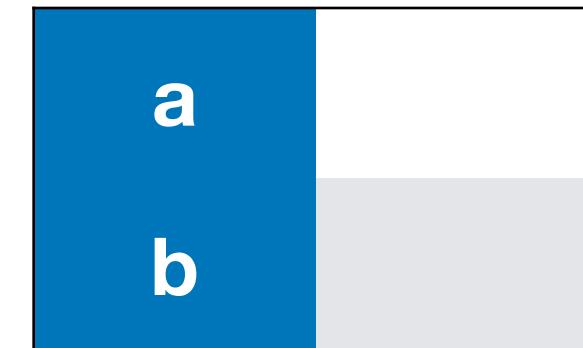
- The abstract semantic operator for returns:

$$\begin{aligned} pop-context^\# : \mathbb{K}^\# &\rightarrow \mathbb{L}^\# \times \mathbb{E}^\# \times \mathbb{K}^\# \\ pop-context^\#(\kappa^\#) &= \langle l^\#, \sigma^\#, \bullet \rangle \\ \text{where } l^\# &= \{ l \mid \langle l, -, -, -, - \rangle \hookrightarrow^\# \langle body(f), -, -, -, - \rangle \} \\ \text{and } \sigma^\# &= \lambda x. \bullet \end{aligned}$$

All possible call-sites for f

# Example

```
void main() { → int f(int x) { → int g(int y) {  
    a = f(1); ← v = g(x); ← return y;  
    b = f(3); ← w = g(x + 1); }  
} } return v + w;
```



\*Assume a proper widening is used

# Example

```
void main() {  
    a = f(1);  
    b = f(3);  
}
```

a	
b	

```
int f(int x) {  
    v = g(x);  
    w = g(x + 1);  
    return v + w;  
}
```

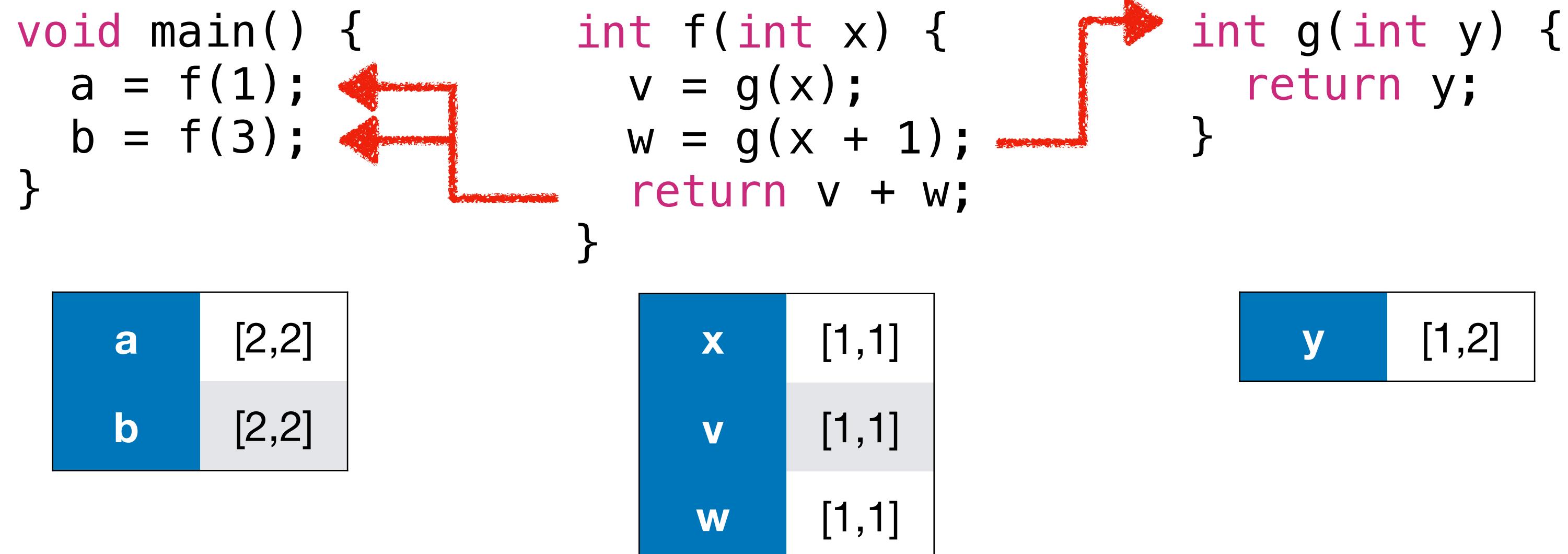
x	[1,1]
v	[1,1]
w	[1,1]

```
int g(int y) {  
    return y;  
}
```

y	[1,1]
---	-------

\*Assume a proper widening is used

# Example



\*Assume a proper widening is used

# Example

```
void main() {  
    a = f(1);  
    b = f(3);  
}
```

```
int f(int x) {  
    v = g(x);  
    w = g(x + 1);  
    return v + w;  
}
```

```
int g(int y) {  
    return y;  
}
```

a	[2,2]
b	[2,2]

x	[1,3]
v	[1,2]
w	[1,2]

y	[1,2]
---	-------

...

\*Assume a proper widening is used

# Example

```
void main() {  
    a = f(1);  
    b = f(3);  
}
```

a	[2,8]
b	[2,8]

```
int f(int x) {  
    v = g(x);  
    w = g(x + 1);  
    return v + w;  
}
```

x	[1,3]
v	[1,4]
w	[1,4]

```
int g(int y) {  
    return y;  
}
```

y	[1,4]
---	-------

Fixed point!

# Context-sensitivity

- How to abstract different instances of machine states at each call?
- How to abstract different continuations at each call?

```
1: void sum(int x) {  
2:   if (x == 0) {  
3:     z = 0;  
4:     return;  
5:   } else {  
6:     sum(x - 1);  
7:     z = z + x;  
8:     return;  
9:   }  
10: }  
11: sum(2);
```

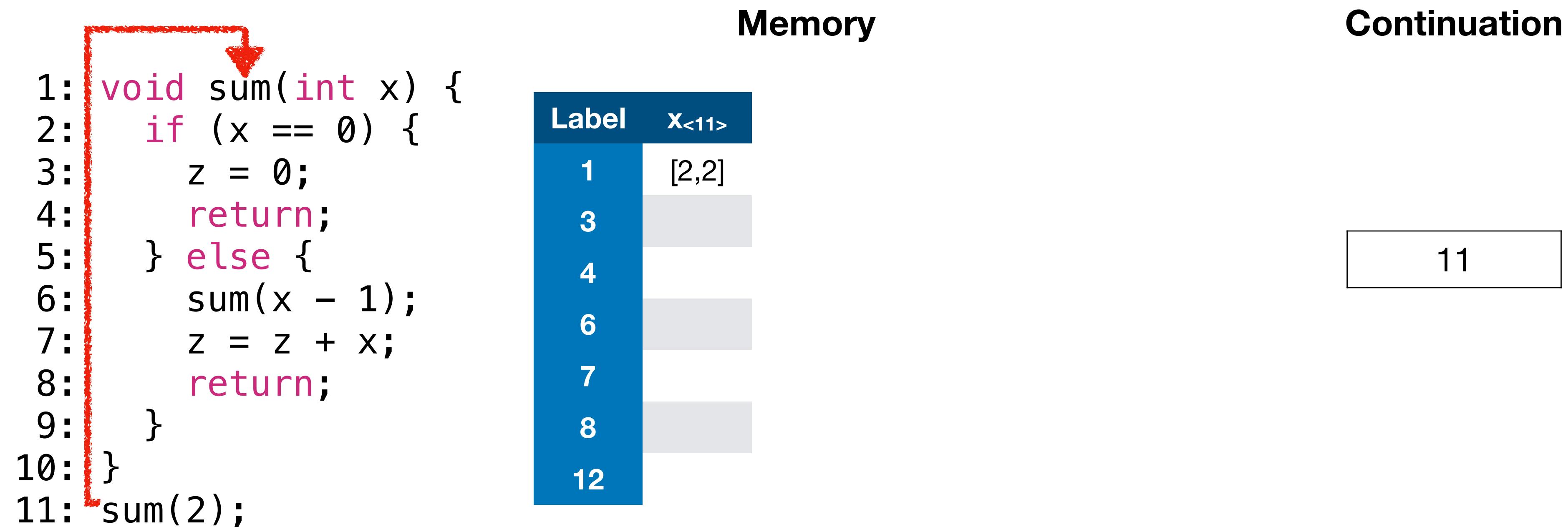
Memory			Continuation
Label	x	z	
1	[0,2]		
3	[0,2]		
4	[0,2]	[0,0]	
6	[1,2]		
7	[0,2]	[0,+∞]	
8	[0,2]	[0,+∞]	
12	[0,2]	[0,+∞]	



# Context-sensitive Analysis

- Distinguish different contexts (instances and continuations)
- In principle, many possible designs for context-sensitive analysis
- In this lecture, we use a well-known approach based on call-string
  - Call-string: an ordered sequence of call-sites

# Context-sensitive Analysis



# Context-sensitive Analysis

Memory		Continuation
		11
Label	x<11>	
1	[2,2]	
3		
4		
6	[2,2]	
7		
8		
12		

1: void sum(int x) { [ ] }  
2: if (x == 0) { }  
3: z = 0;  
4: return;  
5: } else {  
6: sum(x - 1); ←  
7: z = z + x;  
8: return;  
9: }  
10: }  
11: sum(2);

# Context-sensitive Analysis

```
1: void sum(int x) {  
2:     if (x == 0) {  
3:         z = 0;  
4:         return;  
5:     } else {  
6:         sum(x - 1);  
7:         z = z + x;  
8:         return;  
9:     }  
10: }  
11: sum(2);
```

Memory				Continuation
Label	$x_{11}$	$x_{6,11}$		
1	[2,2]	[1,1]		
3				6
4				11
6	[2,2]			
7				
8				
12				

# Context-sensitive Analysis

```
1: void sum(int x) {  
2:   if (x == 0) {  
3:     z = 0;  
4:     return;  
5:   } else {  
6:     sum(x - 1);  
7:     z = z + x;  
8:     return;  
9:   }  
10: }  
11: sum(2);
```

Memory		Continuation	
Label	x<11>	x<6,11>	
1	[2,2]	[1,1]	
3			6
4			11
6	[2,2]	[1,1]	
7			
8			
12			

# Context-sensitive Analysis

```
1: void sum(int x) {  
2:     if (x == 0) {  
3:         z = 0;  
4:         return;  
5:     } else {  
6:         sum(x - 1);  
7:         z = z + x;  
8:         return;  
9:     }  
10: }  
11: sum(2);
```

Memory					Continuation
Label	$x_{11}$	$x_{6,11}$	$x_{6,6,11}$		
1	[2,2]	[1,1]	[0,0]		6
3					6
4					11
6	[2,2]	[1,1]			
7					
8					
12					

# Context-sensitive Analysis

**Memory**

```
1: void sum(int x) {  
2:     if (x == 0) {  
3:         z = 0; ← [red box]  
4:         return;  
5:     } else {  
6:         sum(x - 1);  
7:         z = z + x;  
8:         return;  
9:     }  
10: }  
11: sum(2);
```

**Continuation**

6
6
11

Label	X<11>	X<6,11>	X<6,6,11>	Z<6,6,11>
1	[2,2]	[1,1]	[0,0]	
3			[0,0]	
4			[0,0]	[0,0]
6	[2,2]	[1,1]		
7				
8				
12				

# Context-sensitive Analysis

```
1: void sum(int x) {  
2:     if (x == 0) {  
3:         z = 0;  
4:         return; ———  
5:     } else {  
6:         sum(x - 1);  
7:         z = z + x; ←  
8:         return;  
9:     }  
10: }  
11: sum(2);
```

Label	Memory					Continuation
	X<11>	X<6,11>	X<6,6,11>	Z<6,6,11>	Z<6,11>	
1	[2,2]	[1,1]	[0,0]			6
3			[0,0]			11
4			[0,0]	[0,0]		
6	[2,2]	[1,1]				
7				[0,0]		
8				[1,1]		
12					Z<6,11> + X<6,11>	

# Context-sensitive Analysis

```
1: void sum(int x) {  
2:     if (x == 0) {  
3:         z = 0;  
4:         return;  
5:     } else {  
6:         sum(x - 1);  
7:         z = z + x;   
8:         return;   
9:     }  
10: }  
11: sum(2);
```

Label	Memory						Continuation
	X<11>	X<6,11>	X<6,6,11>	Z<6,6,11>	Z<6,11>	Z<11>	
1	[2,2]	[1,1]	[0,0]				
3			[0,0]				
4				[0,0]	[0,0]		11
6	[2,2]	[1,1]					
7				[0,0]	[1,1]		
8				[1,1]	[3,3]		
12							$Z_{11} + X_{11}$

# Context-sensitive Analysis

```
1: void sum(int x) {  
2:     if (x == 0) {  
3:         z = 0;  
4:         return;  
5:     } else {  
6:         sum(x - 1);  
7:         z = z + x;  
8:     }  
9: }  
10:  
11: sum(2);
```

Memory

Label	X<11>	X<6,11>	X<6,6,11>	Z<6,6,11>	Z<6,11>	Z<11>	z
1	[2,2]	[1,1]	[0,0]				
3			[0,0]				
4				[0,0]	[0,0]		
6	[2,2]	[1,1]					
7					[0,0]	[1,1]	
8					[1,1]	[3,3]	
12							[3,3]

# Problems of Full Context-sensitivity

- Precision vs Cost

Label	x	z
1	[0,2]	
3	[0,0]	
4	[0,0]	[0,0]
6	[1,2]	
7	[0,2]	[0,4]
8	[0,2]	[0,2]
12	[0,2]	[0,4]

Context-insensitive

Label	$x_{<11>}$	$x_{<6,11>}$	$x_{<6,6,11>}$	$z_{<6,6,11>}$	$z_{<6,11>}$	$z_{<11>}$	z
1	[2,2]	[1,1]	[0,0]				
3			[0,0]				
4			[0,0]	[0,0]			
6	[2,2]	[1,1]					
7				[0,0]	[1,1]		
8				[1,1]	[3,3]		
12							[3,3]

Fully Context-sensitive

- May not terminate because of infinitely long call-strings

```
1: void sum(int x) {
2:   if (???) {
3:     z = 0;
4:     return;
5:   } else {
6:     sum(x - 1);
7:     z = z + x;
8:     return;
9:   }
10: }
11: sum(2);
```

Label	$x_{<11>}$	$x_{<6,11>}$	$x_{<6,6,11>}$	$x_{<6,6,6,11>}$	...
1	[2,2]	[1,1]	[0,0]	[-1,-1]	
3					
4					
6	[2,2]	[1,1]	[0,0]	[-1,-1]	
7					
8					
12					

# Partial Context-sensitivity

- The most common way: keep only the top-most  $k$  continuations (so-called  $k$ -CFA)
  - $k = 0$ : ignore all contexts, i.e., context-insensitive
  - $k = \infty$ : keep all contexts, i.e., fully context-sensitive

$$\mathbb{I}^\sharp = \bigcup_{0 \leq i \leq k} \mathbb{L}^i$$

$$\mathbb{K}^\sharp = \bigcup_{0 \leq i \leq k} (\mathbb{L} \times \mathbb{E}^\sharp)^i$$

$$\alpha_{\mathbb{K}}(K) = \{\kappa^\sharp \mid \kappa^\sharp \text{ is a prefix of } \kappa \in K \text{ and } |\kappa^\sharp| \leq k\}$$

$$\gamma_{\mathbb{K}}(\kappa^\sharp) = \{\kappa \in \mathbb{K} \mid \kappa^\sharp \text{ is a prefix of } \kappa\}$$

# Example

- Suppose  $k = 1$

```
void main() {  
1:   a = f(1);  
2:   b = f(3);  
}  
  
int f(int x) {  
3:   v = g(x);  
4:   w = g(x + 1);  
5:   return v + w;  
}  
  
int g(int y) {  
6:   return y;  
}
```

$x_{<1>}$	[1,1]
$x_{<2>}$	[3,3]

$y_{<3>}$	[1,3]
$y_{<4>}$	[2,4]

# Example

- Suppose  $k = 1$

```
void main() {  
1:   a = f(1);  
2:   b = f(3);  
}
```

```
int f(int x) {  
3:   v = g(x);  
4:   w = g(x + 1);  
5:   return v + w;  
}
```

```
int g(int y) {  
6:   return y;  
}
```

$x_{<1>}$	[1,1]
$x_{<2>}$	[3,3]
$v_{<1>}$	[1,3]
$v_{<2>}$	[1,3]
$w_{<1>}$	[2,4]
$w_{<2>}$	[2,4]

$y_{<3>}$	[1,3]
$y_{<4>}$	[2,4]

# Example

- Suppose  $k = 1$

```
void main() {  
1:   a = f(1); ←  
2:   b = f(3); ←  
}  
int f(int x) {  
3:   v = g(x);  
4:   w = g(x + 1);  
5:   return v + w;  
}
```

a	[3,7]
b	[3,7]

x <sub>&lt;1&gt;</sub>	[1,1]
x <sub>&lt;2&gt;</sub>	[3,3]
v <sub>&lt;1&gt;</sub>	[1,3]
v <sub>&lt;2&gt;</sub>	[1,3]
w <sub>&lt;1&gt;</sub>	[2,4]
w <sub>&lt;2&gt;</sub>	[2,4]

```
int g(int y) {  
6:   return y;  
}
```

y <sub>&lt;3&gt;</sub>	[1,3]
y <sub>&lt;4&gt;</sub>	[2,4]

Fixed point!

# Example

- Suppose  $k \geq 2$

```
void main() {  
1:   a = f(1);  
2:   b = f(3);  
}  
  
int f(int x) {  
3:   v = g(x);  
4:   w = g(x + 1);  
5:   return v + w;  
}
```

$x_{<1>}$	[1,1]
$x_{<2>}$	[3,3]

$y_{<3,1>}$	[1,1]
$y_{<3,2>}$	[3,3]
$y_{<4,1>}$	[2,2]
$y_{<4,2>}$	[4,4]

# Example

- Suppose  $k \geq 2$

```
void main() {  
1:   a = f(1);  
2:   b = f(3);  
}
```

```
int f(int x) {  
3:   v = g(x);  
4:   w = g(x + 1);  
5:   return v + w;  
}
```

```
int g(int y) {  
6:   return y;  
}
```

$x_{<1>}$	[1,1]
$x_{<2>}$	[3,3]
$v_{<1>}$	[1,1]
$v_{<2>}$	[3,3]
$w_{<1>}$	[2,2]
$w_{<2>}$	[4,4]

$y_{<3,1>}$	[1,1]
$y_{<3,2>}$	[3,3]
$y_{<4,1>}$	[2,2]
$y_{<4,2>}$	[4,4]

# Example

- Suppose  $k \geq 2$

void main() {	int f(int x) {	int g(int y) {
1: a = f(1);	3: v = g(x);	6: return y;
2: b = f(3);	4: w = g(x + 1);	}
}	5: return v + w;	
	}	

a	[3,3]
b	[7,7]

x <sub>&lt;1&gt;</sub>	[1,1]
x <sub>&lt;2&gt;</sub>	[3,3]
v <sub>&lt;1&gt;</sub>	[1,1]
v <sub>&lt;2&gt;</sub>	[3,3]
w <sub>&lt;1&gt;</sub>	[2,2]
w <sub>&lt;2&gt;</sub>	[4,4]

y <sub>&lt;3,1&gt;</sub>	[1,1]
y <sub>&lt;3,2&gt;</sub>	[3,3]
y <sub>&lt;4,1&gt;</sub>	[2,2]
y <sub>&lt;4,2&gt;</sub>	[4,4]

# Example

- Suppose  $k \geq 2$

```
void main() {  
1:   a = f(1);  
2:   b = f(3);  
}
```

a	[3,3]
b	[7,7]

```
int f(int x) {  
3:   v = g(x);  
4:   w = g(x + 1);  
5:   return v + w;  
}
```

x <sub>&lt;1&gt;</sub>	[1,1]
x <sub>&lt;2&gt;</sub>	[3,3]
v <sub>&lt;1&gt;</sub>	[1,1]
v <sub>&lt;2&gt;</sub>	[3,3]
w <sub>&lt;1&gt;</sub>	[2,2]
w <sub>&lt;2&gt;</sub>	[4,4]

```
int g(int y) {  
6:   return y;  
}
```

y <sub>&lt;3,1&gt;</sub>	[1,1]
y <sub>&lt;3,2&gt;</sub>	[3,3]
y <sub>&lt;4,1&gt;</sub>	[2,2]
y <sub>&lt;4,2&gt;</sub>	[4,4]

Fixed point!

# Summary

- Handing more **advanced programming features** (pointers and functions)
  - Crucial parts for precision and scalability
- Many possible **choices of abstractions** for such features
  - E.g., allocation-site-based heap abstraction, call-string-based context abstraction, etc