Introduction to Program Analysis

9. Static Analysis for Advanced Programming Features

Kihong Heo

KAIST

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
	|	$E \circledcirc E$	comparison operation
	|	$\&x$	location of a variable
	|	`malloc`	location of a newly allocated memory
	|	$*E$	dereference of a memory location
$\oplus$	::=	$+ \mid - \mid \times \mid /$	
$\circledcirc$	::=	$< \mid \leq \mid > \mid \geq \mid == \mid !=$	
C	::=	$x := E$	assignment
	|	`br` E L L	conditional jump
	|	`goto` L	unconditional jump
	|	$x :=$ `input()`	input
	|	`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

$$
\begin{array}{rcll}
\mathbb{S} & = & \mathbb{L} \times \mathbb{M} & \text{states} \\
\mathbb{M} & = & \mathbb{A} \to \mathbb{V} & \text{memories} \\
\mathbb{A} & = & \mathbb{X} \cup \mathbb{H} & \text{addresses (locations)} \\
\mathbb{V} & = & \mathbb{Z} \cup \mathbb{A} & \text{values} \\
\mathbb{X} & & & \text{set of variables} \\
\mathbb{H} & & & \text{set of allocated heap addresses} \\
\mathbb{L} & & & \text{set of statement labels}
\end{array}
$$

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 μ, writing mallocμ
  - For example, the addresses of fresh locations from a malloc-site μ are (μ, 0), (μ, 1), (μ, 2), …

```
  while(*) {
μ:  p = malloc(size);
  }
```

State Transition

- The semantics is specified by a transition system $(\mathbb{S}, \hookrightarrow)$
  - $\mathbb{S} = \mathbb{L} \times \mathbb{M}$: the set of states $\langle l, m \rangle$
  - $(\hookrightarrow) \subseteq \mathbb{S} \times \mathbb{S}$: the transition relation that describes computation steps

- New rule: the transition for indirect assignments

$$*E_1 := E_2 \quad : \quad \langle l, m \rangle \hookrightarrow \langle \texttt{next}(l), update(m, eval_{E_1}(m), eval_{E_2}(m)) \rangle$$

Semantic Operators

- The memory read/write operations

$$fetch : \mathbb{M} \times \mathbb{V} \to \mathbb{V}$$
$$fetch(m, v) = m(v)$$

$$update : \mathbb{M} \times \mathbb{V} \times \mathbb{V} \to \mathbb{M}$$
$$update(m, v_1, v_2) = m\{v_1 \mapsto v_2\}$$

- The expression-evaluation operation

$$\begin{array}{rcll} eval_E & : & \mathbb{M} \to \mathbb{V} & \\ eval_x(m) & = & fetch(m, x) & \\ eval_{\&x}(m) & = & x & \text{variable as a location } \texttt{\&x} \\ eval_{\texttt{malloc}_\mu}(m) & = & (\mu, z) & \text{new number } z \text{ for malloc site } \mu \\ eval_{*E}(m) & = & fetch(m, eval_E(m)) & \text{dereference of a location} \end{array}$$

- The memory filter operation

$$filter_E : \mathbb{M} \to \mathbb{M}$$
$$filter_E(m) = m \quad \text{if } eval_E(m) = \mathsf{true}$$

Concrete Semantics

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

$$F : \wp(\mathbb{S}) \to \wp(\mathbb{S})$$
$$F(X) = I \cup Step(X)$$

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

$$Step : \wp(\mathbb{S}) \to \wp(\mathbb{S})$$
$$Step(X) = \{s' \mid s \hookrightarrow s', s \in X\}$$

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

Abstract Domains

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

$$
\begin{array}{rcll}
\mathbb{S}^\sharp & = & \mathbb{L} \times \mathbb{M}^\sharp & \text{abstract states} \\
\mathbb{M}^\sharp & = & (\mathbb{X} \cup \mathbb{H}^\sharp) \to \mathbb{V}^\sharp & \text{abstract memories} \\
\mathbb{A}^\sharp & = & \wp(\mathbb{X} \cup \mathbb{H}^\sharp) & \text{abstract addresses (locations)} \\
\mathbb{V}^\sharp & = & \mathbb{Z}^\sharp \times \mathbb{A}^\sharp & \text{abstract values} \\
\mathbb{X} & & & \text{set of variables} \\
\mathbb{H}^\sharp & & & \text{set of abstract heap addresses} \\
\mathbb{L} & & & \text{set of statement labels}
\end{array}
$$

Abstraction of Values

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

$$\wp(\mathbb{V} = \mathbb{Z} \cup \mathbb{A}) \underset{\alpha_{\mathbb{V}}}{\overset{\gamma_{\mathbb{V}}}{\leftrightarrows}} \mathbb{V}^\sharp = \mathbb{Z}^\sharp \times \mathbb{A}^\sharp$$

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

- 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 μ into a single abstract address (so called allocation-site-based abstraction)

$$\wp(\mathbb{H} = \mathbb{N}_{site} \times \mathbb{N}) \underset{\alpha_{\mathbb{H}}}{\overset{\gamma_{\mathbb{H}}}{\leftrightarrows}} \mathbb{H}^\sharp = \wp(\mathbb{N}_{site})$$

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

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

- 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

$$\wp(\mathbb{A} = \mathbb{X} \cup \mathbb{H}) \underset{\alpha_{\mathbb{A}}}{\overset{\gamma_{\mathbb{A}}}{\leftrightarrows}} \mathbb{A}^\sharp = \wp(\mathbb{X} \cup \mathbb{H}^\sharp)$$

Abstract State Transition

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

- The abstract transition for indirect assignments

$$*E_1 \texttt{ := } E_2 \quad : \quad \langle l, m^\sharp \rangle \hookrightarrow^\sharp \langle \texttt{next}(l), update^\sharp(m^\sharp, eval^\sharp_{E_1}(m^\sharp), eval^\sharp_{E_2}(m^\sharp)) \rangle$$

Abstract Semantic Operators

- The abstract expression-evaluation operation:

$$
\begin{array}{rcl}
eval^\sharp_E & : & \mathbb{M}^\sharp \to \mathbb{V}^\sharp \\
eval^\sharp_x(m^\sharp) & = & fetch^\sharp(m^\sharp, x) \\
eval^\sharp_{\&x}(m^\sharp) & = & \{x\} \\
eval^\sharp_{\texttt{malloc}_\mu}(m^\sharp) & = & \{\mu\} \\
eval^\sharp_{*E}(m^\sharp) & = & fetch^\sharp(m^\sharp, eval^\sharp_E(m^\sharp))
\end{array}
$$

- The abstract memory filter operation:

$$
filter^\sharp_E : \mathbb{M}^\sharp \to \mathbb{M}^\sharp
$$

$$
filter^\sharp_E(m^\sharp) = \alpha_{\mathbb{M}}(\{m \in \gamma_{\mathbb{M}}(m^\sharp) \mid eval_E(m) = \mathsf{true}\})
$$

Safe Memory Read Operation

- The abstract memory **read** operation $fetch^\sharp : \mathbb{M}^\sharp \times \mathbb{A}^\sharp \rightharpoonup \mathbb{V}^\sharp$ 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:

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

Safe Memory Write Operation

- The abstract memory **write** operation: $update^\sharp : \mathbb{M}^\sharp \times \mathbb{A}^\sharp \times \mathbb{V}^\sharp \to \mathbb{M}^\sharp$
  - **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***)

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

Example

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

$\{x \mapsto [0,0], y \mapsto [1,1], p \mapsto \{x\}\}$

$\{x \mapsto [0,0], y \mapsto [1,1], p \mapsto \{y\}\}$

$\{x \mapsto [0,0], y \mapsto [1,1], p \mapsto \{x,y\}, z \mapsto [0,1]\}$

$\{x \mapsto [0,2], y \mapsto [1,2], p \mapsto \{x,y\}, z \mapsto [0,1]\}$

Abstract Semantics

- Abstract domain: $\mathbb{D}^\sharp = \mathbb{L} \to \mathbb{M}^\sharp$
- The abstract semantic function:

$$F^\sharp : (\mathbb{L} \to \mathbb{M}^\sharp) \to (\mathbb{L} \to \mathbb{M}^\sharp)$$
$$F^\sharp(X) = \alpha(I) \sqcup Step^\sharp(X)$$

- The abstract semantic functions for transition and partition:

$$Step^\sharp : (\mathbb{L} \to \mathbb{M}^\sharp) \to (\mathbb{L} \to \mathbb{M}^\sharp)$$
$$Step^\sharp(X) = \wp(id, \sqcup) \circ \pi \circ \breve{\wp}(\hookrightarrow^\sharp)(X)$$

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

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

Soundness Proof

$\mathbf{lfp}F \subseteq \gamma(\bigsqcup_{i \geq 0} F^\sharp(\bot))$ 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:

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

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

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

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

The ThriLLVM Language

- Extension 2: Functions

$$
\begin{array}{lcll}
E & ::= & \cdots & \text{expression, as before} \\
 & | & \texttt{f} & \text{function name} \\
C & ::= & \cdots & \text{statement, as before} \\
 & | & E(E) & \text{function call} \\
 & | & \texttt{return} & \text{return from call} \\
F & ::= & \texttt{f}(\texttt{x}) = C^+ & \\
P & ::= & F^+ C^+ & \text{program}
\end{array}
$$

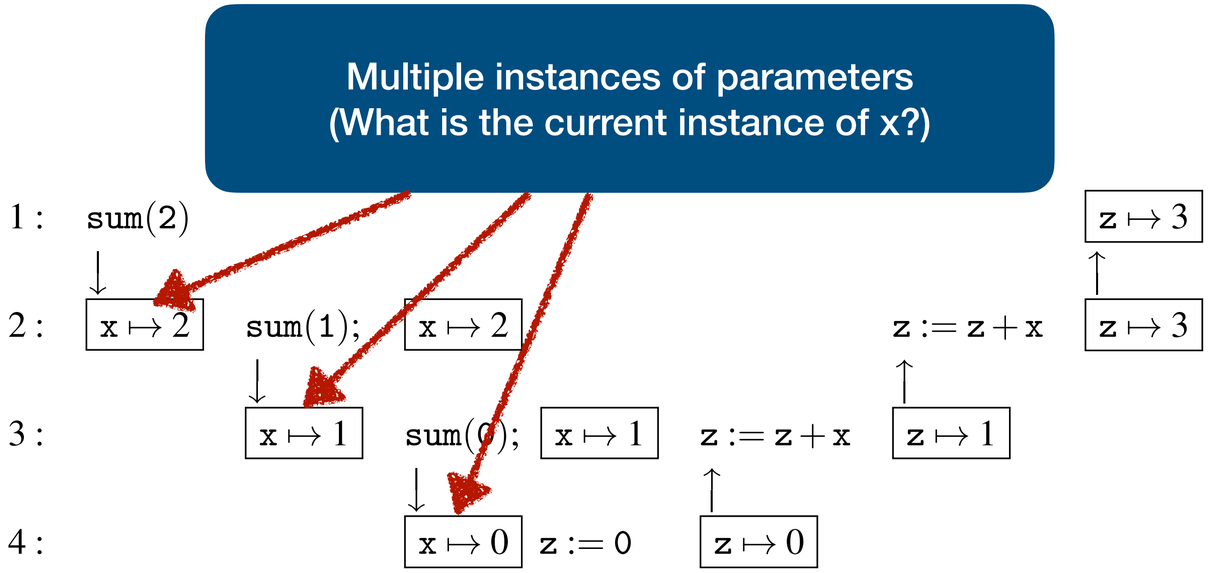
Assume:

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);
```



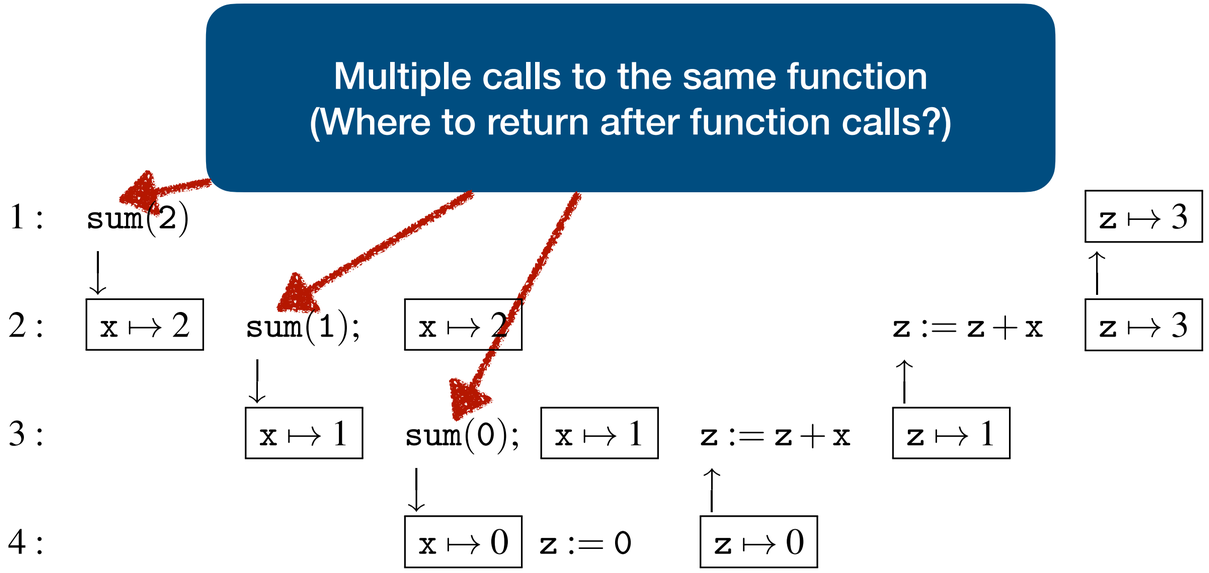
What components are needed to support functions?

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);
```



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

$$
\begin{array}{rcll}
\langle l, m, \sigma, \kappa, \phi \rangle \in & \mathbb{S} & = & \mathbb{L} \times \mathbb{M} \times \mathbb{E} \times \mathbb{K} \times \mathbb{I} \\
m \in & \mathbb{M} & = & \mathbb{A} \to \mathbb{V} \quad \text{memories} \\
\sigma \in & \mathbb{E} & = & \mathbb{X} \to \mathbb{I} \quad \text{environments} \\
\kappa \in & \mathbb{K} & = & (\mathbb{L} \times \mathbb{E})^* \quad \text{continuations} \\
\phi \in & \mathbb{I} & & \quad \text{instances} \\
a \in & \mathbb{A} & = & \mathbb{X} \times \mathbb{I} \quad \text{addresses} \\
v \in & \mathbb{V} & = & \mathbb{Z} \cup \mathbb{F} \quad \text{values} \\
& \mathbb{X} & & \quad \text{set of variables and parameters} \\
& \mathbb{L} & & \quad \text{set of statement labels} \\
& \mathbb{F} & & \quad \text{set of function names}
\end{array}
$$

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

Environment

Var	Instance
z	0

Continuation

Instance

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

Addr	Val
<x, 1>	2

Environment

Var	Instance
z	0
x	1

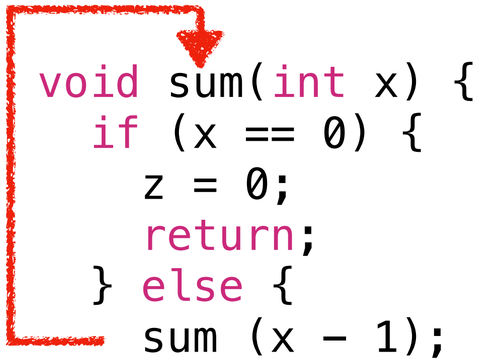
Continuation

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

Addr	Val
<x, 1>	2
<x, 2>	1

Environment

Var	Instance
z	0
x	2

Continuation

6

11

Instance

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

Addr	Val
<x, 1>	2
<x, 2>	1

Environment

Var	Instance
z	0
x	2

Continuation

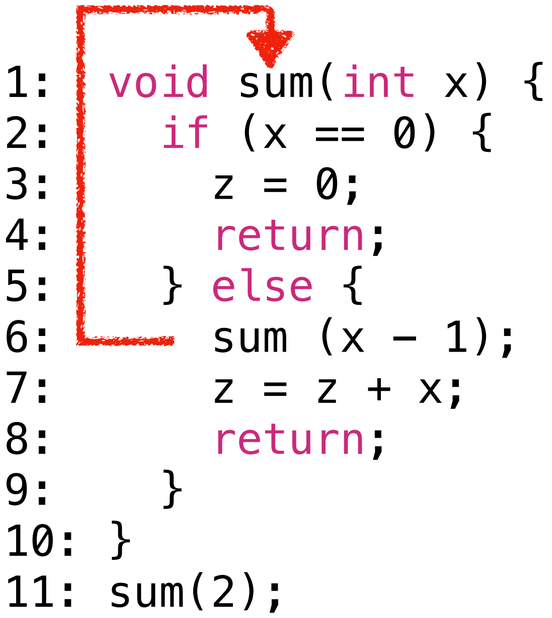
6

11

Instance

2

Example



Memory

Addr	Val
<x, 1>	2
<x, 2>	1
<x, 3>	0

Environment

Var	Instance
z	0
x	3

Continuation

6
6
11

Instance

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

Addr	Val
<x, 1>	2
<x, 2>	1
<x, 3>	0
<z, 0>	0

Environment

Var	Instance
z	0
x	3

Continuation

6
6
11

Instance

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

Addr	Val
<x, 1>	2
<x, 2>	1
<x, 3>	0
<z, 0>	1

Environment

Var	Instance
z	0
x	2

Continuation

6

11

Instance

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

Addr	Val
<x, 1>	2
<x, 2>	1
<x, 3>	0
<z, 0>	3

Environment

Var	Instance
z	0
x	1

Continuation

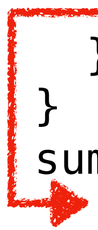
11

Instance

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

Addr	Val
<x, 1>	2
<x, 2>	1
<x, 3>	0
<z, 0>	3

Environment

Var	Instance
z	0

Continuation

Instance

3

State Transitions

- The transition for function calls:

$$
\begin{array}{rcl}
E_0(E_1) & : & \langle l, m, \sigma, \kappa, \phi \rangle \\
& & \hookrightarrow \langle body(\texttt{f}), \\
& & \quad bind_x(m, \phi', v), \\
& & \quad new\text{-}env_x(\sigma, \phi'), \\
& & \quad push\text{-}context(\kappa, l, \sigma), \\
& & \quad \phi' \rangle
\end{array}
$$

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

- The transition for returns:

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

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

Concrete Semantic Operators

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

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

$$new\text{-}env_{\mathtt{x}} : \mathbb{E} \times \mathbb{I} \to \mathbb{E}$$
$$new\text{-}env_{\mathtt{x}}(\sigma, \phi) = \sigma[\mathtt{x} \mapsto \phi]$$

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

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

- Semantic operators for returns:

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

Concrete Semantics

- Concrete semantic function:

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

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

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

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

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

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

Abstract Domains

- The abstract domain $\mathbb{D}^\sharp = \mathbb{L} \to \mathbb{M}^\sharp \times \mathbb{E}^\sharp \times \mathbb{K}^\sharp \times \mathbb{I}^\sharp$

- Element-wise Galois connection

$$\wp(\mathbb{L} \times \mathbb{M} \times \mathbb{E} \times \mathbb{K} \times \mathbb{I}) \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \mathbb{L} \to \mathbb{M}^\sharp \times \mathbb{E}^\sharp \times \mathbb{K}^\sharp \times \mathbb{I}^\sharp$$

where

$$\wp(\mathbb{M}) \underset{\alpha_{\mathbb{M}}}{\overset{\gamma_{\mathbb{M}}}{\leftrightarrows}} \mathbb{M}^\sharp \qquad \wp(\mathbb{E}) \underset{\alpha_{\mathbb{E}}}{\overset{\gamma_{\mathbb{E}}}{\leftrightarrows}} \mathbb{E}^\sharp$$

$$\wp(\mathbb{K}) \underset{\alpha_{\mathbb{K}}}{\overset{\gamma_{\mathbb{K}}}{\leftrightarrows}} \mathbb{K}^\sharp \qquad \wp(\mathbb{I}) \underset{\alpha_{\mathbb{I}}}{\overset{\gamma_{\mathbb{I}}}{\leftrightarrows}} \mathbb{I}^\sharp$$

Abstract Domains (Cont'd)

- An example of a memory abstract domain:

$$\mathbb{M}^\sharp = (\mathbb{X} \times \mathbb{I}^\sharp) \to \mathbb{V}^\sharp \quad \text{where} \quad \wp(\mathbb{M}) \underset{\alpha_\mathbb{M}}{\overset{\gamma_\mathbb{M}}{\leftrightarrows}} \mathbb{M}^\sharp \quad \text{and} \quad \mathbb{M} = (\mathbb{X} \times \mathbb{I}) \to \mathbb{V}$$

- An example of an environment abstract domain:

$$\mathbb{E}^\sharp = \mathbb{X} \to \mathbb{I}^\sharp \quad \text{where} \quad \wp(\mathbb{E}) \underset{\alpha_\mathbb{E}}{\overset{\gamma_\mathbb{E}}{\leftrightarrows}} \mathbb{E}^\sharp \quad \text{and} \quad \mathbb{E} = \mathbb{X} \to \mathbb{I}$$

- An example of a value abstract domain:

$$\mathbb{V}^\sharp = \mathbb{Z}^\sharp \times \mathbb{F}^\sharp \quad \text{where} \quad \wp(\mathbb{V}) \underset{\alpha_\mathbb{V}}{\overset{\gamma_\mathbb{V}}{\leftrightarrows}} \mathbb{V}^\sharp \quad \wp(\mathbb{Z}) \underset{\alpha_\mathbb{Z}}{\overset{\gamma_\mathbb{Z}}{\leftrightarrows}} \mathbb{Z}^\sharp \quad \wp(\mathbb{F}) \underset{\alpha_\mathbb{F}}{\overset{\gamma_\mathbb{F}}{\leftrightarrows}} \mathbb{F}^\sharp$$

- An example of a location abstract domain:

$$\mathbb{L}^\sharp \quad \text{where} \quad \wp(\mathbb{L}) \underset{\alpha_\mathbb{L}}{\overset{\gamma_\mathbb{L}}{\leftrightarrows}} \mathbb{L}^\sharp$$

Abstractions of instances and continuations are parameters

Abstract Semantics

- The abstract semantic function:

$$F^\sharp : (\mathbb{L} \to \mathbb{M}^\sharp \times \mathbb{E}^\sharp \times \mathbb{K}^\sharp \times \mathbb{I}^\sharp) \to (\mathbb{L} \to \mathbb{M}^\sharp \times \mathbb{E}^\sharp \times \mathbb{K}^\sharp \times \mathbb{I}^\sharp)$$
$$F^\sharp(X) = \alpha(I) \sqcup Step^\sharp(X)$$

- The abstract semantic functions for transition and partition:

$$Step^\sharp : (\mathbb{L} \to \mathbb{M}^\sharp \times \mathbb{E}^\sharp \times \mathbb{K}^\sharp \times \mathbb{I}^\sharp) \to (\mathbb{L} \to \mathbb{M}^\sharp \times \mathbb{E}^\sharp \times \mathbb{K}^\sharp \times \mathbb{I}^\sharp)$$
$$Step^\sharp(X) = \wp(id, \sqcup) \circ \pi \circ \breve{\wp}(\hookrightarrow^\sharp)(X)$$

$$\pi : \wp(\mathbb{S}^\sharp) \to (\mathbb{L} \to \wp(\mathbb{M}^\sharp \times \mathbb{E}^\sharp \times \mathbb{K}^\sharp \times \mathbb{I}^\sharp))$$
$$\pi(X) = \lambda l.\{\langle m^\sharp, \sigma^\sharp, \kappa^\sharp, \phi^\sharp \rangle \mid \langle l, m^\sharp, \sigma^\sharp, \kappa^\sharp, \phi^\sharp \rangle \in X\}$$

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

Abstract State Transition

- The transition for function calls:

$$
\begin{array}{rcl}
E_0(E_1) & : & \langle l, m^\sharp, \sigma^\sharp, \kappa^\sharp, \phi^\sharp \rangle \\
 & & \hookrightarrow^\sharp \langle body(\mathtt{f}), \\
 & & \quad bind^\sharp_{\mathtt{x}}(m^\sharp, \phi'^\sharp, v^\sharp), \\
 & & \quad new\text{-}env^\sharp_{\mathtt{x}}(\sigma^\sharp, \phi'^\sharp), \\
 & & \quad push\text{-}context^\sharp(\kappa^\sharp, l, \sigma^\sharp), \\
 & & \quad \phi'^\sharp \rangle
\end{array}
$$

How to abstract different machine states at each call?

How to abstract different continuations at each call?

$$\text{where } \mathtt{f} \in eval^\sharp_{E_0}(m^\sharp, \sigma^\sharp), \mathtt{x} = param(\mathtt{f}), v^\sharp = eval^\sharp_{E_1}(m^\sharp, \sigma^\sharp), \phi'^\sharp = tick^\sharp(\phi^\sharp)$$

- The transition for returns:

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

$$\text{where } \langle l'^\sharp, \sigma'^\sharp, \kappa'^\sharp \rangle = pop\text{-}context^\sharp(\kappa) \text{ and } l' \in l^\sharp$$

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?

```
 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		
3		
4		
6		
7		
8		
12		

Continuation

Abstract Domains for Context-insensitive Analysis

- Context-insensitive: abstract all possible contexts into a single abstract one

- An example of an abstract instance domain:

$$\wp(\mathbb{I}) \underset{\alpha_{\mathbb{I}}}{\overset{\gamma_{\mathbb{I}}}{\leftrightarrows}} \mathbb{I}^\sharp \qquad \text{where} \qquad \mathbb{I}^\sharp = \{\cdot\}$$

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

- An example of an abstract continuation domain:

$$\wp(\mathbb{K}) \underset{\alpha_{\mathbb{K}}}{\overset{\gamma_{\mathbb{K}}}{\leftrightarrows}} \mathbb{K}^\sharp \qquad \text{where} \qquad \mathbb{K} = (\mathbb{L} \times \mathbb{E})^* \quad \text{and} \quad \mathbb{K}^\sharp = \{\cdot\}$$

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

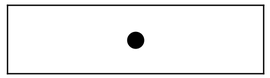
Context-insensitive Analysis

```
 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);
```

Memory

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

Continuation



***Assume a proper widening is used**

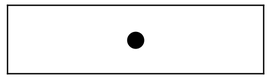
Context-insensitive Analysis

```
 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	[2,2]	
3		
4		
6	[2,2]	
7		
8		
12		

Continuation



***Assume a proper widening is used**

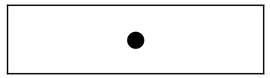
Context-insensitive Analysis

```
 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	[1,2]	
3		
4		
6	[2,2]	
7		
8		
12		

Continuation



***Assume a proper widening is used**

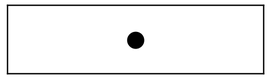
Context-insensitive Analysis

```
 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	[1,2]	
3		
4		
6	[1,2]	
7		
8		
12		

Continuation



*Assume a proper widening is used

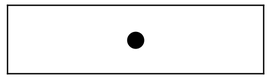
Context-insensitive Analysis

```
 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		
4		
6	[1,2]	
7		
8		
12		

Continuation



***Assume a proper widening is used**

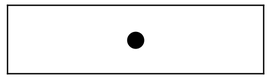
Context-insensitive Analysis

```
 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		
8		
12		

Continuation



***Assume a proper widening is used**

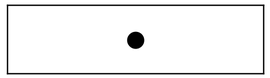
Context-insensitive Analysis

```
 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**

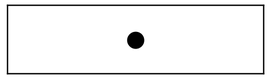
Context-insensitive Analysis

```
 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

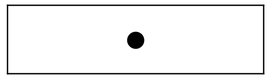
```
 1: void sum(int x) {
 2:   if (x == 0) {
 3:     z = 0;
 4:     return;
 5:   } else {
 6:     sum(x - 1);
 7:     z = z + x;
 8:     return;
 9:   }
10: }
11: sum(2);
```

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**

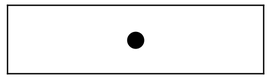
Context-insensitive Analysis

```
 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);
```

Memory

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,+∞]

Continuation



Fixed point!

***Assume a proper widening is used**

Abstract Semantics for Context-insensitive Analysis

- The abstract semantic operators for function calls:

$$bind^\sharp_{\mathtt{x}} : \mathbb{M}^\sharp \times \mathbb{I}^\sharp \times \mathbb{V}^\sharp \to \mathbb{M}^\sharp$$
$$bind^\sharp_{\mathtt{x}}(m^\sharp, \phi^\sharp, v^\sharp) = m^\sharp\{\langle \mathtt{x}, \bullet \rangle \mapsto v^\sharp\}$$

$$new\text{-}env^\sharp_{\mathtt{x}} : \mathbb{E}^\sharp \times \mathbb{I}^\sharp \to \mathbb{E}^\sharp$$
$$new\text{-}env^\sharp_{\mathtt{x}}(\sigma^\sharp, \phi^\sharp) = \sigma^\sharp\{\mathtt{x} \mapsto \bullet\}$$

$$push\text{-}context^\sharp : \mathbb{K}^\sharp \times \mathbb{L} \times \mathbb{E}^\sharp \to \mathbb{K}^\sharp$$
$$push\text{-}context^\sharp(\kappa^\sharp, l, \sigma^\sharp) = \bullet$$

Abstract all continuations

$$tick^\sharp : \mathbb{I}^\sharp \to \mathbb{I}^\sharp$$
$$tick^\sharp(\phi^\sharp) = \bullet$$

Abstract all instances

- The abstract semantic operator for returns:

$$pop\text{-}context^\sharp : \mathbb{K}^\sharp \to \mathbb{L}^\sharp \times \mathbb{E}^\sharp \times \mathbb{K}^\sharp$$
$$pop\text{-}context^\sharp(\kappa^\sharp) = \langle l^\sharp, \sigma^\sharp, \bullet \rangle$$
$$\text{where } l^\sharp = \{l \mid \langle l, \_, \_, \_, \_ \rangle \hookrightarrow^\sharp \langle body(\mathtt{f}), \_, \_, \_, \_ \rangle\}$$
$$\text{and } \sigma^\sharp = \lambda x.\bullet$$

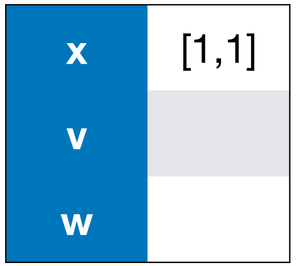
All possible call-sites for `f`

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;
}
```



*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

```
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;
}
```

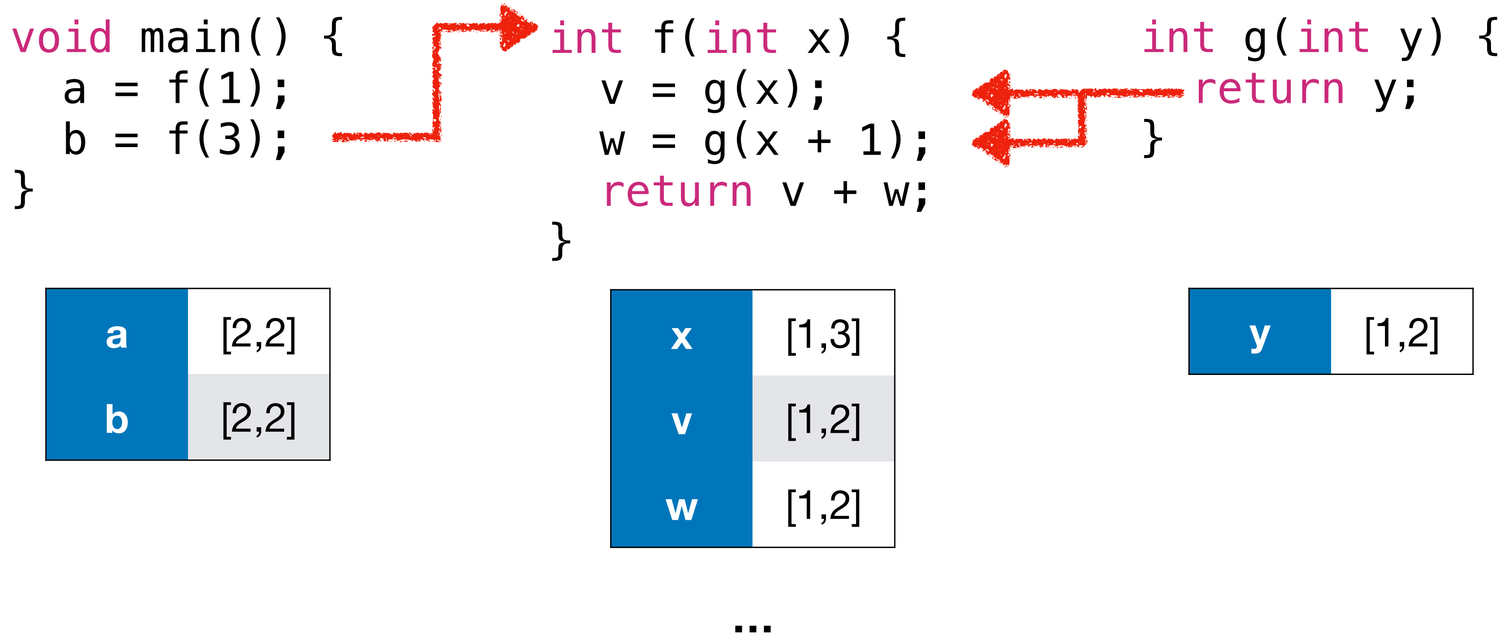
a	[2,2]
b	[2,2]

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

y	[1,2]

*Assume a proper widening is used

Example



*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]

Fixed point!

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

y	[1,4]

Context-sensitivity

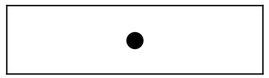
- How to abstract different instances of machine states at each call?
- How to abstract different continuations at each call?

```
 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

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,+∞]

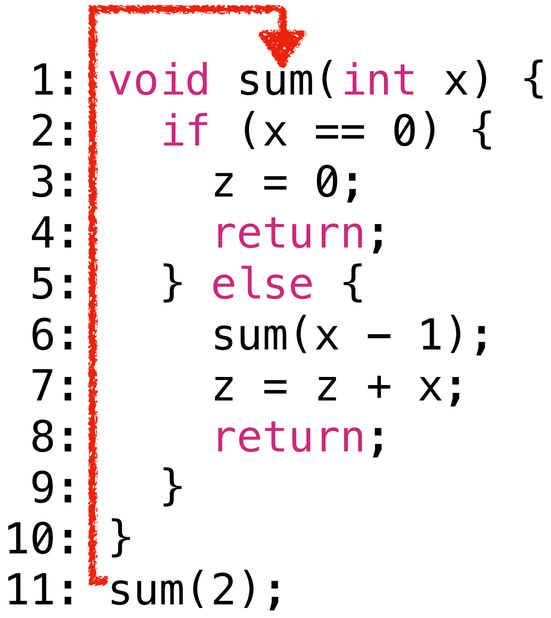
Continuation



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



```
 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	$x_{<11>}$
1	[2,2]
3	
4	
6	
7	
8	
12	

Continuation

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);
```

Memory

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

Continuation

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);
```

Memory

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

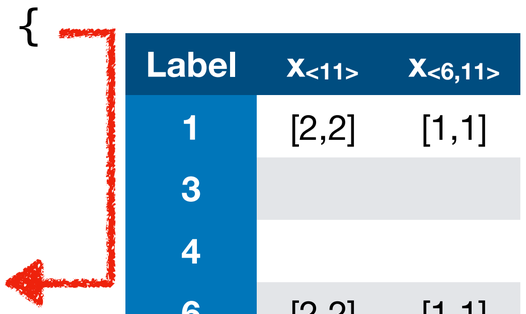
Continuation

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);
```



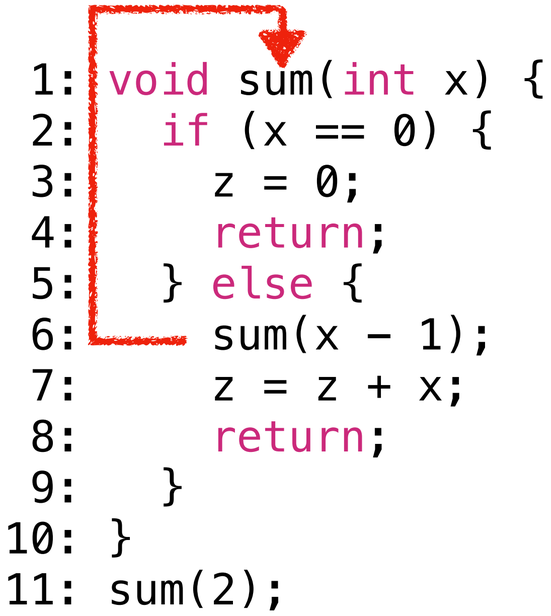
Memory

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

Continuation

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);
```

Memory

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

Continuation

6
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);
```

Memory

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				

Continuation

6

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);
```

Memory

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

$z_{<6,11>} + x_{<6,11>}$

Continuation

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);
```

Memory

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

Continuation

11

$z_{<11>} + x_{<11>}$

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:      return;
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

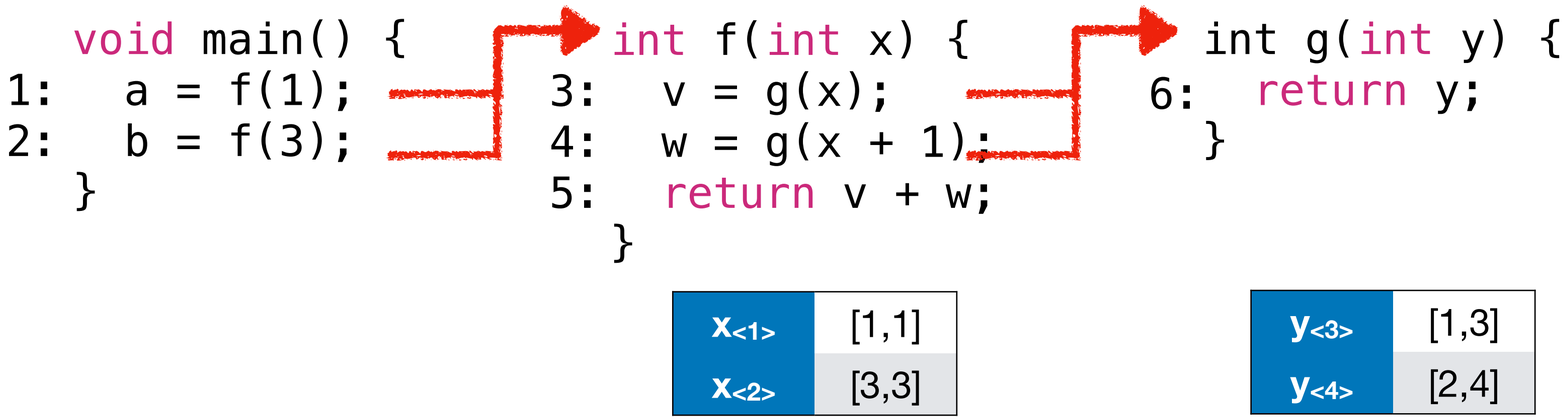
- The most common way: keep only the top-most k continuations (so-called k-CFA)
  - k = 0: ignore all contexts, i.e., context-insensitive
  - k = ∞: keep all contexts, i.e., fully context-sensitive

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

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

Example

- Suppose **k = 1**



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;
   }
```

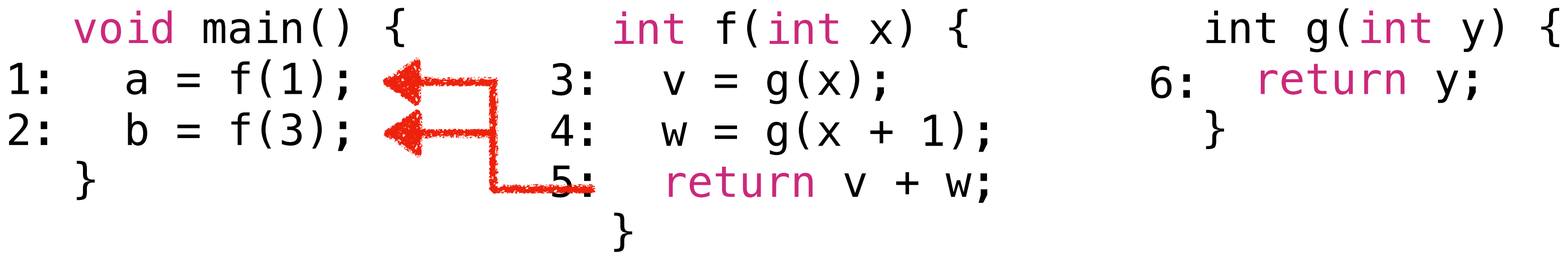
```
   int g(int y) {
6:   return y;
   }
```

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

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

Example

- Suppose **k = 1**



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

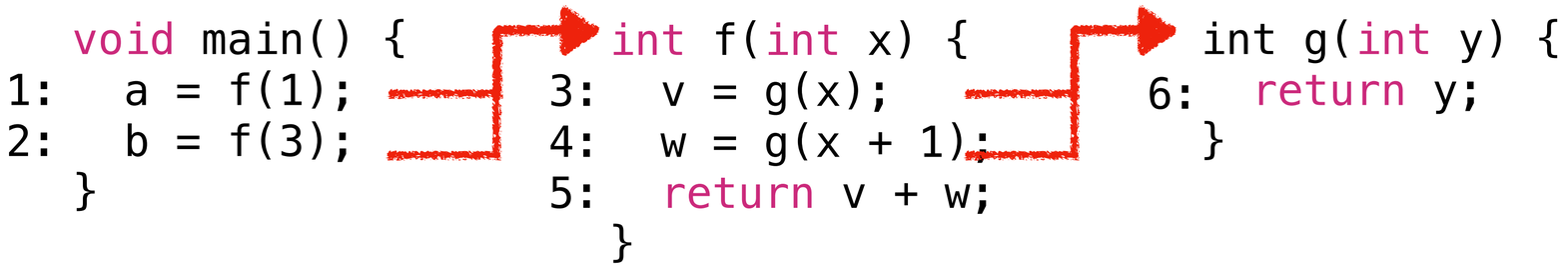
$x_{<1>}$	[1,1]
$x_{<2>}$	[3,3]
$v_{<1>}$	[1,4]
$v_{<2>}$	[1,4]
$w_{<1>}$	[1,4]
$w_{<2>}$	[1,4]

Fixed point!

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

Example

- Suppose **k >= 2**



$x_{<1>}$	[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 >= 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 >= 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;
   }
```

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

$x_{<1>}$	[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 >= 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;
   }
```

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

Fixed point!

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

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

Abstract Semantics for Partially Context-sensitive Analysis

- The abstract semantic operators for function calls:

$$bind^\sharp_{\mathtt{x}} : \mathbb{M}^\sharp \times \mathbb{I}^\sharp \times \mathbb{V}^\sharp \to \mathbb{M}^\sharp$$
$$bind^\sharp_{\mathtt{x}}(m^\sharp, \phi^\sharp, v^\sharp) = \bigsqcup_{\iota \in \phi^\sharp} m^\sharp[\langle \mathtt{x}, \iota \rangle \mapsto v^\sharp]$$

$$new\text{-}env^\sharp_{\mathtt{x}} : \mathbb{E}^\sharp \times \mathbb{I}^\sharp \to \mathbb{E}^\sharp$$
$$new\text{-}env^\sharp_{\mathtt{x}}(\sigma^\sharp, \phi^\sharp) = \sigma^\sharp[\mathtt{x} \mapsto \phi^\sharp]$$

$$push\text{-}context^\sharp : \mathbb{K}^\sharp \times \mathbb{L}^\sharp \times \mathbb{E}^\sharp \to \mathbb{K}^\sharp$$
$$push\text{-}context^\sharp(\kappa^\sharp, l, \sigma^\sharp) = \{\lceil \langle l, \sigma^\sharp \rangle \cdot K^\sharp \rceil_{\mathsf{k}} \mid K^\sharp \in \kappa^\sharp\}$$

Top-k contexts

$$tick^\sharp : \mathbb{I}^\sharp \to \mathbb{I}^\sharp$$
$$tick^\sharp(\phi^\sharp) = \{\lceil l \cdot K^\sharp \rceil_{\mathsf{k}} \mid K^\sharp \in \kappa^\sharp\}$$

Top-k contexts

- The abstract semantic operator for returns:

$$pop\text{-}context^\sharp : \mathbb{K}^\sharp \to \mathbb{L}^\sharp \times \mathbb{E}^\sharp \times \mathbb{K}^\sharp$$
$$pop\text{-}context^\sharp(\kappa^\sharp) = \bigsqcup\{pop^\sharp(K^\sharp) \mid K^\sharp \in \kappa^\sharp\}$$

$$\text{where } pop^\sharp(K^\sharp) = \begin{cases} \langle l^\sharp, \sigma^\sharp, K^{\sharp\prime} \rangle & \text{if } K^\sharp = \langle l^\sharp, \sigma^\sharp \rangle \cdot K^{\sharp\prime} \\ \langle \{l \mid \langle l, \_, \_, \_, \_ \rangle \hookrightarrow^\sharp \langle body(\mathtt{f}), \_, \_, \_, \_ \rangle\}, \lambda x.\{\epsilon\}, \epsilon \rangle & \text{if } K^\sharp = \epsilon \end{cases}$$

All possible call-sites for f

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