Static Analysis for Advanced Programming Features

Woosuk Lee

CSE 6049 Program Analysis

Hanyang University, Korea

Some slides are borrowed from https://github.com/prosyslab-classroom/is593-2020-spring/blob/master/slides/lecture7.pdf

Goal of This Lecture

- Learn how to design static analysis supporting more advanced features such as pointers and functions

Extended Language for Pointers

expression	E	::=	$\cdots$	as before
		|	`&x`	location of a variable
		|	**`malloc`**	location of newly allocated heap memory
		|	$*E$	dereference a location
statement	C	::=	$\cdots$	as before
		|	$*E := E$	indirect assignment
program	P	::=	C	

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

- A memory location is either a variable or a heap address
- A value is either an integer or a memory location or a program label (due to the goto statement)

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

Domain of Heap Addresses

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

- Set of the locations generated by the `malloc` expressions
- every `malloc` expression is assumed to have a unique number μ (allocation-site), writing $\texttt{malloc}_{\mu}$
- A heap address is represented as a pair of an allocation-site and a counter
- For example, the addresses of fresh locations from a malloc-site μ are $(\mu, 0), (\mu, 1), (\mu, 2), \ldots$

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

Transitional Concrete Semantics

- 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

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

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

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

Review: Transitional 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

- The abstract domains are Galois connected with the corresponding concrete domains:

$$\wp(\mathbb{L} \times \mathbb{M}) \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \mathbb{L} \to \mathbb{M}^\sharp \qquad\qquad \wp(\mathbb{M}) \underset{\alpha_M}{\overset{\gamma_M}{\leftrightarrows}} \mathbb{M}^\sharp,$$

where

$$\begin{aligned} \alpha(S) &= \{l \mapsto \alpha_M(\{m \mid (l,m) \in S\}) \mid l \in \mathbb{L}\}, \\ \gamma(s^\sharp) &= \{(l,m) \mid m \in \gamma_M(s^\sharp(l)), l \in \mathbb{L}\}. \end{aligned}$$

Abstraction of Memories

$$\mathbb{M}^\sharp = (\mathbb{X} \cup \mathbb{N}_{site}) \to \mathbb{V}^\sharp,$$

- The $\mathbb{X}$ *and* $\mathbb{N}_{site}$ are finite sets of variables and allocation sites in the input programs.

- abstract all heap locations located at a malloc site into a single abstract address (*allocation-site-based abstraction*)

- The abstraction and concretization functions:

$$\begin{array}{rcll}
\alpha_M(M)(\mathtt{x}) & = & \alpha_V(\{m(\mathtt{x}) \mid m \in M\}) & \textit{if } \mathtt{x} \in \mathbb{X} \\
\alpha_M(M)(\mu) & = & \alpha_V(\{m(\mu, n) \mid m \in M, n \in \mathbb{N}\}) & \textit{if } \mu \in \mathbb{N}_{site} \\
\gamma_M(M^\sharp) & = & \{m \in \mathbb{M} \mid \forall \mathtt{x} \in \mathbb{X} : m(\mathtt{x}) \in \gamma_V(M^\sharp(\mathtt{x})), & \\
 & & \quad \forall \mu \in \mathbb{N}_{site}, \forall n \in \mathbb{N} : m(\mu, n) \in \gamma_V(M^\sharp(\mu))\} &
\end{array}$$

Abstraction of Values

$$\wp(\mathbb{V}) \underset{\alpha_V}{\overset{\gamma_V}{\leftrightarrows}} \mathbb{V}^\sharp,$$

- Abstract kind-wise a set of values (integers, labels, and/or locations)

$$\wp(\mathbb{Z} \cup \mathbb{L} \cup \mathbb{A}) \quad \underset{\alpha_V}{\overset{\gamma_V}{\leftrightarrows}} \quad \mathbb{Z}^\sharp \times \mathbb{L}^\sharp \times \mathbb{A}^\sharp,$$

where

$$\begin{array}{rcl} \alpha_V(V) & = & (\alpha_Z(V \cap \mathbb{Z}), \alpha_L(V \cap \mathbb{L}), \alpha_A(V \cap \mathbb{A})), \\ \gamma_V(z^\sharp, l^\sharp, a^\sharp) & = & \gamma_Z(z^\sharp) \cup \gamma_L(l^\sharp) \cup \gamma_A(a^\sharp). \end{array}$$

Abstraction of Values

- The abstract integers, labels, and locations are Galois connected:

$$\wp(\mathbb{Z}) \underset{\alpha_Z}{\overset{\gamma_Z}{\leftrightarrows}} \mathbb{Z}^\sharp \qquad\qquad \wp(\mathbb{L}) \underset{\alpha_L}{\overset{\gamma_L}{\leftrightarrows}} \mathbb{L}^\sharp$$

$$\wp(\mathbb{A} = \mathbb{X} \cup \mathbb{N}_{site} \times \mathbb{N}) \underset{\alpha_A}{\overset{\gamma_A}{\leftrightarrows}} \mathbb{A}^\sharp = \wp(\mathbb{X} \cup \mathbb{N}_{site})$$

- Note that $\mathbb{X}$ *and* `malloc` *sites* $\mathbb{N}_{site}$ are finite; thus the power sets of those sets can be used as finite abstract domains.

- All the operators (join, widening, narrowing, …) are defined as kind-wise.

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 memory read operation $fetch^{\sharp}(M^{\sharp}, v^{\sharp})$ looks up the abstract memory entry at the abstract location $l^{\sharp} \in \wp(\mathbb{X} \cup \mathbb{N}_{site})$ of $v^{\sharp}$.

- Since the abstract location is a set of variables and malloc sites, the result is the join of all the entries:

$$\bigsqcup_{a \in l^{\sharp}} M^{\sharp}(a)$$

Safe Memory Write Operation

- The memory write operation $update^\sharp(M^\sharp, v_1^\sharp, v_2^\sharp)$ overwrites the memory entry (called ***strong update***) when the abstract target location $l^\sharp$ of $v_1^\sharp$ means a single concrete location.

- Otherwise, the update cannot overwrite the memory. Every entry in the abstract memory that constitutes the target abstract location $l^\sharp \in \wp(\mathbb{X} \cup \mathbb{N}_{site})$ must be joined with (called ***weak update***)

$$\bigsqcup_{a \in l^\sharp} M^\sharp[a \mapsto M^\sharp(a) \sqcup v_2^\sharp]$$

Example

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

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

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

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

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

Review: Transitional 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

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

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

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

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

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

Further Extended Language for Functions

Expression	E	::=	$\cdots$	as before
		$\mid$	`f`	function id
Statement	C	::=	$\cdots$	as before
		$\mid$	$E(E)$	function call
		$\mid$	**`return`**	return from call
Function	F	::=	$\texttt{f}(\texttt{x}) = C$	function definition
Program	P	::=	$F^+ C$	

Assumptions

- Every function has one parameter.
- Function parameter names are unique.
- No nested function definitions
- Variables other than function parameters are all global variables.
- Returning a value from a function is an assignment to a global variable.
- Function names can be stored in variables or passed on to other functions.

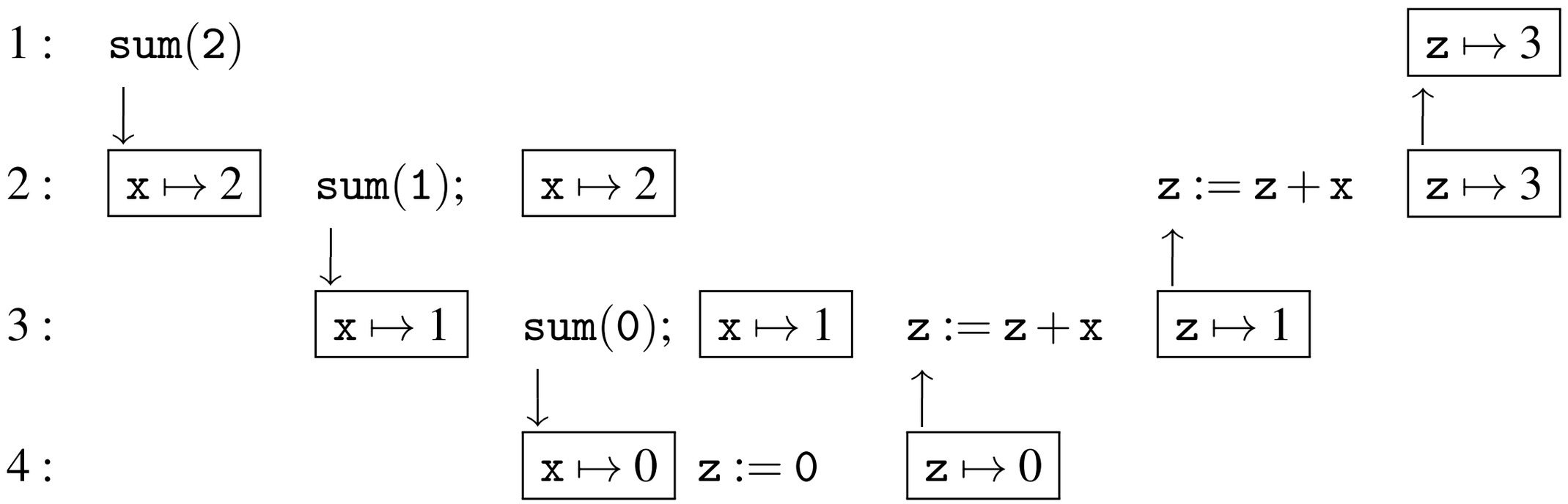
Semantics of Recursive Function Calls

- Example program

$$\texttt{sum(x)} \;=\; \textbf{if}(\texttt{x} = 0)\,\{\texttt{z} := 0;\ \textbf{return}\}\,\textbf{else}\,\{\texttt{sum}(\texttt{x}-1);\ \texttt{z} := \texttt{z}+\texttt{x};\ \textbf{return}\}$$

$$\texttt{sum}(2)$$

- Memory snapshots during the execution (↓:call, ↑:ret)



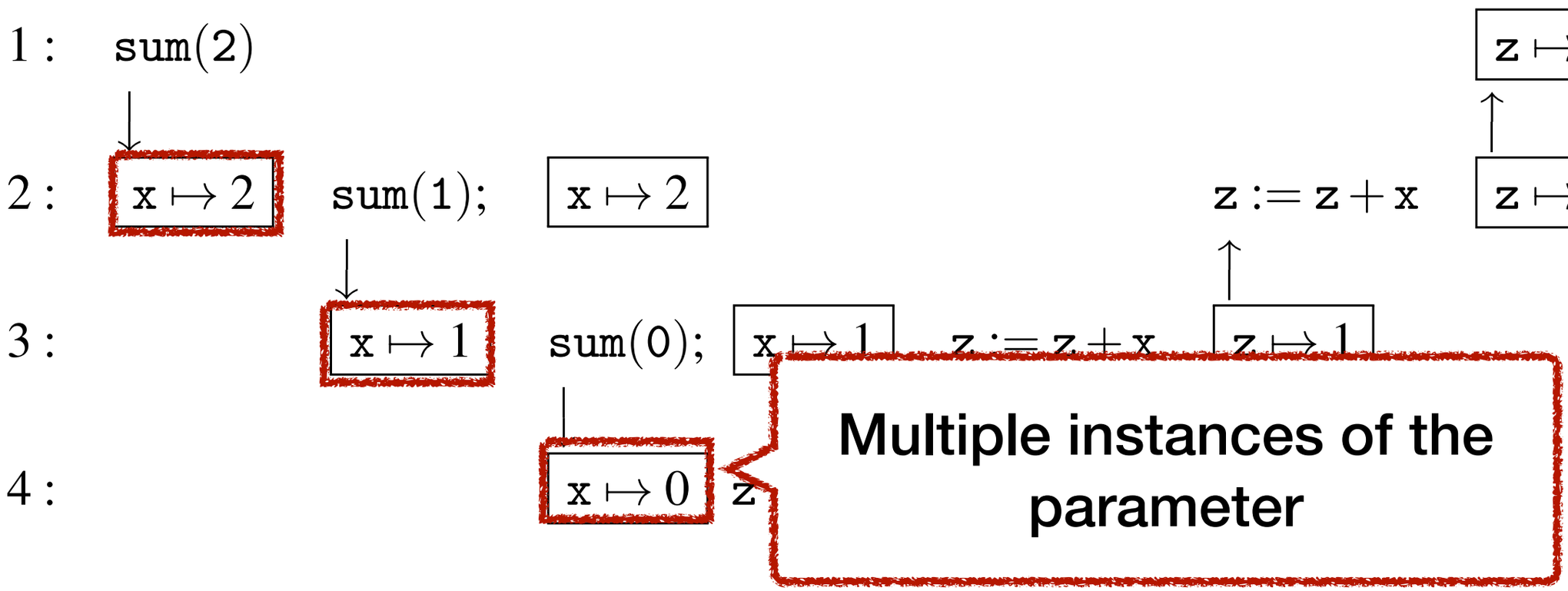
Semantics of Recursive Function Calls

- Example program

$$\texttt{sum(x)} = \textbf{if}(\texttt{x} = 0)\ \{\texttt{z} := 0;\ \textbf{return}\}\ \textbf{else}\ \{\texttt{sum}(\texttt{x} - 1);\ \texttt{z} := \texttt{z} + \texttt{x};\ \textbf{return}\}$$

$$\texttt{sum}(2)$$

- Memory snapshots during the execution (↓:call, ↑:ret)



Semantics of Recursive Function Calls

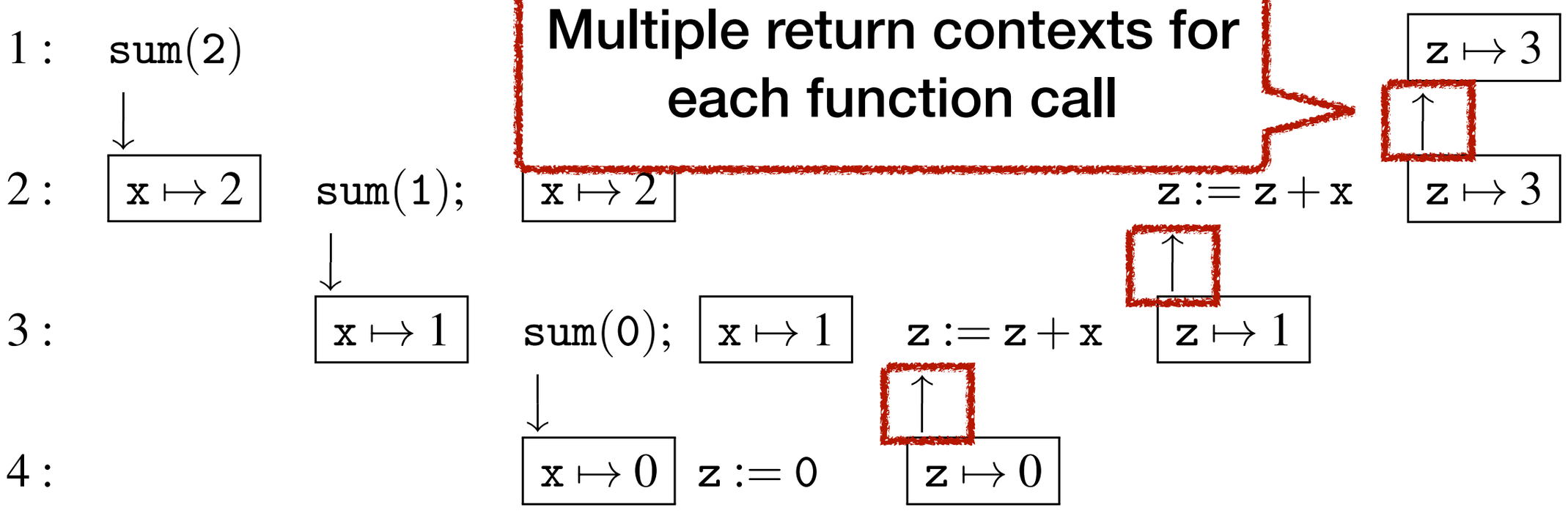
- Example program

$\texttt{sum(x)} = \textbf{if}(\texttt{x} = 0)\ \{\texttt{z} := 0;\ \textbf{return}\}\ \textbf{else}\ \{\texttt{sum}(\texttt{x} - 1);\ \texttt{z} := \texttt{z} + \texttt{x};\ \textbf{return}\}$

$\texttt{sum}(2)$

- Memory snapshots during the execution (↓:call, ↑:ret)

Multiple return contexts for each function call



Concrete Domains

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

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	1

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
<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

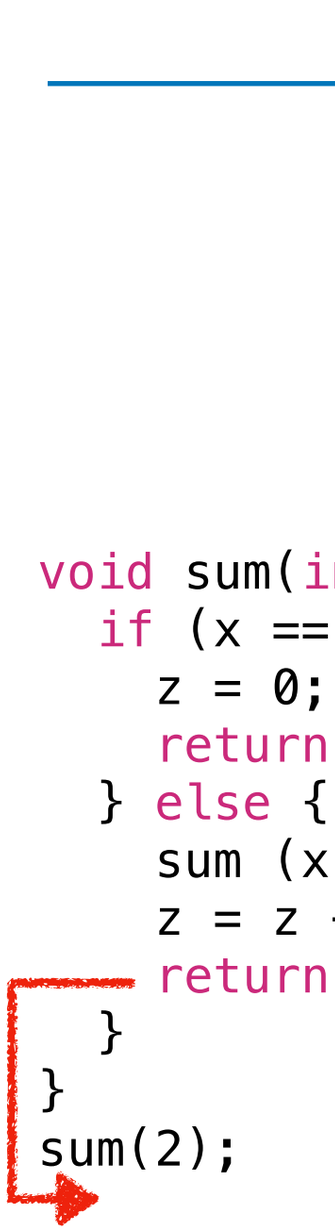
11

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

Concrete Semantics

- The state transition relation $\langle l, m, \sigma, \kappa, \phi \rangle \hookrightarrow \langle l', m', \sigma', \kappa', \phi' \rangle$

- $body(\texttt{f})$: label of body statement of function f

- $param(\texttt{f})$: formal parameter name of function f

Concrete Semantics

- The transition relation for function calls:

$$
\begin{aligned}
E_0(E_1) \quad : \quad \langle l, m, \sigma, \kappa, \phi \rangle \quad \hookrightarrow \quad & \langle \mathit{body}(\mathtt{f}), \\
& \mathit{bind}_{\mathtt{x}}(m, \phi', v), \qquad \text{parameter binding} \\
& \mathit{new\text{-}env}_{\mathtt{x}}(\sigma, \phi'), \quad \text{new environment} \\
& \mathit{push\text{-}context}(\kappa, \mathtt{next}(l), \sigma), \text{ new continuation} \\
& \phi' \rangle \\
\text{where } \mathtt{f} &= \mathit{eval}_{E_0}(m, \sigma) \\
\mathtt{x} &= \mathit{param}(\mathtt{f}) \\
v &= \mathit{eval}_{E_1}(m, \sigma) \\
\phi' &= \mathit{tick}(\phi)
\end{aligned}
$$

Concrete Semantics

$$
\begin{aligned}
\mathit{body}: &\quad \mathbb{F} \to \mathbb{L} \\
\mathit{bind}_{\mathtt{x}}: &\quad \mathbb{M} \times \mathbb{I} \times \mathbb{V} \to \mathbb{M} \\
\mathit{new\text{-}env}_{\mathtt{x}}: &\quad \mathbb{E} \times \mathbb{I} \to \mathbb{E} \\
\mathit{push\text{-}context}: &\quad \mathbb{K} \times \mathbb{L} \times \mathbb{E} \to \mathbb{K} \\
\mathit{pop\text{-}context}: &\quad \mathbb{K} \to \mathbb{L} \times \mathbb{E} \times \mathbb{K} \\
\mathit{tick}: &\quad \mathbb{I} \to \mathbb{I}
\end{aligned}
$$

where

$$
\begin{aligned}
\mathit{bind}_{\mathtt{x}}(m, \phi, v) &= m[\langle \mathrm{x}, \phi \rangle \mapsto v] \\
\mathit{new\text{-}env}_{\mathtt{x}}(\sigma, \phi) &= \sigma[\mathrm{x} \mapsto \phi] \\
\mathit{push\text{-}context}(\kappa, l, \sigma) &= \langle l, \sigma \rangle . \kappa \quad \text{(stack top } \langle l, \sigma \rangle \text{ and the rest } \kappa\text{)} \\
\mathit{pop\text{-}context}(\langle l, \sigma \rangle . \kappa) &= \langle l, \sigma, \kappa \rangle \\
\mathit{tick}(\phi) &= \phi' \qquad \text{(new } \phi'\text{)}
\end{aligned}
$$

Concrete Semantics

- The transition relation for function returns:

$$\texttt{return} \quad : \quad \langle l, m, \sigma, \kappa, \phi \rangle \quad \hookrightarrow \quad \langle l', m, \sigma', \kappa', \phi \rangle$$
$$\text{where } \langle l', \sigma', \kappa' \rangle = \mathit{pop\text{-}context}(\kappa)$$

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 state is Galois connected with $\wp(\mathbb{S})$

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

where

$$\wp(\mathbb{M}) \underset{\alpha_M}{\overset{\gamma_M}{\leftrightarrows}} \mathbb{M}^\sharp \qquad \wp(\mathbb{E}) \underset{\alpha_E}{\overset{\gamma_E}{\leftrightarrows}} \mathbb{E}^\sharp$$

$$\wp(\mathbb{K}) \underset{\alpha_K}{\overset{\gamma_K}{\leftrightarrows}} \mathbb{K}^\sharp \qquad \wp(\mathbb{I}) \underset{\alpha_I}{\overset{\gamma_I}{\leftrightarrows}} \mathbb{I}^\sharp$$

Memory and Environment Abstract Domains

- An example of a memory abstract domain is

$$\mathbb{M}^\sharp = (\mathbb{X} \times \mathbb{I}^\sharp) \to \mathbb{V}^\sharp,$$

  specified by $\wp(\mathbb{M}) \underset{\alpha_M}{\overset{\gamma_M}{\leftrightarrows}} \mathbb{M}^\sharp.$

- An example of an abstract environment domain is

$$\mathbb{E}^\sharp = \mathbb{X} \to \mathbb{I}^\sharp.$$

  specified by $\wp(\mathbb{E}) \underset{\alpha_E}{\overset{\gamma_E}{\leftrightarrows}} \mathbb{E}^\sharp.$

Abstract Value Domain

- An example of a value abstract domain:

$$\wp(\mathbb{Z} \cup \mathbb{L} \cup \mathbb{F}) \quad \underset{\alpha_V}{\overset{\gamma_V}{\leftrightarrows}} \quad \mathbb{Z}^\sharp \times \mathbb{L}^\sharp \times \mathbb{F}^\sharp,$$

where abstract integers, labels, and locations are

$$\wp(\mathbb{Z}) \quad \underset{\alpha_Z}{\overset{\gamma_Z}{\leftrightarrows}} \quad \mathbb{Z}^\sharp \qquad \wp(\mathbb{L}) \quad \underset{\alpha_T}{\overset{\gamma_T}{\leftrightarrows}} \quad \mathbb{L}^\sharp \qquad \wp(\mathbb{F}) \quad \underset{\alpha_F}{\overset{\gamma_F}{\leftrightarrows}} \quad \mathbb{F}$$

- Note that the variables $\mathbb{X}$ and the function names $\mathbb{F}$ are finite sets; thus the power sets those sets are already finite, eligible to be used as abstract domains.

Abstract State Transition

- The transition for function calls:

$$
\begin{aligned}
E_0(E_1) \quad : \quad \langle l, M^\sharp, \sigma^\sharp, \kappa^\sharp, \phi^\sharp \rangle \quad \hookrightarrow^\sharp \quad & \langle \mathit{body}(\mathtt{f}), \\
& \mathit{bind}^\sharp_{\mathtt{x}}(M^\sharp, \phi'^\sharp, v^\sharp), \quad \text{parameter binding} \\
& \mathit{new\text{-}env}^\sharp_{\mathtt{x}}(\sigma^\sharp, \phi'^\sharp), \quad \text{new environment} \\
& \mathit{push\text{-}context}^\sharp(\kappa^\sharp, \mathtt{next}(l), \sigma^\sharp), \ \text{new continuation} \\
& \phi'^\sharp \rangle \\
\text{where } \mathtt{f} \in & \mathit{eval}^\sharp_{E_0}(M^\sharp, \sigma^\sharp) \\
& \mathtt{x} = \mathit{param}(\mathtt{f}) \\
& v^\sharp = \mathit{eval}^\sharp_{E_1}(M^\sharp, \sigma^\sharp) \\
& \phi'^\sharp = \mathit{tick}^\sharp(\phi^\sharp)
\end{aligned}
$$

Abstract State Transition

- The transition for function returns:

$$\texttt{return} \quad : \quad \langle l, M^\sharp, \sigma^\sharp, \kappa^\sharp, \phi^\sharp \rangle \quad \hookrightarrow^\sharp \quad \langle l', M^\sharp, \sigma'^\sharp, \kappa'^\sharp, \phi^\sharp \rangle$$

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

Abstract Semantic Operators

- The abstract semantic operators are similar to their concrete correspondent except that they are defined over abstract domains:

$$\begin{aligned}
\mathit{bind}^\sharp_{\mathtt{x}} : &\quad \mathbb{M}^\sharp \times \mathbb{I}^\sharp \times \mathbb{V}^\sharp \to \mathbb{M}^\sharp \\
\mathit{new\text{-}env}^\sharp_{\mathtt{x}} : &\quad \mathbb{E}^\sharp \times \mathbb{I}^\sharp \to \mathbb{E}^\sharp \\
\mathit{push\text{-}context}^\sharp : &\quad \mathbb{K}^\sharp \times \mathbb{L}^\sharp \times \mathbb{E}^\sharp \to \mathbb{K}^\sharp \\
\mathit{pop\text{-}context}^\sharp : &\quad \mathbb{K}^\sharp \to \mathbb{L}^\sharp \times \mathbb{E}^\sharp \times \mathbb{K}^\sharp \\
\mathit{tick}^\sharp : &\quad \mathbb{I}^\sharp \to \mathbb{I}^\sharp \\
\mathit{eval}^\sharp_E : &\quad \mathbb{M}^\sharp \times \mathbb{E}^\sharp \to \mathbb{V}^\sharp \\
\mathit{update}_{\mathtt{x}}{}^\sharp : &\quad \mathbb{M}^\sharp \times \mathbb{V}^\sharp \times \mathbb{E}^\sharp \to \mathbb{M}^\sharp \\
\mathit{filter}^\sharp_B : &\quad \mathbb{M}^\sharp \times \mathbb{E}^\sharp \to \mathbb{M}^\sharp
\end{aligned}$$

Varying the Context Sensitivity

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

- How to abstract different continuations at each call?

- ***Context-insensitive***: all machine states at function calls are abstracted into a single abstract state

- ***Context-sensitive***: multiple abstract states to distinguish some differences of call contexts

Context-insensitivity

- The context sensitivity boils down to how we abstract $\mathbb{I}$:

$$\wp(\mathbb{I}) \underset{\alpha_I}{\overset{\gamma_I}{\leftrightarrows}} \mathbb{I}^\sharp$$

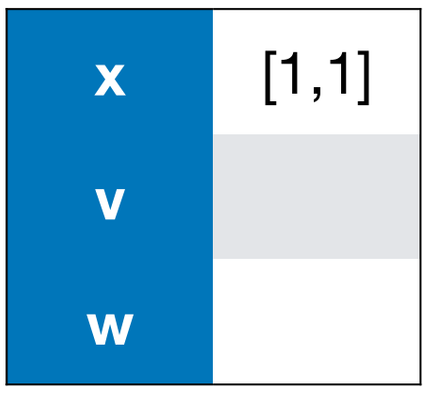
- If $\mathbb{I}^\sharp$ is a singleton set (i.e., $\mathbb{I}^\sharp = \{\cdot\}$, we don't differentiate the parameter instances in abstract semantics), then we follow the context insensitivity.

Informal 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;
}
```



Informal 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	
b	

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

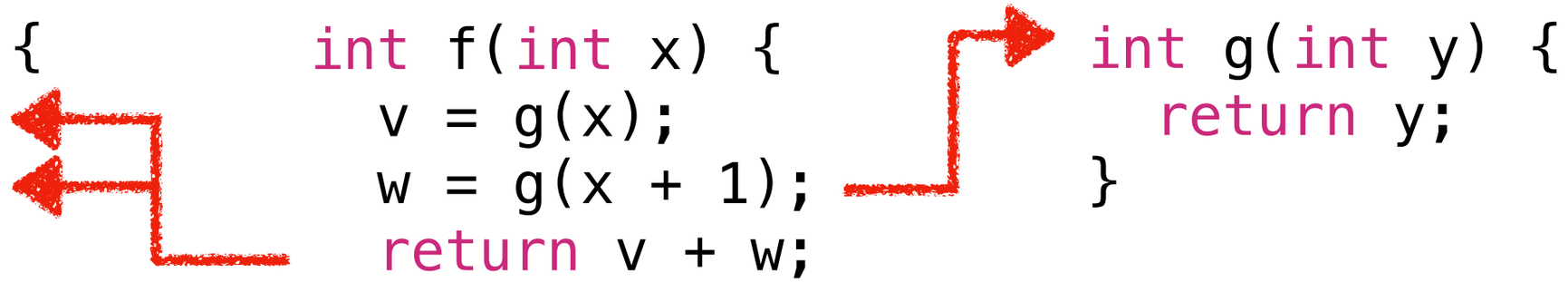
y	[1,1]

Informal 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]

Informal 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]

...

Informal 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]

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

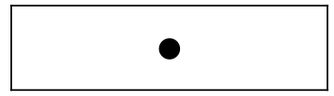
Example

```
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**

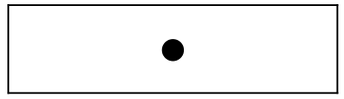
Example

```
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

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

Continuation



***Assume a proper widening is used**

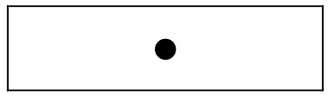
Example

```
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

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

Continuation



***Assume a proper widening is used**

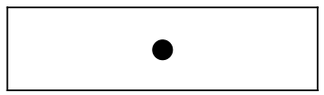
Example

```
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

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

Continuation



***Assume a proper widening is used**

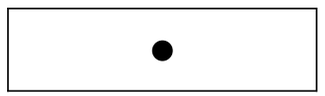
Example

```
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

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

Continuation



***Assume a proper widening is used**

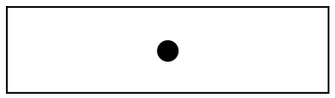
Example

```
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

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

Continuation



For brevity, we do not consider effects of the abstract filtering function.

***Assume a proper widening is used**

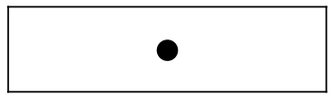
Example

```
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

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	[0,2]	[0,0]

Continuation



***Assume a proper widening is used**

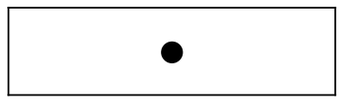
Example

```
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

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

Continuation



***Assume a proper widening is used**

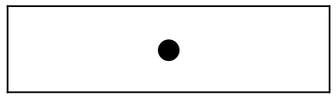
Example

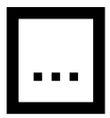
```
1: void sum(int x) {
2:   if (x == 0) {
3:     z = 0;
4:     return;
5:   } else {
6:     sum(x - 1);
7:     z = z + x;
8:     return;
9:   }
10: }
11: sum(2);
```

Memory

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]	[0,6]

Continuation





***Assume a proper widening is used**

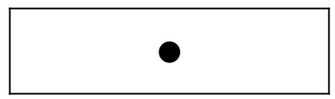
Example

```
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

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



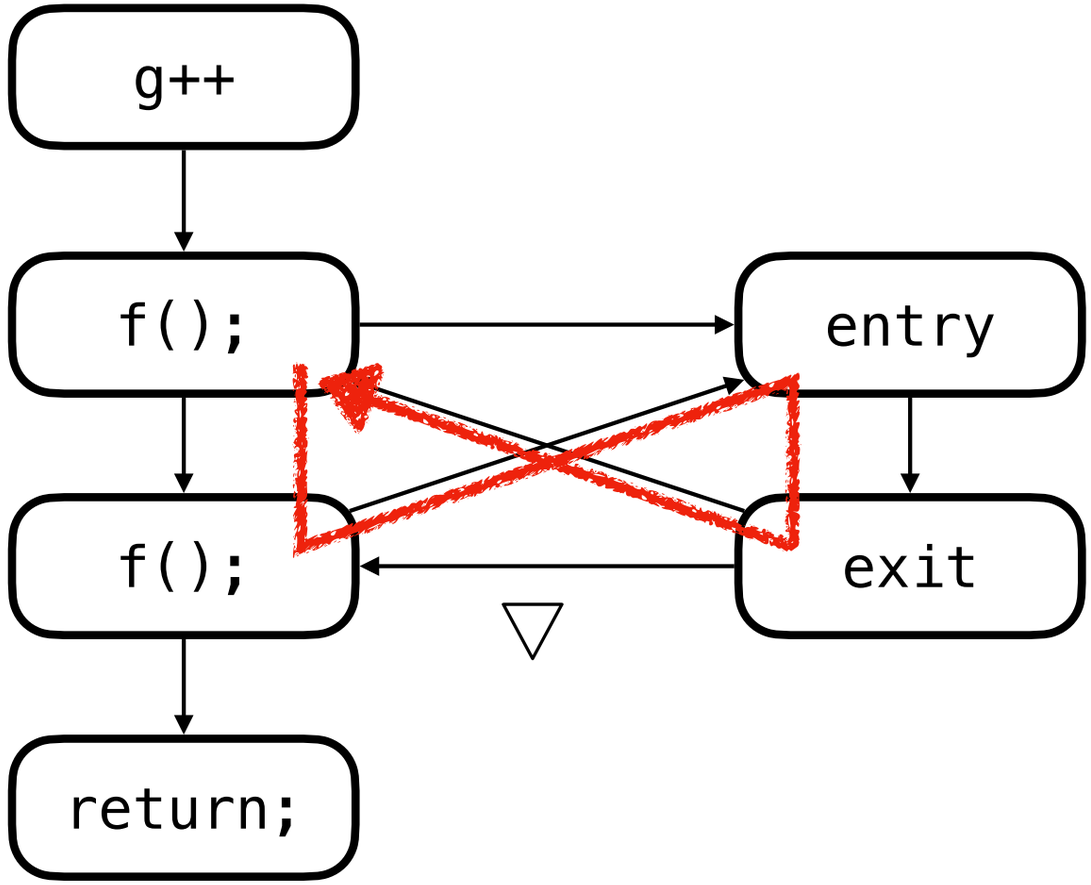
Fixed point!

***Assume a proper widening is used**

Spurious Cycles in Context-insensitive Analysis

- Even with non-recursive functions, spurious-cycle happens; widening is needed.

```
int main() {
  g++;
  f();  // non-recursive
  f();
  return;
}
```



Context-sensitive Analysis

- $\mathbb{I}^\sharp$ is an ordered sequence of call-sites (called *call-string*)

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

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

Continuation

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

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

Continuation

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

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

Continuation

6
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

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

Continuation

6
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

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

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

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

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

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

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

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

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

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

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 Analysis is Precise but Costly

- 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* call-strings (called *k*-CFA)
  - k = 0: ignore all contexts, i.e., context-insensitive
  - k = ∞: keep all contexts, i.e., fully context-sensitive
- In practice, k ≥ 2 is too expensive.

Summary

- Various choices of abstractions for pointers and functions
  - allocation-site-based abstraction, call-string-based context abstraction, …
- Choices can be made depending on desirable precision & analysis performance.