

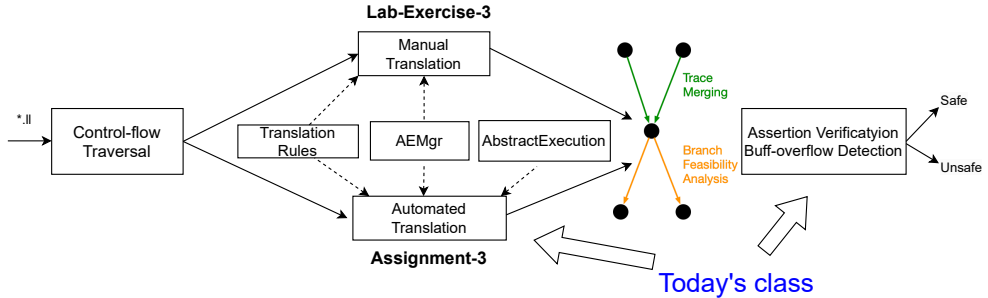
# Abstract Interpretation for Code Analysis and Verification

## (Week 9)

Yulei Sui

School of Computer Science and Engineering  
University of New South Wales, Australia

# Today's class



# Topological Order

## Analysis Order of Nodes on Control-Flow Graph

? How to analyze a program **free of loop**?

✓ Analyze each node **once** adhering to the **topological order** on the acyclic control-flow graph of the program.

# Topological Order

## Analysis Order of Nodes on Control-Flow Graph

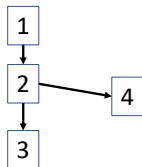
? How to analyze a program **free of loop**?

✓ Analyze each node **once** adhering to the **topological order** on the acyclic control-flow graph of the program.

A **topological order** of a graph  $G(V, E)$  is a linear ordering of its nodes such that for every directed edge  $a \rightarrow b$ , node  $a$  always precedes node  $b$  in the ordering.

- Must be a **direct acyclic graph** (DAG) and has at least one topo ordering.
- The ordering respects the **direction of edges**.

**Example of topological order:**



acyclic graph G

1 2 3 4 ✓

1 2 4 3 ✓

1 3 2 4 ✗

Valid/invalid topological order

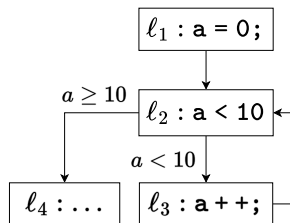
# Weak Topological Order

## Analysis Order of Nodes on Control-Flow Graph

? How to analyze a program **containing loops**?

✓ We can analyze a program containing loops adhering to the **weak topological order** (WTO) on its control flow graph.

What is the weak topological order?



Control Flow Graph

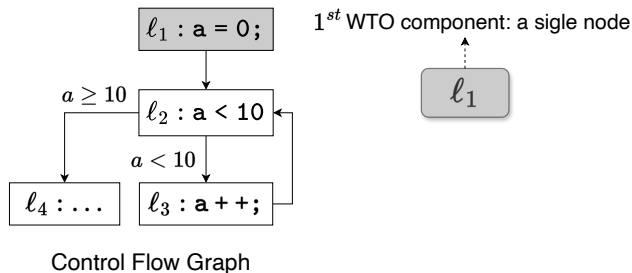
# Weak Topological Order

## Analysis Order of Nodes on Control-Flow Graph

? How to analyze a program **containing loops**?

✓ We can analyze a program containing loops adhering to the **weak topological order** (WTO) on its control flow graph.

What is the weak topological order?



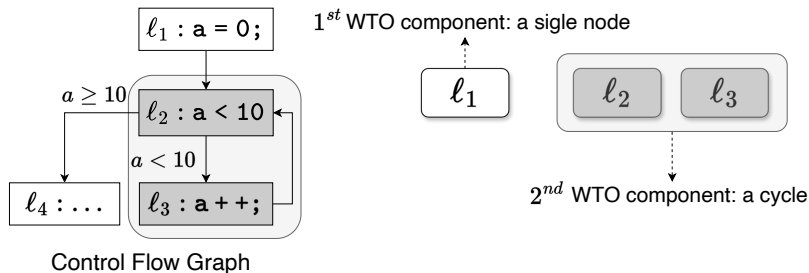
# Weak Topological Order

## Analysis Order of Nodes on Control-Flow Graph

? How to analyze a program **containing loops**?

✓ We can analyze a program containing loops adhering to the **weak topological order** (WTO) on its control flow graph.

What is the weak topological order?



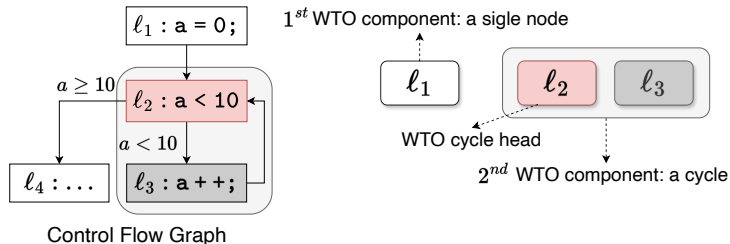
# Weak Topological Order

## Analysis Order of Nodes on Control-Flow Graph

? How to analyze a program **containing loops**?

✓ We can analyze a program containing loops adhering to the **weak topological order** (WTO) on its control flow graph.

What is the weak topological order?





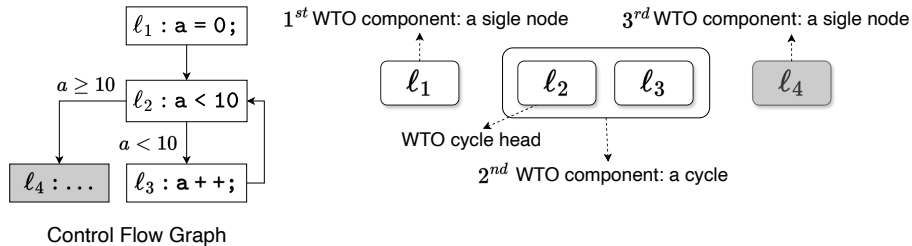
# Weak Topological Order

## Analysis Order of Nodes on Control-Flow Graph

? How to analyze a program **containing loops**?

✓ We can analyze a program containing loops adhering to the **weak topological order** (WTO) on its control flow graph.

What is the weak topological order?

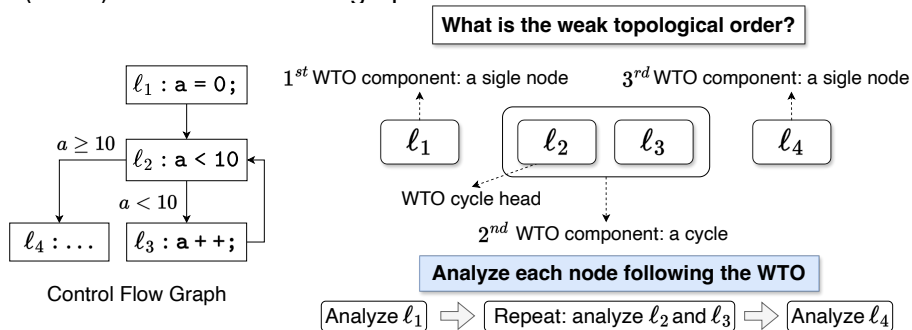


# Weak Topological Order

## Analysis Order of Nodes on Control-Flow Graph

? How to analyze a program **containing loops**?

✓ We can analyze a program containing loops adhering to the **weak topological order** (WTO) on its control flow graph.

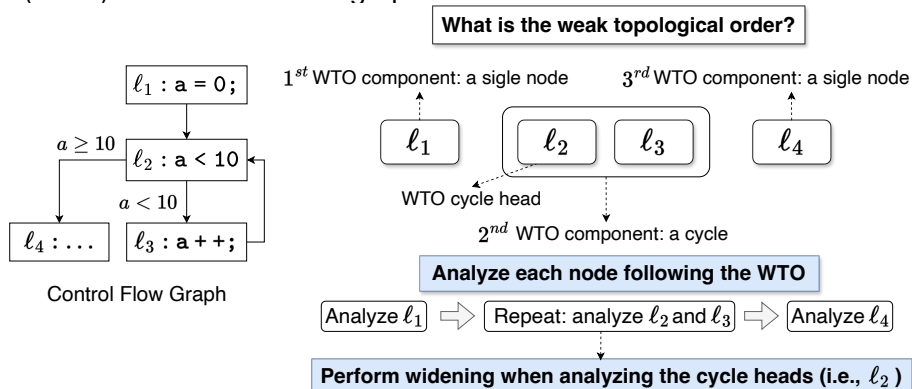


# Weak Topological Order

## Analysis Order of Nodes on Control-Flow Graph

? How to analyze a program **containing loops**?

✓ We can analyze a program containing loops adhering to the **weak topological order** (WTO) on its control flow graph.



# WTO, Widening and Narrowing

## Why Weak Topological Order?

- Handling cyclic dependencies
- Efficient fixed-point computation

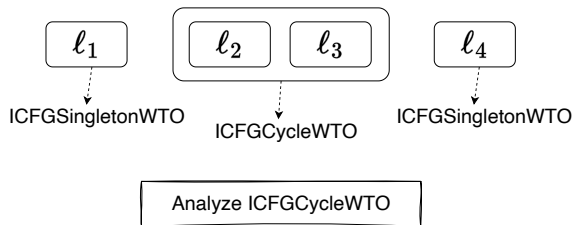
## Why Widening?

- Over-approximation
- Prevent non-termination

## Why Narrowing?

- Refine precision after widening converges
- The specific conditions or constraints used for narrowing:
  - Loop exit conditions (this course)
  - Type constraints (8-bit integer ranging from  $[-128, 127]$ )
  - Bounds from arithmetic operations If  $x = y + z$ , and  $y \in [1, 5]$  and  $z \in [2, 3]$ , then  $x \in [3, 8]$ . If widening gives  $[1, 10]$ , narrowing can refine this to  $[3, 8]$ .
  - User-specification (assertions and guard conditions)

# Widening and Narrowing

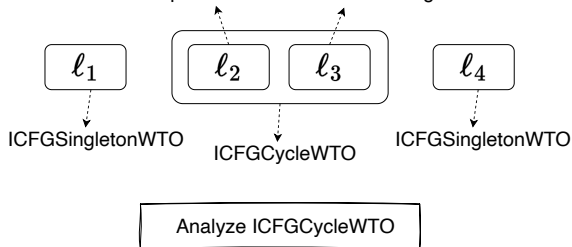


## Algorithm 1: Abstract execution guided by WTO (part 2)

```
1 Function handleCycleWTO(cycle):  
2   cycle_head := cycle→head()→node();  
3   increasing := true;  
4   cur_iter := 0;  
5   while true do  
6     cur_iter++;  
7     if cur_iter ≥ Options :: WidenDelay() then  
8       prev_head_state := postAbsTrace[cycle_head];  
9       handleSingletonWTO(cycle→head());  
10      cur_head_state := postAbsTrace[cycle_head];  
11      if increasing then  
12        postAbsTrace[cycle_head] := prev_head_state.widen(cur_head_state);  
13        if postAbsTrace[cycle_head] == prev_head_state then  
14          increasing := false;  
15          continue;  
16        else  
17          postAbsTrace[cycle_head] := prev_head_state.narrow(cur_head_state);  
18          if postAbsTrace[cycle_head] == prev_head_state then  
19            break;  
20      else  
21        handleSingletonWTO(cycle→head());  
22      handleWTOComponents(cycle→getWTOComponents());
```

# Widening and Narrowing

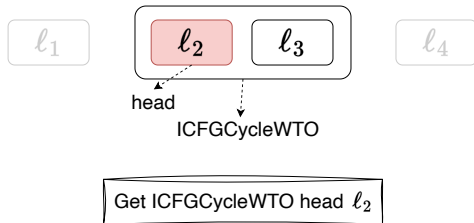
Sub WTO Components: each is an ICFGSingletonWTO



## Algorithm 2: Abstract execution guided by WTO (part 2)

```
1 Function handleCycleWTO(cycle):
2   cycle_head := cycle→head()→node();
3   increasing := true;
4   cur_iter := 0;
5   while true do
6     cur_iter++;
7     if cur_iter ≥ Options :: WideningDelay() then
8       prev_head_state := postAbsTrace[cycle_head];
9       handleSingletonWTO(cycle→head());
10      cur_head_state := postAbsTrace[cycle_head];
11      if increasing then
12        postAbsTrace[cycle_head] := prev_head_state.widen(cur_head_state);
13        if postAbsTrace[cycle_head] == prev_head_state then
14          increasing := false;
15          continue;
16      else
17        postAbsTrace[cycle_head] := prev_head_state.narrow(cur_head_state);
18        if postAbsTrace[cycle_head] == prev_head_state then
19          break;
20    else
21      handleSingletonWTO(cycle→head());
22    handleWTOComponents(cycle→getWTOComponents());
```

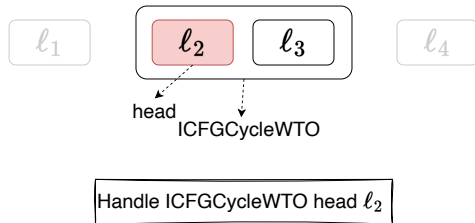
# Weak Topological Order



**Algorithm 3:** Abstract execution guided by WTO (part 2)

```
1 Function handleCycleWTO(cycle):  
2   cycle_head := cycle→head()→node();  
3   increasing := true;  
4   cur_iter := 0;  
5   while true do  
6     cur_iter++;  
7     if cur_iter ≥ Options::WidenDelay() then  
8       prev_head_state := postAbsTrace[cycle_head];  
9       handleSingletonWTO(cycle→head());  
10      cur_head_state := postAbsTrace[cycle_head];  
11      if increasing then  
12        postAbsTrace[cycle_head] := prev_head_state.widen(cur_head_state);  
13        if postAbsTrace[cycle_head] == prev_head_state then  
14          increasing := false;  
15          continue;  
16      else  
17        postAbsTrace[cycle_head] := prev_head_state.narrow(cur_head_state);  
18        if postAbsTrace[cycle_head] == prev_head_state then  
19          break;  
20      else  
21        handleSingletonWTO(cycle→head());  
22      handleWTOComponents(cycle→getWTOComponents());
```

# Weak Topological Order

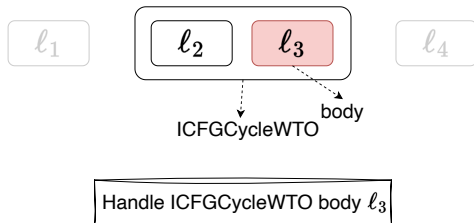


**Algorithm 4:** Abstract execution guided by WTO (part 2)

```
1 Function handleCycleWTO(cycle):  
2   cycle_head := cycle→head()→node();  
3   increasing := true;  
4   cur_iter := 0;  
5   while true do  
6     cur_iter++;  
7     if cur_iter ≥ Options :: WidenDelay() then  
8       prev_head_state := postAbsTrace[cycle_head];  
9       handleSingletonWTO(cycle→head());  
10      cur_head_state := postAbsTrace[cycle_head];  
11      if increasing then  
12        postAbsTrace[cycle_head] := prev_head_state.widen(cur_head_state);  
13        if postAbsTrace[cycle_head] == prev_head_state then  
14          increasing := false;  
15          continue;  
16      else  
17        postAbsTrace[cycle_head] := prev_head_state.narrow(cur_head_state);  
18        if postAbsTrace[cycle_head] == prev_head_state then  
19          break;  
20      else  
21        handleSingletonWTO(cycle→head());  
22      handleWTOComponents(cycle→getWTOComponents());
```



# Widening and Narrowing

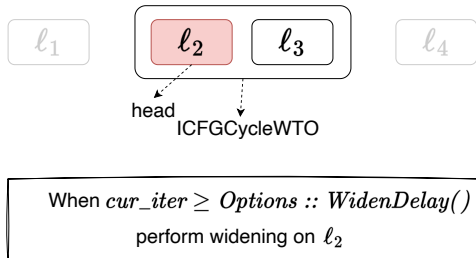


## Algorithm 5: Abstract execution guided by WTO (part 2)

```
1 Function handleCycleWTO(cycle):  
2   cycle_head := cycle→head()→node();  
3   increasing := true;  
4   cur_iter := 0;  
5   while true do  
6     cur_iter++;  
7     if cur_iter ≥ Options :: WidenDelay() then  
8       prev_head_state := postAbsTrace[cycle_head];  
9       handleSingletonWTO(cycle→head());  
10      cur_head_state := postAbsTrace[cycle_head];  
11      if increasing then  
12        postAbsTrace[cycle_head] := prev_head_state.widen(cur_head_state);  
13        if postAbsTrace[cycle_head] == prev_head_state then  
14          increasing := false;  
15          continue;  
16        else  
17          postAbsTrace[cycle_head] := prev_head_state.narrow(cur_head_state);  
18          if postAbsTrace[cycle_head] == prev_head_state then  
19            break;  
20      else  
21        handleSingletonWTO(cycle→head());  
22      handleWTOComponents(cycle→getWTOComponents());
```

**Note:** getWTOComponents returns Cycle WTO body, i.e.,  $l_3$

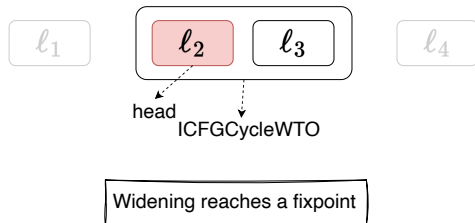
# Widening and Narrowing



## Algorithm 6: Abstract execution guided by WTO (part 2)

```
1 Function handleCycleWTO(cycle):  
2   cycle_head := cycle→head()→node();  
3   increasing := true;  
4   cur_iter := 0;  
5   while true do  
6     cur_iter++;  
7     if cur_iter ≥ Options :: WidenDelay() then  
8       prev_head_state := postAbsTrace[cycle_head];  
9       handleSingletonWTO(cycle→head());  
10      cur_head_state := postAbsTrace[cycle_head];  
11      if increasing then  
12        postAbsTrace[cycle_head] := prev_head_state.widen(cur_head_state);  
13        if postAbsTrace[cycle_head] == prev_head_state then  
14          increasing := false;  
15          continue;  
16        else  
17          postAbsTrace[cycle_head] := prev_head_state.narrow(cur_head_state);  
18          if postAbsTrace[cycle_head] == prev_head_state then  
19            break;  
20      else  
21        handleSingletonWTO(cycle→head());  
22      handleWTOComponents(cycle→getWTOComponents());
```

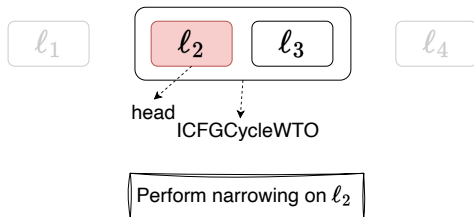
# Widening and Narrowing



**Algorithm 7:** Abstract execution guided by WTO (part 2)

```
1 Function handleCycleWTO(cycle):  
2   cycle_head := cycle→head()→node();  
3   increasing := true;  
4   cur_iter := 0;  
5   while true do  
6     cur_iter++;  
7     if cur_iter ≥ Options :: WidenDelay() then  
8       prev_head_state := postAbsTrace[cycle_head];  
9       handleSingletonWTO(cycle→head());  
10      cur_head_state := postAbsTrace[cycle_head];  
11      if increasing then  
12        postAbsTrace[cycle_head] := prev_head_state.widen(cur_head_state);  
13        if postAbsTrace[cycle_head] == prev_head_state then  
14          increasing := false;  
15          continue;  
16        else  
17          postAbsTrace[cycle_head] := prev_head_state.narrow(cur_head_state);  
18          if postAbsTrace[cycle_head] == prev_head_state then  
19            break;  
20      else  
21        handleSingletonWTO(cycle→head());  
22      handleWTOComponents(cycle→getWTOComponents());
```

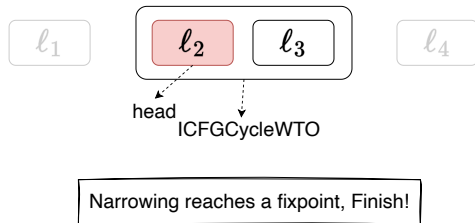
# Widening and Narrowing



**Algorithm 8:** Abstract execution guided by WTO (part 2)

```
1 Function handleCycleWTO(cycle):  
2   cycle_head := cycle→head()→node();  
3   increasing := true;  
4   cur_iter := 0;  
5   while true do  
6     cur_iter++;  
7     if cur_iter ≥ Options :: WidenDelay() then  
8       prev_head_state := postAbsTrace[cycle_head];  
9       handleSingletonWTO(cycle→head());  
10      cur_head_state := postAbsTrace[cycle_head];  
11      if increasing then  
12        postAbsTrace[cycle_head] := prev_head_state.widen(cur_head_state);  
13        if postAbsTrace[cycle_head] == prev_head_state then  
14          increasing := false;  
15          continue;  
16      else  
17        postAbsTrace[cycle_head] := prev_head_state.narrow(cur_head_state);  
18        if postAbsTrace[cycle_head] == prev_head_state then  
19          break;  
20      else  
21        handleSingletonWTO(cycle→head());  
22      handleWTOComponents(cycle→getWTOComponents());
```

# Widening and Narrowing



**Algorithm 9:** Abstract execution guided by WTO (part 2)

```
1 Function handleCycleWTO(cycle):  
2   cycle.head := cycle→head()→node();  
3   increasing := true;  
4   cur_iter := 0;  
5   while true do  
6     cur_iter++;  
7     if cur_iter ≥ Options :: WidenDelay() then  
8       prev_head_state := postAbsTrace[cycle.head];  
9       handleSingletonWTO(cycle→head());  
10      cur_head_state := postAbsTrace[cycle.head];  
11      if increasing then  
12        postAbsTrace[cycle.head] := prev_head_state.widen(cur_head_state);  
13        if postAbsTrace[cycle.head] == prev_head_state then  
14          increasing := false;  
15          continue;  
16      else  
17        postAbsTrace[cycle.head] := prev_head_state.narrow(cur_head_state);  
18        if postAbsTrace[cycle.head] == prev_head_state then  
19          break;  
20      else  
21        handleSingletonWTO(cycle→head());  
22      handleWTOComponents(cycle→getWTOComponents());
```

# Abstract Interpretation on SVFIR

## Week 9

Yulei Sui

School of Computer Science and Engineering

University of New South Wales, Australia

# Abstract Interpretation on Pointer-Free SVFIR

## Interval Domain

- For simplicity, let's first consider abstract execution on a pointer-free language.
- This means there are no operations for memory allocation (like  $p = \text{alloc}_o$ ) or for indirect memory accesses (such as  $p = *q$  or  $*p = q$ ).
- Here are the pointer-free SVFSTMTs and their C-like forms:

SVFSTMT	C-Like form
CONSTMT	$\ell : p = c$
COPYSTMT	$\ell : p = q$
BINARYSTMT	$\ell : r = p \otimes q$
PHISTMT	$\ell : r = \text{phi}(p_1, p_2, \dots, p_n)$
SEQUENCE	$\ell_1; \ell_2$
BRANCHSTMT	$\ell_1 : \text{if}(x < c) \text{ then } \ell_2 \text{ else } \ell_3$

# Abstract Interpretation on Pointer-Free SVFIR

## Interval Domain

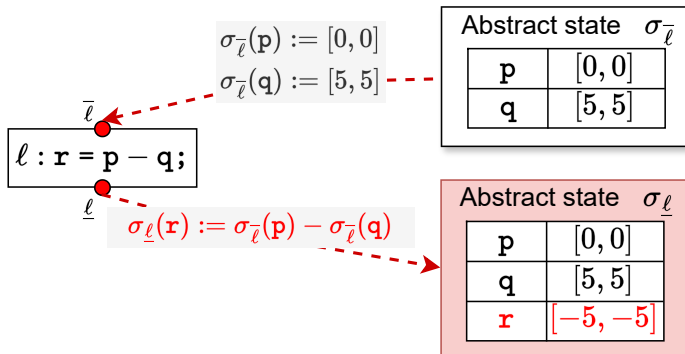
Let's use the *Interval* abstract domain to update  $\sigma$  based on the following rules for different SVFSTMT:

SVFSTMT	C-Like form	Abstract Execution Rule
CONSTMT	$\ell : p = c$	$\sigma_{\underline{\ell}}(p) := [c, c]$
COPYSTMT	$\ell : p = q$	$\sigma_{\underline{\ell}}(p) := \sigma_{\underline{\ell}}(q)$
BINARYSTMT	$\ell : r = p \otimes q$	$\sigma_{\underline{\ell}}(r) := \sigma_{\underline{\ell}}(p) \hat{\otimes} \sigma_{\underline{\ell}}(q)$
PHISTMT	$\ell : r = \text{phi}(p_1, p_2, \dots, p_n)$	$\sigma_{\underline{\ell}}(r) := \bigsqcup_{i=1}^n \sigma_{\underline{\ell}}(p_i)$
SEQUENCE	$\ell_1; \ell_2$	$\forall v \in \mathbb{V}, \sigma_{\underline{\ell}_2}(v) \supseteq \sigma_{\underline{\ell}_1}(v)$
BRANCHSTMT	$\ell_1 : \text{if}(x < c) \text{ then } \ell_2 \text{ else } \ell_3$	$\begin{aligned} \sigma_{\underline{\ell}_2}(x) &:= \sigma_{\underline{\ell}_1}(x) \sqcap [-\infty, c - 1], \text{ if } \sigma_{\underline{\ell}_1}(x) \sqcap [-\infty, c - 1] \neq \perp \\ \sigma_{\underline{\ell}_3}(x) &:= \sigma_{\underline{\ell}_1}(x) \sqcap [c, +\infty], \text{ if } \sigma_{\underline{\ell}_1}(x) \sqcap [c, +\infty] \neq \perp \end{aligned}$



# Abstract Interpretation on BINARYSTMT

SVFSTMT	C-Like form	Abstract Execution Rule
BINARYSTMT	$\ell : r = p \otimes q$	$\sigma_{\underline{\ell}}(r) := \sigma_{\bar{\ell}}(p) \hat{\otimes} \sigma_{\bar{\ell}}(q)$

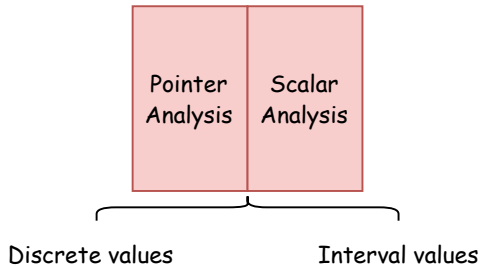


# Abstract Interpretation in the Presence of Pointers

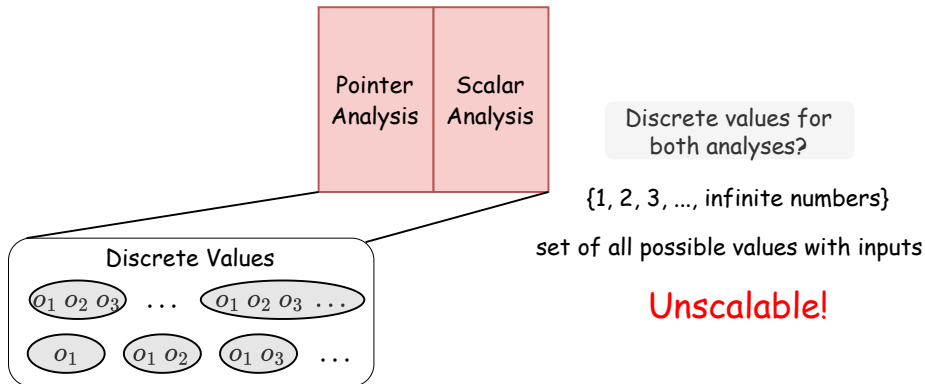
- SVFIR in the presence of pointers contain pointer-related statements including ADDRSTMT, GEPSTMT, LOADSTMT and STORESTMT.
- Abstract interpretation needs to be performed on **a combined domain of intervals and addresses**.

SVFSTMT	C-Like form
CONSTMT	$\ell : p = c$
COPYSTMT	$\ell : p = q$
BINARYSTMT	$\ell : r = p \otimes q$
PHISTMT	$\ell : r = \text{phi}(p_1, p_2, \dots, p_n)$
SEQUENCE	$\ell_1; \ell_2$
BRANCHSTMT	$\ell_1 : \text{if}(x < c) \text{ then } \ell_2 \text{ else } \ell_3$
ADDRSTMT	$\ell : p = \text{alloc}$
GEPSTMT	$\ell : p = \&(q \rightarrow i) \text{ or } p = \&q[i]$
LOADSTMT	$\ell : p = *q$
STORESTMT	$\ell : *p = q$

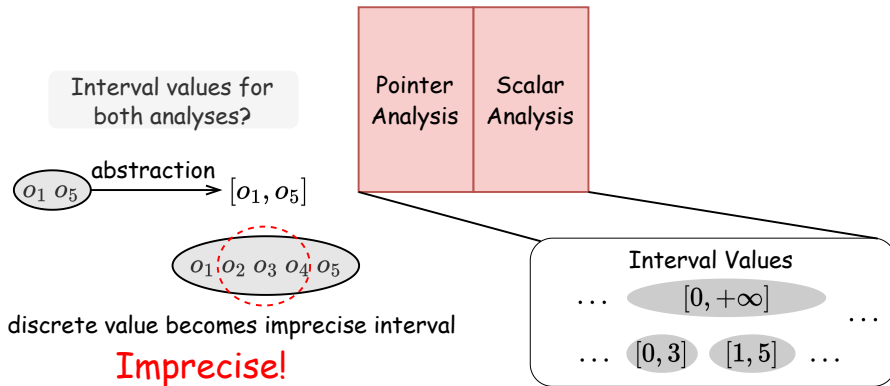
# Combined Analysis



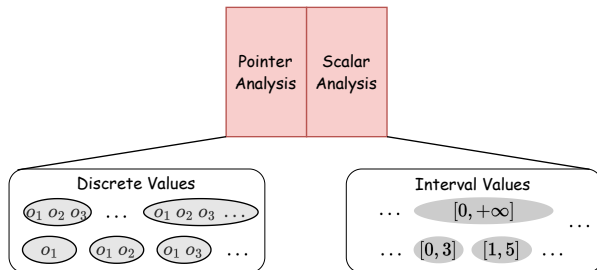
# Combined Analysis Using Discrete Values



# Combined Analysis Using Interval Values



# Abstract Interpretation Over a Combined Domain



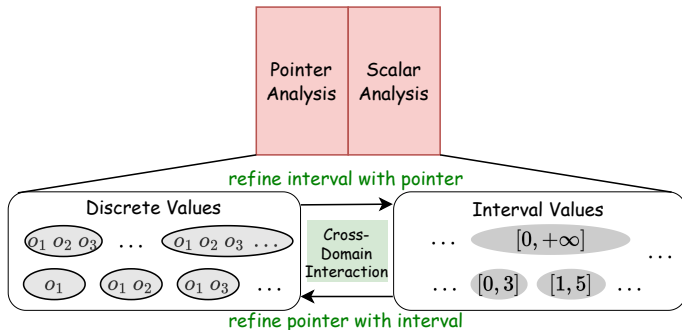
$p = \text{malloc}(\textcolor{red}{m} * \text{sizeof}(\text{int}));$  //  $p$  points to an array of size  $m$

$q = \text{malloc}(n * \text{sizeof}(\text{int}));$  //  $q$  points to an array of size  $n$

$\textcolor{red}{m} = r[\textcolor{blue}{i}];$

- The discrete values for points-to set of  $p$ ,  $q$  depend on interval values of  $m$  and  $n$ .
- The interval value of  $m$  depends on the pointer aliasing between  $p$ ,  $q$  and  $\&r[i]$ .
- Cyclic dependency between two domains requiring a bi-directional refinement. (variables highlighted in  $\textcolor{blue}{blue}$  and  $\textcolor{red}{red}$  denote the discrete values and interval values dependent),

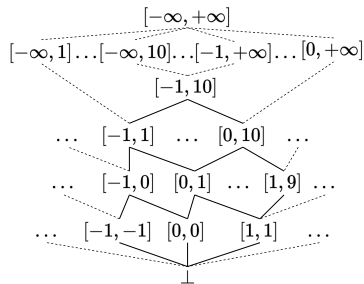
# Abstract Interpretation Over a Combined Domain



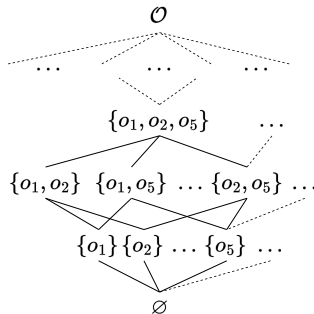
We require **a combination of interval and memory address domains** to precisely and efficiently perform abstract execution on SVFIR in the presence of pointers.

# Abstract Interpretation over Interval and MemAddress Domains

## A Combined Domain of Intervals and Discrete Memory Addresses



*Interval* domain for scalar variables



*MemAddress* domain for discrete memory address values



# SVF Program Variables (SVFVar)

Program Variables	Domain	Meanings
SVFVar	$\mathbb{V} = \mathbb{P} \cup \mathbb{O}$	Program Variables
ValVar	$\mathbb{P}$	Top-level variables (scalars and pointers)
ObjVar	$\mathbb{O} = \mathbb{S} \cup \mathbb{G} \cup \mathbb{H} \cup \mathbb{C}$	Memory Objects (constant data, stack, heap, global) (function objects are considered as global objects)
FIObjVar	$\mathbf{o} \in (\mathbb{S} \cup \mathbb{G} \cup \mathbb{H})$	A single (base) memory object
GepObjVar	$\mathbf{o}_i \in (\mathbb{S} \cup \mathbb{G} \cup \mathbb{H}) \times \mathbb{P}$	$i$ -th subfield/element of an (aggregate) object
ConstantData	$\mathbb{C}$	Constant data (e.g., numbers and strings)
Program Statement	$\ell \in \mathbb{L}$	Statements labels

# Abstract Trace for The Combined Domain

- For top-level variables  $\mathbb{P}$ , we use  $\sigma \in \mathbb{L} \times \mathbb{P} \rightarrow Interval \times MemAddress$  to track the memory addresses or interval values of these variables.
- For memory objects  $\mathbb{O}$ , we use  $\delta \in \mathbb{L} \times \mathbb{O} \rightarrow Interval \times MemAddress$  to track their abstract values

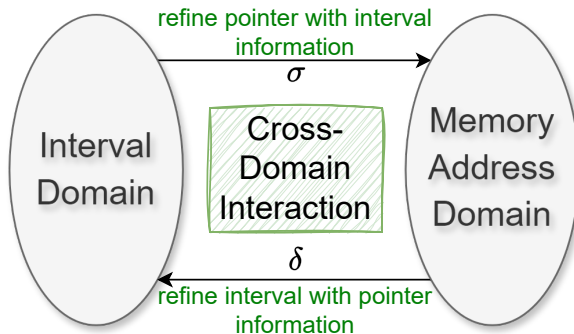
	Notation	Domain	Data Structure Implementation
Abstract trace	$\sigma$	$\mathbb{L} \times \mathbb{P} \rightarrow Interval \times MemAddress$	$preAbsTrace, postAbsTrace$
	$\delta$	$\mathbb{L} \times \mathbb{O} \rightarrow Interval \times MemAddress$	
Abstract state	$\sigma_L$	$\mathbb{P} \rightarrow Interval \times MemAddress$	$AbstractState.varToAbsVal$
	$\delta_L$	$\mathbb{O} \rightarrow Interval \times MemAddress$	$AbstractState.addrToAbsVal$
Abstract value	$\sigma_L(p)$	$Interval \times MemAddress$	$AbstractValue$
	$\delta_L(o)$		

- *Interval* is used for tracking the interval value of **scalar variables**  $\mathbb{P}$ .
- *MemAddress* is used for tracking the memory addresses of **memory address variables**  $\mathbb{O}$ .

# Implementation of Abstract State and Abstract Trace

- For a program point  $L$ , the abstract state is an instance of the class named *AEState*, consisting of:
  - Top-level variable, *varToAbsVal* :  $\sigma_L \in \mathbb{P} \rightarrow Interval \times MemAddress$
  - Memory object, *addrToAbsVal* :  $\delta_L \in MemAddress \rightarrow Interval \times MemAddress$
- The abstract trace has two maps, *preAbsTrace* and *postAbsTrace*, which maintains abstract states before and after each *ICFGNode* respectively.
  - For an *ICFGNode*  $\ell$ , *preAbsTrace*( $\ell$ ) retrieves the abstract state  $\langle \sigma_{\ell}, \delta_{\ell} \rangle$ , and *postAbsTrace*( $\ell$ ) represents  $\langle \sigma_{\ell}, \delta_{\ell} \rangle$ .
  - For each abstract state  $\langle \sigma_{\ell}, \delta_{\ell} \rangle$  we use *as*[*varId*] to operate  $\sigma_{\ell}$  and use *storeValue* and *loadValue* to operate  $\delta_{\ell}$ .
  - Each variable's *AbstractValue* (e.g., *as*[*VarId*]) is initialized as  $\perp$  in an *AbstractState* before assigned a new value.
  - Each *AbstractValue* (e.g., *as*[*VarId*]) is a 2-element tuple consisting of an *interval* *as*[*VarId*].getInterval() and an address set *as*[*VarId*].getAddrs().
  - Print out *SVFVars* and their *AbstractValues* in an *AbstractState* by invoking *as.printAbstractState()*

# Abstract Trace for The Combined Domain



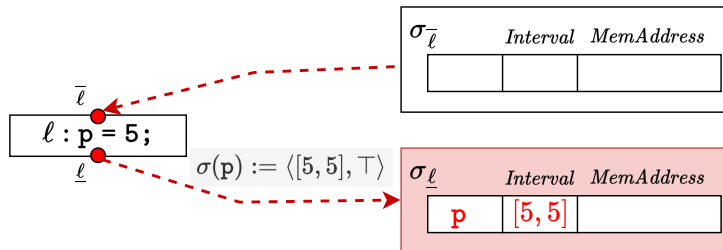
# Abstract Execution Rules on SVFIR in the Presence of Pointers

Now let's use the  $Interval \times MemAddress$  abstract domain to update  $\sigma$  and  $\delta$  based on the following rules for different SVFSTMT:

SVFSTMT	C-Like form	Abstract Execution Rule
CONSTMT	$\ell : p = c$	$\sigma_{\underline{\ell}}(p) := \langle [c, c], \perp \rangle$
COPYSTMT	$\ell : p = q$	$\sigma_{\underline{\ell}}(p) := \sigma_{\underline{\ell}}(q)$
BINARYSTMT	$\ell : r = p \otimes q$	$\sigma_{\underline{\ell}}(r) := \sigma_{\underline{\ell}}(p) \hat{\otimes} \sigma_{\underline{\ell}}(q)$
PHISTMT	$\ell : r = \text{phi}(p_1, p_2, \dots, p_n)$	$\sigma_{\underline{\ell}}(r) := \bigsqcup_{i=1}^n \sigma_{\underline{\ell}}(p_i)$
BRANCHSTMT	$\ell_1 : \text{if}(x < c) \text{ then } \ell_2 \text{ else } \ell_3$	$\begin{aligned} \sigma_{\underline{\ell}_2}(x) &:= \sigma_{\underline{\ell}_1}(x) \sqcap [-\infty, c - 1], \text{ if } \sigma_{\underline{\ell}_1}(x) \sqcap [-\infty, c - 1] \neq \perp \\ \sigma_{\underline{\ell}_3}(x) &:= \sigma_{\underline{\ell}_1}(x) \sqcap [c, +\infty], \text{ if } \sigma_{\underline{\ell}_1}(x) \sqcap [c, +\infty] \neq \perp \end{aligned}$
SEQUENCE	$\ell_1; \ell_2$	$\delta_{\underline{\ell}_2} \sqsupseteq \delta_{\underline{\ell}_1}, \sigma_{\underline{\ell}_2} \sqsupseteq \sigma_{\underline{\ell}_1}$
ADDRSTMT	$\ell : p = \text{alloc}_{o_i}$	$\sigma_{\underline{\ell}}(p) := \langle \top, \{o_i\} \rangle$
GEPSTMT	$\ell : p = \&(q \rightarrow i) \text{ or } p = \&q[i]$	$\sigma_{\underline{\ell}}(p) := \bigsqcup_{o \in \gamma(\sigma_{\underline{\ell}}(q))} \bigsqcup_{j \in \gamma(\sigma_{\underline{\ell}}(i))} \langle \top, \{o.\text{fld}_j\} \rangle$
LOADSTMT	$\ell : p = *q$	$\sigma_{\underline{\ell}}(p) := \bigsqcup_{o \in \{o \mid o \in \sigma_{\underline{\ell}}(q)\}} \delta_{\underline{\ell}}(o)$
STORESTMT	$\ell : *p = q$	$\delta_{\underline{\ell}} := (\{o \mapsto \sigma_{\underline{\ell}}(q) \mid o \in \gamma(\sigma_{\underline{\ell}}(p))\} \sqcup \delta_{\underline{\ell}})$

# Abstract Execution on CONSTMT

SVFSTMT	C-Like form	Abstract Execution Rule
CONSTMT	$\ell : p = c$	$\sigma_{\ell}(p) := \langle [c, c], \perp \rangle$



## Algorithm 10: Abstract Execution Rule for CONSTMT

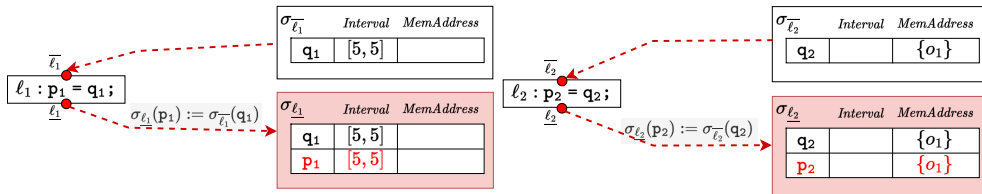
```

1 Function updateStateOnAddr(addr):
2   node = addr → getICFGNode();
3   as = getAbsStateFromTrace(node);
4   initObjVar(as, SVFUtil :: cast(ObjVar)(addr → getRHSVar()));
5   as[addr → getLHSVarID()] = as[addr → getRHSVarID()];

```

# Abstract Execution on COPYSTMT

SVFSTMT	C-Like form	Abstract Execution Rule
COPYSTMT	$\ell : p = q$	$\sigma_{\underline{\ell}}(p) := \sigma_{\bar{\ell}}(q)$



## Algorithm 11: Abstract Execution Rule for COPYSTMT

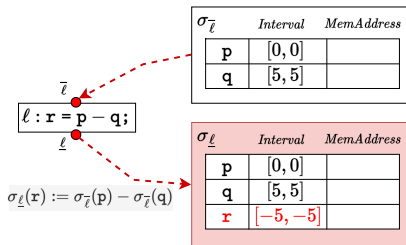
```

1 Function updateStateOnCopy(copy):
2   node = copy → getICFGNode();
3   as = getAbsStateFromTrace(node);
4   lhs = copy → getLHSVarID();
5   rhs = copy → getRHSVarID();
6   as[lhs] = as[rhs];

```

# Abstract Execution on BINARYSTMT

SVFSTMT	C-Like form	Abstract Execution Rule
BINARYSTMT	$\ell : \mathbf{r} = \mathbf{p} \otimes \mathbf{q}$	$\sigma_{\underline{\ell}}(\mathbf{r}) := \sigma_{\bar{\ell}}(\mathbf{p}) \hat{\otimes} \sigma_{\bar{\ell}}(\mathbf{q})$



## Algorithm 12: Abstract Execution Rule for BINARYSTMT

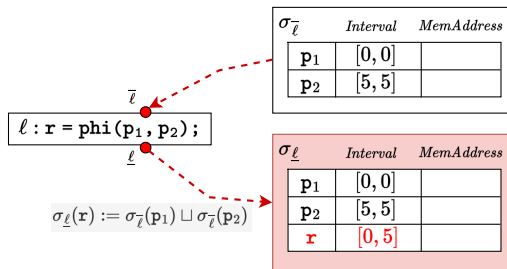
```

1 Function updateStateOnBinary(binary):
2   node = binary → getICFGNode();
3   as = getAbsStateFromTrace(node);
4   op0 = binary → getOpVarID(0);
5   op1 = binary → getOpVarID(1);
6   res = binary → getResID();
7   as[res] = as[op0]  $\hat{\otimes}$  as[op1]
  
```



# Abstract Execution on PHISTMT

SVFSTMT	C-Like form	Abstract Execution Rule
PHISTMT	$\ell : r = \text{phi}(p_1, p_2, \dots, p_n)$	$\sigma_{\underline{\ell}}(r) := \bigsqcup_{i=1}^n \sigma_{\bar{\ell}}(p_i)$



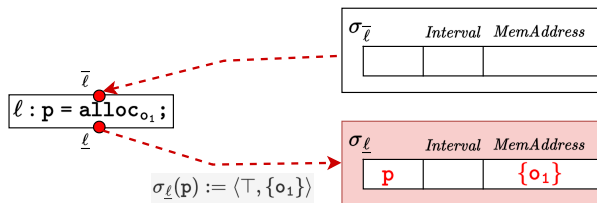
## Algorithm 13: Abstract Execution Rule for PHISTMT

```

1 Function updateStateOnPhi(phi):
2   node = phi → getICFGNode();
3   as = getAbsStateFromTrace(node);
4   res = phi → getResID();
5   rhs = AbstractValue();
6   for i = 0; i < phi → getOpVarNum(); i ++ do
7     curId = phi → getOpVarID(i);
8     rhs.join_with(as[curId])
9   as[res] = rhs
    
```

# Abstract Execution on ADDRSTMT

SVFSTMT	C-Like form	Abstract Execution Rule
ADDRSTMT	$\ell : p = \text{alloc}_{o_1}$	$\sigma_{\underline{\ell}}(p) := \langle \top, \{o_1\} \rangle$



## Algorithm 14: Abstract Execution Rule for ADDRSTMT

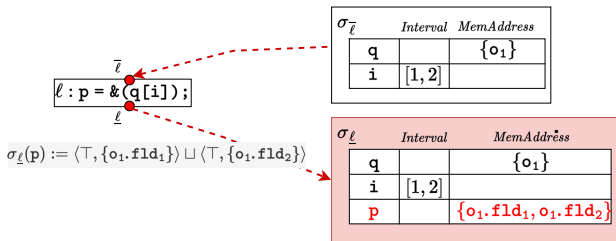
```

1 Function updateStateOnAddr(addr):
2   node = addr → getICFGNode();
3   as = getAbsStateFromTrace(node);
4   initObjVar(as, SVFUtil :: cast<ObjVar>(addr → getRHSVar()));
5   as[addr → getLHSVarID()] = as[addr → getRHSVarID()];

```

# Abstract Execution on GEPSTMT

SVFSTMT	C-Like form	Abstract Execution Rule
GEPSTMT	$\ell : p = \&(q \rightarrow i) \text{ or } p = \&q[i]$	$\sigma_{\underline{\ell}}(p) := \bigsqcup_{o \in \gamma(\sigma_{\bar{\ell}}(q))} \bigsqcup_{j \in \gamma(\sigma_{\bar{\ell}}(i))} \langle \top, \{o.fld_j\} \rangle$



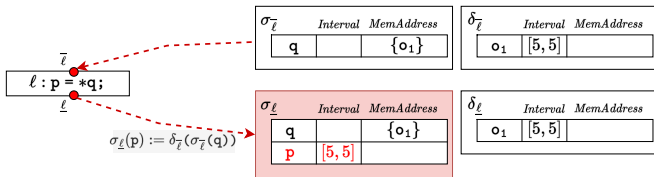
## Algorithm 15: Abstract Execution Rule for GEPSTMT

```

1 Function updateStateOnGep(gep):
2   node = gep → getICFGNode();
3   as = getAbsStateFromTrace(node);
4   rhs = gep → getRHSVarID();
5   lhs = gep → getLHSVarID();
6   as[lhs] = as.getGepObjAddrs(rhs, as.getElementIndex(gep));
  
```

# Abstract Execution on LOADSTMT

SVFSTMT	C-Like form	Abstract Execution Rule
LOADSTMT	$\ell : p = *q$	$\sigma_{\underline{\ell}}(p) := \bigsqcup_{o \in \{o \mid o \in \sigma_{\bar{\ell}}(q)\}} \delta_{\bar{\ell}}(o)$



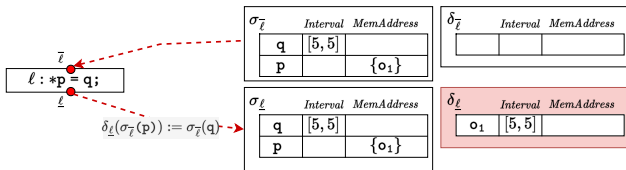
## Algorithm 16: Abstract Execution Rule for LOADSTMT

```

1 Function updateStateOnLoad(load):
2   node = load → getICFGNode();
3   as = getAbsStateFromTrace(node);
4   rhs = load → getRHSVarID();
5   lhs = load → getLHSVarID();
6   as[lhs] = as.loadValue(rhs)
  
```

## Abstract Execution on STORETMT

SVF <sub>STMT</sub>	C-Like form	Abstract Execution Rule
STORE <sub>STMT</sub>	$\ell : *p = q$	$\delta_{\underline{\ell}} := (\{o \mapsto \sigma_{\bar{\ell}}(q) \mid o \in \gamma(\sigma_{\bar{\ell}}(p))\} \sqcup \delta_{\underline{\ell}})$



**Algorithm 17:** Abstract Execution Rule for STORESTMT

```

1 Function updateStateOnStore(store):
2   node = store → getICFGNode();
3   as = getAbsStateFromTrace(node);
4   rhs = store → getRHSVarID();
5   lhs = store → getLHSVarID();
6   as.storeValue(lhs, as[rhs])

```

# Overall Algorithm of Abstract Interpretation

---

## Algorithm 1: Analyse from main function

---

```
1 Function analyse() // driver function to start the analysis:
2   initWTO();
3   handleGlobalNode();
4   if getSVFFunction (main) then
5     wto := funcToWTO[main];
6     handleWTOComponents(wto → getWTOComponents());
```

---

## Algorithm 2: Handle WTO components

---

```
1 Function handleWTOComponents (wtoComps):
2   for wtoNode ∈ wtoComps do
3     if node = SVFUtil :: dyn.cast(ICFGSingletonWTO)(wtoNode) then
4       handleSingletonWTO(node)
5     else if cycle = SVFUtil :: dyn.cast(ICFGCycleWTO)(wtoNode) then
6       handleCycleWTO(cycle)
7     else
8       assert(false && "unknownWTOtype!")
```

---

---

## Algorithm 3: Handle Singleton WTO

---

```
1 Function handleSingletonWTO (singletonWTO):
2   node := singletonWTO → node();
3   feasible := mergeStatesFromPredecessors(node, preAbsTrace[node]);
4   if feasible then
5     postAbsTrace[node] := preAbsTrace[node];
6   else
7     return;
8   foreach stmt ∈ node → getSVFStmts() do
9     updateAbsState(stmt);
10    bufOverflowDetection(stmt);
11   if callnode = SVFUtil :: dyn.cast(CallICFGNode)(node) then
12     funName := callnode → getCallSite() → getCallee() → getName()
13     if funName == "OVERFLOW" && funName == "svf_assert" then
14       // Handle svf_assert and OVERFLOW stub function for
15       correctness validation;
16       handleStubFunctions(callnode);
17     else
18       // Does not analyze recursive functions in this course;
19       handleCallSite(callnode);
```

---

# Overall Algorithm of Abstract Interpretation

## Algorithm 4: Handle Cycle WTO

```
1 Function handleCycleWTO (cycle):  
2   feasible := mergeStatesFromPredecessors(cycle.head, preAbsTrace[cycle.head]);  
3   increasing := true;  
4   if !feasible then  
5     return;  
6   else  
7     cur_iter := 0;  
8     while true do  
9       if cur_iter >= Options.WidenDelay() then  
10        prev_head_as := postAbsTrace[cycle.head];  
11        handleSingletonWTO(cycle.head());  
12        cur_head_as := postAbsTrace[cycle.head];  
13        if increasing then  
14          postAbsTrace[cycle.head] := prev_head_as.widening(cur_head_as);  
15          if postAbsTrace[cycle.head] == prev_head_as then  
16            increasing := false;  
17            Continue;  
18          else  
19            postAbsTrace[cycle.head] := prev_head_as.narrowing(cur_head_as);  
20            if postAbsTrace[cycle.head] == prev_head_as then  
21              Break;  
22        else  
23          handleSingletonWTO(cycle.head());  
24        cur_iter ++;
```

# An Example: Abstract Trace $\sigma$ for Top-level Variables

```
extern void assert(int);

int main(){
  int a = 0;
  while(a < 10) {
    a++;
  }
  assert(a == 10);
  return 0;
}
```

Source Code

Compile to LLVM IR

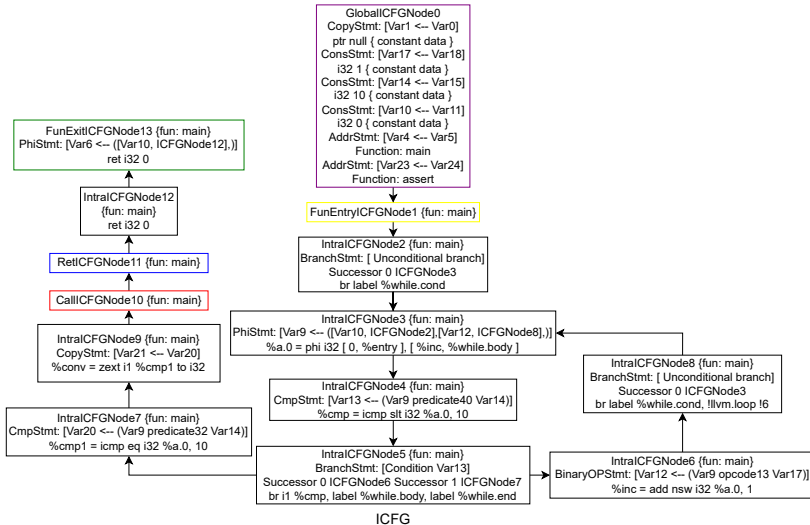


```
define dso_local i32 @main() {
entry:
  br label %while.cond
while.cond:
  %a.0 = phi i32 [ 0, %entry ], [ %inc, %while.body ]
  %cmp = icmp slt i32 %a.0, 10
  br i1 %cmp, label %while.body, label %while.end
while.body:
  %inc = add nsw i32 %a.0, 1
  br label %while.cond,
while.end:
  %cmp1 = icmp eq i32 %a.0, 10
  %conv = zext i1 %cmp1 to i32
  call void @assert(i32 noundef %conv)
  ret i32 0
}
```

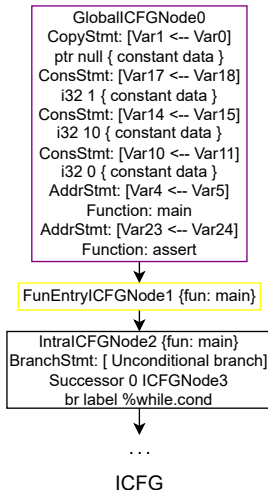
LLVM IR



# An Example: Abstract Trace $\sigma$ for Top-level Variables



# An Example: Abstract Trace $\sigma$ for Top-level Variables



## Algorithm 5: Abstract execution guided by WTO

```
1 Function handleStatement( $\ell$ ):
2    $tmpAS := preAbsTrace[\ell]$ ;
3   if  $\ell$  is CONSSTMT or ADDRSTMT then
4     | updateStateOnAddr( $\ell$ );
5   else if  $\ell$  is COPYSTMT then
6     | updateStateOnCopy( $\ell$ );
7   ...;
```

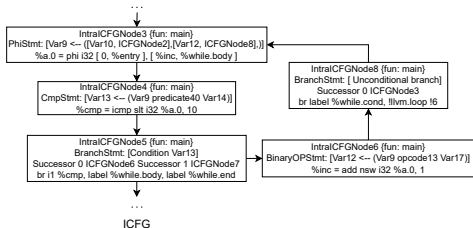
$postAbsTrace[\ell_0].varToAbsVal$  :

SVFVar	AbstractValue $\langle Interval, MemAddress \rangle$
Var0	$\langle \perp, \{0x7f00\} \rangle$
Var1	$\langle \perp, \{0x7f00\} \rangle$
Var18	$\langle [1, 1], \perp \rangle$
Var17	$\langle [1, 1], \perp \rangle$
Var14	$\langle [10, 10], \perp \rangle$
Var15	$\langle [10, 10], \perp \rangle$
Var10	$\langle [0, 0], \perp \rangle$
Var11	$\langle [0, 0], \perp \rangle$

...

Print out the table via `as.printAbstractState()`. The AbstractValue can **either be an interval or addresses**, but not both!

# An Example: Abstract Trace $\sigma$ for Top-level Variables



$postAbsTrace[\ell_3].varToAbsVal :$

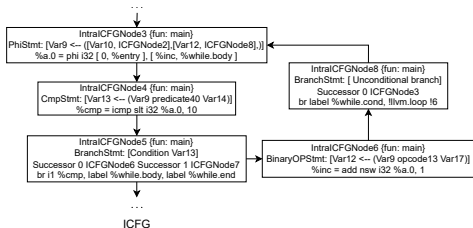
SVFVar	AbstractValue( <i>Interval</i> , <i>MemAddress</i> )
...	
<i>Var10</i>	$\langle [0, 0], \perp \rangle$
<i>Var9</i>	$\langle [0, 0], \perp \rangle$
...	

## Algorithm 12: Handle Cycle WTO

```

1 Function handleCycleWTO(cycle):
2   cycle.head := cycle → head() → node();
3   increasing := true;
4   cur_iter := 0;
5   while true do
6     if cur_iter ≥ Options :: WidenDelay() then
7       prev_head_state := postAbsTrace[cycle.head];
8       handleICFGNode(cycle → head());
9       cur_head_state := postAbsTrace[cycle.head];
10      if increasing then
11        postAbsTrace[cycle.head] := prev_head_state.widen(cur_head_state);
12        if postAbsTrace[cycle.head] == prev_head_state then
13          increasing := false;
14          continue;
15      else
16        postAbsTrace[cycle.head] := prev_head_state.narrow(cur_head_state);
17        if postAbsTrace[cycle.head] == prev_head_state then
18          break;
19    else
20      handleICFGNode(cycle → head());
21    handleWTOComponents(cycle → getWTOComponents());
22    cur_iter++ // cur_iter ≡ 1, Options :: WidenDelay() ≡ 3;
  
```

# An Example: Abstract Trace $\sigma$ for Top-level Variables



$postAbsTrace[l_8].varToAbsVal :$

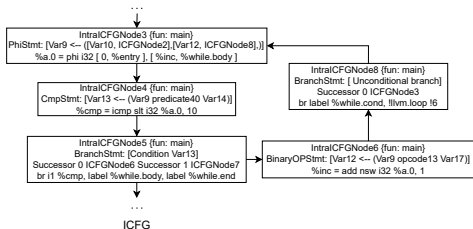
SVFVar	AbstractValue(Interval, MemAddress)
...	
Var10	$\langle [0, 0], \perp \rangle$
Var9	$\langle [0, 0], \perp \rangle$
Var12	$\langle [1, 1], \perp \rangle$
...	

Algorithm 12: Handle Cycle WTO

```

1 Function handleCycleWTO(cycle):
2   cycle_head := cycle → head() → node();
3   increasing := true;
4   cur_iter := 0 // cur_iter ≡ 1, Options :: WidenDelay() ≡ 3;
5   while true do
6     if cur_iter ≥ Options :: WidenDelay() then
7       prev_head_state := postAbsTrace[cycle_head];
8       handleICFGNode(cycle → head());
9       cur_head_state := postAbsTrace[cycle_head];
10      if increasing then
11        postAbsTrace[cycle_head] := prev_head_state.widen(cur_head_state);
12        if postAbsTrace[cycle_head] == prev_head_state then
13          increasing := false;
14          continue;
15      else
16        postAbsTrace[cycle_head] := prev_head_state.narrow(cur_head_state);
17        if postAbsTrace[cycle_head] == prev_head_state then
18          break;
19    else
20      handleICFGNode(cycle → head());
21    handleWTOComponents(cycle → getWTOComponents());
22    cur_iter++ // cur_iter ≡ 2, Options :: WidenDelay() ≡ 3;
  
```

# An Example: Abstract Trace $\sigma$ for Top-level Variables



$postAbsTrace[\ell_3].varToAbsVal :$

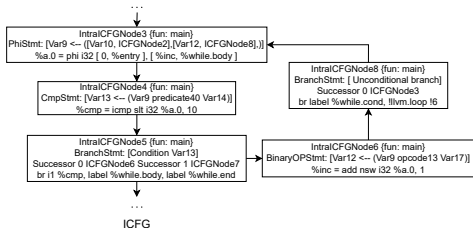
SVFVar	AbstractValue( <i>Interval</i> , <i>MemAddress</i> )
...	
<i>Var9</i>	$\langle [0, 1], \perp \rangle$
<i>Var12</i>	$\langle [1, 1], \perp \rangle$
...	

## Algorithm 12: Handle Cycle WTO

```

1 Function handleCycleWTO(cycle):
2   cycle.head := cycle → head() → node();
3   increasing := true;
4   cur_iter := 0;
5   while true do
6     if cur_iter ≥ Options :: WidenDelay() then
7       prev_head_state := postAbsTrace[cycle.head];
8       handleICFGNode(cycle → head());
9       cur_head_state := postAbsTrace[cycle.head];
10      if increasing then
11        postAbsTrace[cycle.head] := prev_head_state.widen(cur_head_state);
12        if postAbsTrace[cycle.head] == prev_head_state then
13          increasing := false;
14          continue;
15      else
16        postAbsTrace[cycle.head] := prev_head_state.narrow(cur_head_state);
17        if postAbsTrace[cycle.head] == prev_head_state then
18          break;
19    else
20      handleICFGNode(cycle → head());
21    handleWTOComponents(cycle → getWTOComponents());
22    cur_iter++ // cur_iter ≡ 2, Options :: WidenDelay() ≡ 3;
  
```

# An Example: Abstract Trace $\sigma$ for Top-level Variables



$postAbsTrace[\ell_g].varToAbsVal :$

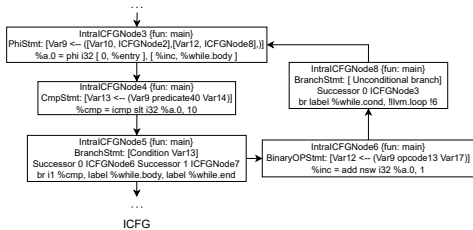
SVFVar	AbstractValue( <i>Interval</i> , <i>MemAddress</i> )
...	
Var9	$\langle [0, 1], \perp \rangle$
Var12	$\langle [1, 2], \perp \rangle$
...	

## Algorithm 12: Handle Cycle WTO

```

1 Function handleCycleWTO(cycle):
2   cycle_head := cycle → head() → node();
3   increasing := true;
4   cur_iter := 0;
5   while true do
6     if cur_iter ≥ Options :: WidenedDelay() then
7       prev_head_state := postAbsTrace[cycle_head];
8       handleICFGNode(cycle → head());
9       cur_head_state := postAbsTrace[cycle_head];
10      if increasing then
11        postAbsTrace[cycle_head] := prev_head_state.widen(cur_head_state);
12        if postAbsTrace[cycle_head] == prev_head_state then
13          increasing := false;
14          continue;
15      else
16        postAbsTrace[cycle_head] := prev_head_state.narrow(cur_head_state);
17        if postAbsTrace[cycle_head] == prev_head_state then
18          break;
19    else
20      handleICFGNode(cycle → head());
21      handleWTOComponents(cycle → getWTOComponents());
22      cur_iter++ // cur_iter ≡ 2, Options :: WidenedDelay() ≡ 3;
  
```

# An Example: Abstract Trace $\sigma$ for Top-level Variables



$postAbsTrace[\ell_3].varToAbsVal :$

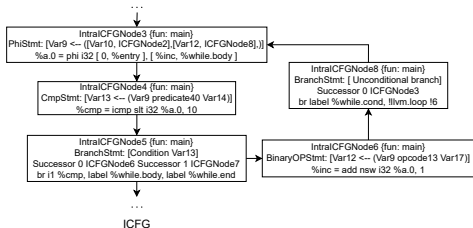
SVFVar	$\langle Interval, MemAddress \rangle$
...	
Var9	$\langle [0, +\infty], \perp \rangle$
Var12	$\langle [1, 2], \perp \rangle$
...	

## Algorithm 12: Handle Cycle WTO

```

1 Function handleCycleWTO (cycle):
2   cycle_head := cycle → head() → node();
3   increasing := true;
4   cur_iter := 0;
5   while true do
6     if cur_iter ≥ Options :: WidenedDelay() then
7       prev_head_state := postAbsTrace[cycle_head];
8       handleICFGNode (cycle → head());
9       cur_head_state := postAbsTrace[cycle_head];
10      if increasing then
11        postAbsTrace[cycle_head] := prev_head_state.widen(cur_head_state);
12        if postAbsTrace[cycle_head] == prev_head_state then
13          increasing := false;
14          continue;
15      else
16        postAbsTrace[cycle_head] := prev_head_state.narrow(cur_head_state);
17        if postAbsTrace[cycle_head] == prev_head_state then
18          break;
19    else
20      handleICFGNode (cycle → head());
21  handleWTOComponents (cycle → getWTOComponents());
22  cur_iter++ // cur_iter ≡ 3, Options :: WidenedDelay ≡ 3;
  
```

# An Example: Abstract Trace $\sigma$ for Top-level Variables



$postAbsTrace[\ell_8].varToAbsVal :$

SVFVar	AbstractValue( <i>Interval</i> , <i>MemAddress</i> )
...	
Var9	$\langle [0, 9], \perp \rangle$
Var12	$\langle [1, 10], \perp \rangle$
...	

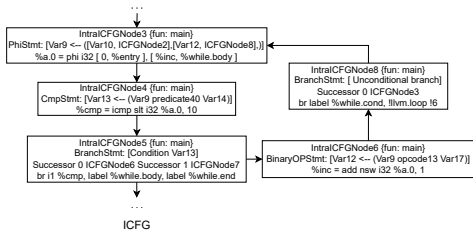
## Algorithm 12: Handle Cycle WTO

```

1 Function handleCycleWTO(cycle):
2   cycle.head := cycle → head() → node();
3   increasing := true;
4   cur_iter := 0;
5   while true do
6     if cur_iter ≥ Options :: WidenDelay() then
7       prev_head_state := postAbsTrace[cycle.head];
8       handleICFGNode(cycle → head());
9       cur_head_state := postAbsTrace[cycle.head];
10      if increasing then
11        postAbsTrace[cycle.head] := prev_head_state.widen(cur_head_state);
12        if postAbsTrace[cycle.head] == prev_head_state then
13          increasing := false;
14          continue;
15      else
16        postAbsTrace[cycle.head] := prev_head_state.narrow(cur_head_state);
17        if postAbsTrace[cycle.head] == prev_head_state then
18          break;
19    else
20      handleICFGNode(cycle → head());
21      handleWTOComponents(cycle → getWTOComponents()) // cur_iter ≡ 3;
22      cur_iter++ // cur_iter ≡ 3, Options :: WidenDelay ≡ 3;
  
```



# An Example: Abstract Trace $\sigma$ for Top-level Variables



$postAbsTrace[\ell_3].varToAbsVal :$

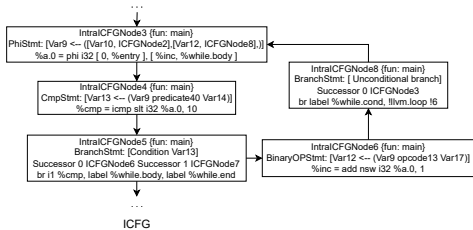
SVFVar	$\langle Interval, MemAddress \rangle$
...	...
<i>Var9</i>	$\langle [0, +\infty], \perp \rangle$
<i>Var12</i>	$\langle [1, 10], \perp \rangle$
...	...

## Algorithm 12: Handle Cycle WTO

```

1 Function handleCycleWTO(cycle):
2   cycle_head := cycle → head() → node();
3   increasing := true;
4   cur_iter := 0;
5   while true do
6     if cur_iter ≥ Options :: WidenDelay() then
7       prev_head_state := postAbsTrace[cycle_head];
8       handleICFGNode(cycle → head());
9       cur_head_state := postAbsTrace[cycle_head];
10      if increasing then
11        postAbsTrace[cycle_head] := prev_head_state.widen(cur_head_state);
12        if postAbsTrace[cycle_head] == prev_head_state then
13          increasing := false;
14          continue;
15      else
16        postAbsTrace[cycle_head] := prev_head_state.narrow(cur_head_state);
17        if postAbsTrace[cycle_head] == prev_head_state then
18          break;
19    else
20      handleICFGNode(cycle → head());
21    handleWTOComponents(cycle → getWTOComponents());
22    cur_iter++ // cur_iter ≡ 4, Options :: WidenDelay ≡ 3;
  
```

# An Example: Abstract Trace $\sigma$ for Top-level Variables



$postAbsTrace[\ell_3].varToAbsVal :$

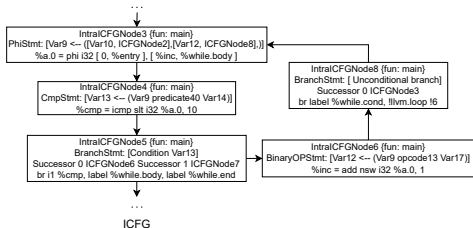
SVFVar	$\langle Interval, MemAddress \rangle$
...	
<i>Var9</i>	$\langle [0, 10], \perp \rangle$
<i>Var12</i>	$\langle [1, 10], \perp \rangle$
...	

## Algorithm 12: Handle Cycle WTO

```

1 Function handleCycleWTO(cycle):
2   cycle_head := cycle → head() → node();
3   increasing := true;
4   cur_iter := 0;
5   while true do
6     if cur_iter ≥ Options :: WidenDelay() then
7       prev_head_state := postAbsTrace[cycle_head];
8       handleICFGNode(cycle → head()) // increasing ≡ false;
9       cur_head_state := postAbsTrace[cycle_head];
10      if increasing then
11        postAbsTrace[cycle_head] := prev_head_state.widen(cur_head_state);
12        if postAbsTrace[cycle_head] == prev_head_state then
13          increasing := false;
14          continue;
15      else
16        postAbsTrace[cycle_head] := prev_head_state.narrow(cur_head_state);
17        if postAbsTrace[cycle_head] == prev_head_state then
18          break;
19    else
20      handleICFGNode(cycle → head());
21  handleWTOComponents(cycle → getWTOComponents());
22  cur_iter++ // cur_iter ≡ 5, Options :: WidenDelay ≡ 3;
  
```

# An Example: Abstract Trace $\sigma$ for Top-level Variables



$postAbsTrace[\ell_8].varToAbsVal :$

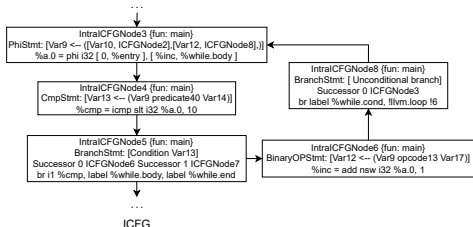
SVFVar	$\langle Interval, MemAddress \rangle$
...	...
Var9	$\langle [0, 9], \perp \rangle$
Var12	$\langle [1, 10], \perp \rangle$
...	...

## Algorithm 12: Handle Cycle WTO

```

1 Function handleCycleWTO(cycle):
2   cycle.head := cycle → head() → node();
3   increasing := true;
4   cur_iter := 0;
5   while true do
6     if cur_iter ≥ Options :: WidenedDelay() then
7       prev_head_state := postAbsTrace[cycle.head];
8       handleICFGNode(cycle → head());
9       cur_head_state := postAbsTrace[cycle.head];
10      if increasing then
11        postAbsTrace[cycle.head] := prev_head_state.widen(cur_head_state);
12        if postAbsTrace[cycle.head] == prev_head_state then
13          increasing := false;
14          continue;
15      else
16        postAbsTrace[cycle.head] := prev_head_state.narrow(cur_head_state);
17        if postAbsTrace[cycle.head] == prev_head_state then
18          break;
19    else
20      handleICFGNode(cycle → head());
21      handleWTOComponents(cycle → getWTOComponents()) // cur_iter ≡ 5;
22      cur_iter++ // cur_iter ≡ 5, Options :: WidenedDelay ≡ 3;
  
```

# An Example: Abstract Trace $\sigma$ for Top-level Variables



$postAbsTrace[\ell_3].varToAbsVal :$

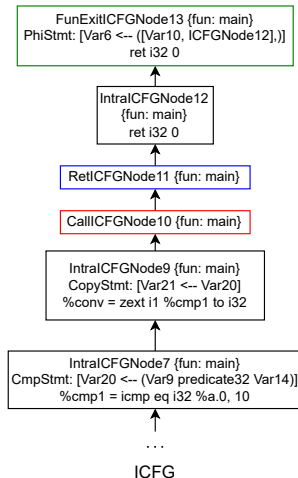
SVFVar	$\langle Interval, MemAddress \rangle$
...	
<i>Var9</i>	$\langle [0, 10], \perp \rangle$
<i>Var12</i>	$\langle [1, 10], \perp \rangle$
...	

## Algorithm 12: Handle Cycle WTO

```

1 Function handleCycleWTO(cycle):
2   cycle_head := cycle → head() → node();
3   increasing := true;
4   cur_iter := 0;
5   while true do
6     if cur_iter ≥ Options :: WidenDelay() then
7       prev_head_state := postAbsTrace[cycle_head];
8       handleICFGNode(cycle → head()) // increasing ≡ false;
9       cur_head_state := postAbsTrace[cycle_head];
10      if increasing then
11        postAbsTrace[cycle_head] := prev_head_state.widen(cur_head_state);
12        if postAbsTrace[cycle_head] == prev_head_state then
13          increasing := false;
14          continue;
15      else
16        postAbsTrace[cycle_head] := prev_head_state.narrow(cur_head_state);
17        if postAbsTrace[cycle_head] == prev_head_state then
18          break;
19    else
20      handleICFGNode(cycle → head());
21    handleWTOComponents(cycle → getWTOComponents());
22    cur_iter++ // cur_iter ≡ 6, Options :: WidenDelay ≡ 3;
  
```

# An Example: Abstract Trace $\sigma$ for Top-level Variables



**Algorithm 13:** Abstract execution guided by WTO

```

1 Function handleStatement( $\ell$ ):
2    $tmpAS := preAbsTrace[\ell]$ ;
3   if  $\ell$  is BINARYSTMT then
4      $updateStateOnBinary(\ell)$ ;
5   ...;
  
```

$postAbsTrace[\ell_7].varToAbsVal$  :

SVFVar	$\langle Interval, MemAddress \rangle$
...	
Var9	$\langle [10, 10], \perp \rangle$
Var20	$\langle [1, 1], \perp \rangle$
...	