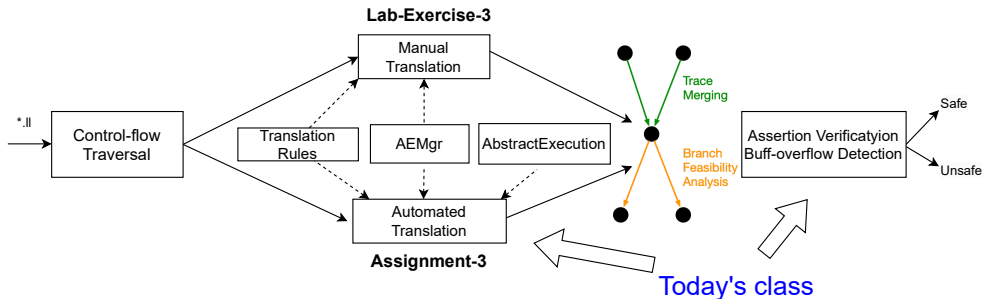# Abstract Interpretation for Code Analysis and Verification

(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:
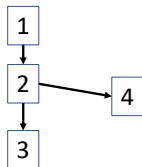
1 2 3 4 ✔

1 2 4 3 ✔

1 3 2 4 ✘

acyclic graph G   Valid/invalid topological order

How About Analyzing Loops?

- **Topological Order** can only be used for directed acyclic graphs (DAGs).
- **Weak Topological Order (WTO)** is a relaxation of the more stringent topological order for graphs with loops.
  - **Cycles Permitted**: allows for cycles within the graph.
  - **Hierarchical Decomposition**: A graph is decomposed into a hierarchical structure where each node or a strongly connected component (SCC) can contain subnodes.
  - **Weak Topological Order or Partial Order**: In a WTO, nodes and SCCs are arranged in a partial order (not enumerating possible infinite loop paths). This order respects the dependencies in a way that allows for iterative analysis.

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

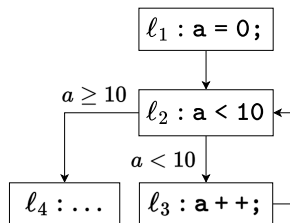# Weak Topological Order

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?**

$\ell_1 : \texttt{a = 0;}$

$a \geq 10$ $\ell_2 : \texttt{a < 10}$

$a < 10$

$\ell_4 : \ldots$ $\ell_3 : \texttt{a + +;}$

Control Flow Graph

1st WTO component: a sigle node

ℓ_1

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?

1st WTO component: a sigle node

ℓ_1 : a = 0;

$a \geq 10$

ℓ_2 : a < 10

$a < 10$

ℓ_4 : ...

ℓ_3 : a + +;

Control Flow Graph

ℓ_1 ℓ_2 ℓ_3
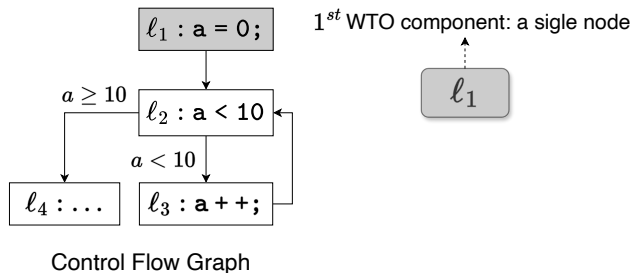
2nd WTO component: a cycle

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.
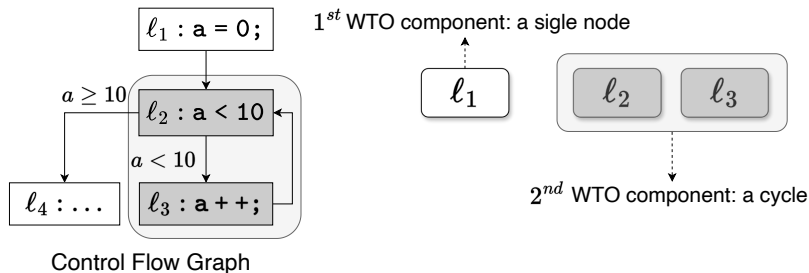

What is the weak topological order?

$\ell_1 : \texttt{a = 0;}$

$a \geq 10$ $\ell_2 : \texttt{a < 10}$

$a < 10$

$\ell_4 : \ldots$ $\ell_3 : \texttt{a + +;}$

Control Flow Graph

1st WTO component: a sigle node

ℓ_1 ℓ_2 ℓ_3

WTO cycle head

2nd WTO component: a cycle

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.
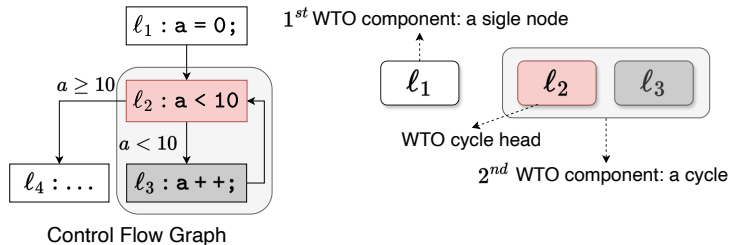

What is the weak topological order?

1st WTO component: a sigle node

3rd WTO component: a sigle node

WTO cycle head

2nd WTO component: a cycle

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.
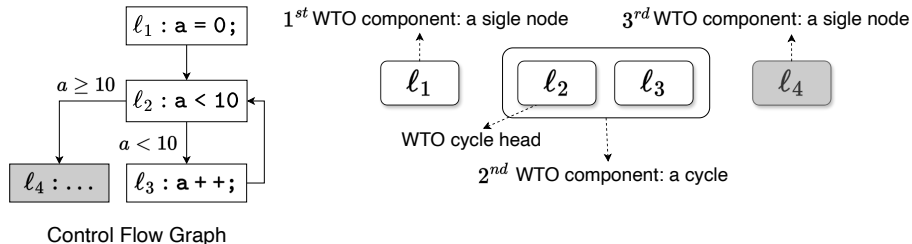

What is the weak topological order?

1st WTO component: a sigle node 3rd WTO component: a sigle node

ℓ_1 ℓ_2 ℓ_3 ℓ_4

WTO cycle head

2nd WTO component: a cycle

Control Flow Graph

Analyze each node following the WTO

Analyze ℓ_1 ⟹ Repeat: analyze ℓ_2 and ℓ_3 ⟹ Analyze ℓ_4

$\ell_1 : \texttt{a = 0;}$

$a \geq 10$ $\ell_2 : \texttt{a < 10}$

$a < 10$

$\ell_4 : \texttt{...}$ $\ell_3 : \texttt{a + +;}$

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.
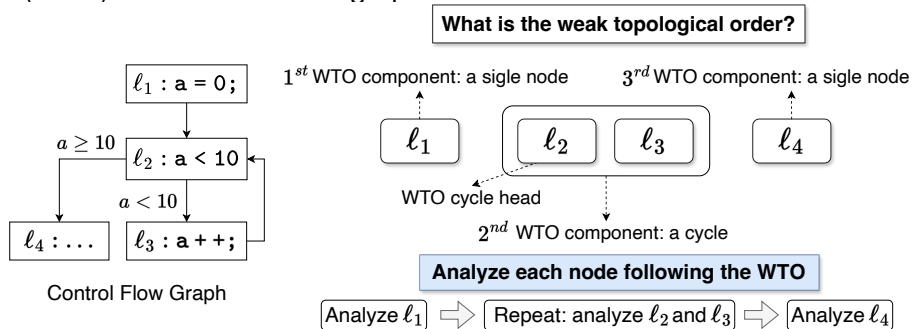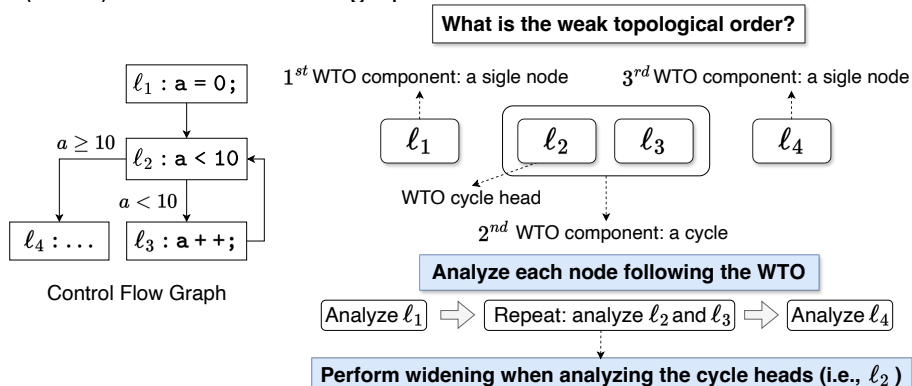

What is the weak topological order?

1st WTO component: a sigle node 3rd WTO component: a sigle node

ℓ_1 ℓ_2 ℓ_3 ℓ_4

WTO cycle head

2nd WTO component: a cycle

Analyze each node following the WTO

Analyze ℓ_1 ⇒ Repeat: analyze ℓ_2 and ℓ_3 ⇒ Analyze ℓ_4

Perform widening when analyzing the cycle heads (i.e., ℓ_2)

Control Flow Graph:
$\ell_1 : a = 0;$
$a \geq 10$
$\ell_2 : a < 10$
$a < 10$
$\ell_4 : \ldots$ $\ell_3 : a + +;$

WTO, Widening and Narrowing

Why Weak Topological Order?

- Handling cyclic dependencies
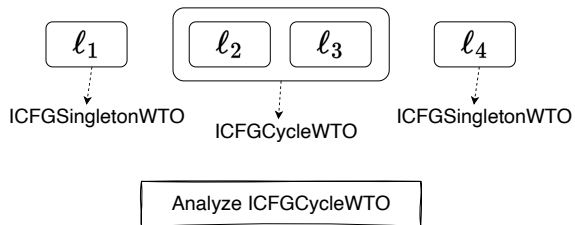- Efficient fixed-point computation

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].
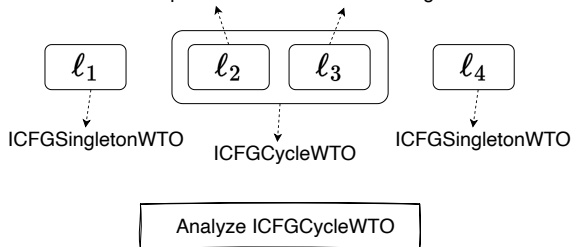 - User-specification (assertions and guard conditions)

Widening and Narrowing

Algorithm 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++;
```
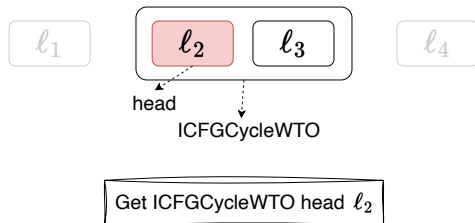
Widening and Narrowing

Sub WTO Components: each is an ICFGSingletonWTO

ICFGSingletonWTO

ICFGCycleWTO

ICFGSingletonWTO

Analyze ICFGCycleWTO

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++;
```

Weak Topological Order

Get ICFGCycleWTO head ℓ_2

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++;
```
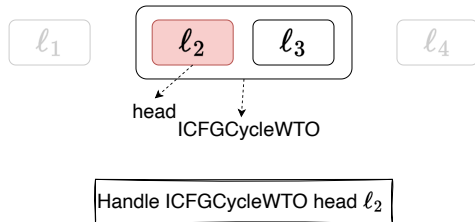
Weak Topological Order

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++;
```
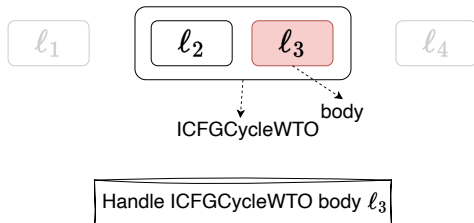
Widening and Narrowing

ℓ_1 ℓ_2 ℓ_3 ℓ_4

ICFGCycleWTO

body

Handle ICFGCycleWTO body ℓ_3

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 $\geq$ 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++;

Note: getIWTOcomponents returns Cycle WTO body, i.e., ℓ_3

Widening and Narrowing

ℓ_1 ℓ_2 ℓ_3 ℓ_4

head

ICFGCycleWTO

When $cur\_iter \geq Options :: WidenDelay()$

perform widening on ℓ_2

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++;
```
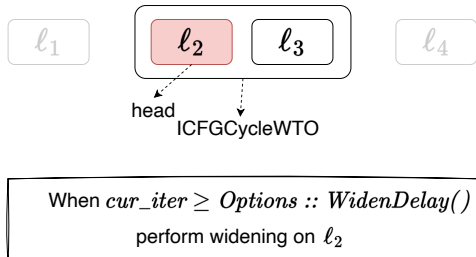
Widening and Narrowing

Widening reaches a fixpoint

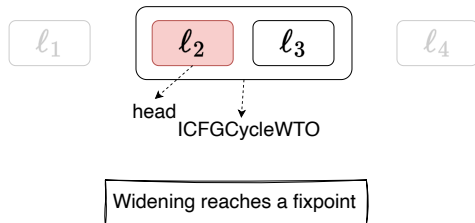**Algorithm 12:** Handle Cycle WTO

```
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++;
```
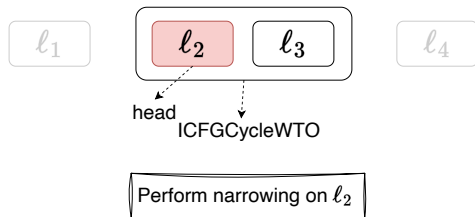
Widening and Narrowing

ℓ_1 ℓ_2 ℓ_3 ℓ_4

head

ICFGCycleWTO

Perform narrowing on ℓ_2

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++;

Widening and Narrowing

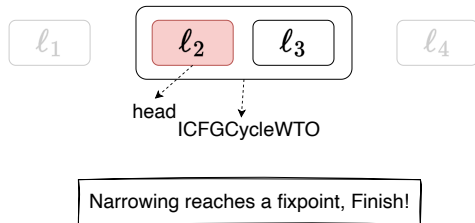ICFGCycleWTO

Narrowing reaches a fixpoint, Finish!

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++;

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
CONSSTMT	$\ell : p = c$
COPYSTMT	$\ell : p = q$
BINARYSTMT	$\ell : r = p \otimes q$
PHISTMT	$\ell : r = \text{phi}(p_1, p_2, \ldots, p_n)$
SEQUENCE	$\ell_1; \ell_2$
BRANCHSTMT	$\ell_1 : \text{if}(x < c)$ then ℓ_2 else ℓ_3

Abstract Interpretation on Pointer-Free SVFIR
Interval Domain

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

SVFSTMT	C-Like form	Abstract Execution Rule
CONSSTMT	$\ell : \mathtt{p} = \mathtt{c}$	$\sigma_{\underline{\ell}}(\mathtt{p}) := [\mathtt{c}, \mathtt{c}]$
COPYSTMT	$\ell : \mathtt{p} = \mathtt{q}$	$\sigma_{\underline{\ell}}(\mathtt{p}) := \sigma_{\overline{\ell}}(\mathtt{q})$
BINARYSTMT	$\ell : \mathtt{r} = \mathtt{p} \otimes \mathtt{q}$	$\sigma_{\underline{\ell}}(r) := \sigma_{\overline{\ell}}(p) \hat{\otimes} \sigma_{\overline{\ell}}(q)$
PHISTMT	$\ell : \mathtt{r} = \mathtt{phi}(\mathtt{p_1}, \mathtt{p_2}, \ldots, \mathtt{p_n})$	$\sigma_{\underline{\ell}}(r) := \bigsqcup_{i=1}^{n} \sigma_{\overline{\ell}}(p_i)$
SEQUENCE	$\ell_1 ; \ell_2$	$\forall v \in \mathbb{V}, \sigma_{\overline{\ell_2}}(v) \sqsupseteq \sigma_{\underline{\ell_1}}(v)$
BRANCHSTMT	$\ell_1 :$ if$(x < c)$ then ℓ_2 else ℓ_3	$\sigma_{\overline{\ell_2}}(x) := \sigma_{\underline{\ell_1}}(x) \sqcap [-\infty, c-1]$, if $\sigma_{\underline{\ell_1}}(x) \sqcap [-\infty, c-1] \neq \bot$ $\sigma_{\overline{\ell_3}}(x) := \sigma_{\underline{\ell_1}}(x) \sqcap [c, +\infty]$, if $\sigma_{\underline{\ell_1}}(x) \sqcap [c, +\infty] \neq \bot$

Abstract Interpretation on BINARYSTMT

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

$\sigma_{\overline{\ell}}(p) := [0,0]$
$\sigma_{\overline{\ell}}(q) := [5,5]$

Abstract state	$\sigma_{\overline{\ell}}$
p	$[0,0]$
q	$[5,5]$

$\overline{\ell}$

$\ell : r = p - q;$

$\underline{\ell}$

$\sigma_{\underline{\ell}}(r) := \sigma_{\overline{\ell}}(p) - \sigma_{\overline{\ell}}(q)$

Abstract state	$\sigma_{\underline{\ell}}$
p	$[0,0]$
q	$[5,5]$
r	$[-5,-5]$
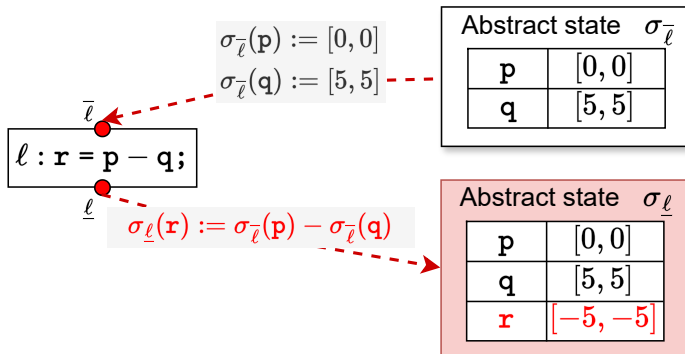
Abstract Interpretation in the Presence of Pointers

- 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
CONSSTMT	$\ell : \mathtt{p} = \mathtt{c}$
COPYSTMT	$\ell : \mathtt{p} = \mathtt{q}$
BINARYSTMT	$\ell : \mathtt{r} = \mathtt{p} \otimes \mathtt{q}$
PHISTMT	$\ell : \mathtt{r} = \mathrm{phi}(\mathtt{p_1}, \mathtt{p_2}, \ldots, \mathtt{p_n})$
SEQUENCE	$\ell_1 ; \ell_2$
BRANCHSTMT	$\ell_1 : \mathrm{if}(x < c)$ then ℓ_2 else ℓ_3
ADDRSTMT	$\ell : \mathtt{p} = \mathtt{alloc}$
GEPSTMT	$\ell : \mathtt{p} = \&(\mathtt{q} \rightarrow \mathtt{i})$ or $\mathtt{p} = \&\mathtt{q[i]}$
LOADSTMT	$\ell : \mathtt{p} = *\mathtt{q}$
STORESTMT	$\ell : *\mathtt{p} = \mathtt{q}$

Combined Analysis
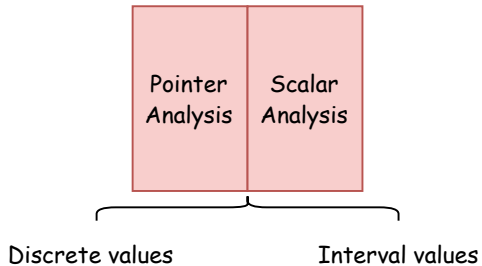
Combined Analysis Using Discrete Values

Pointer Analysis | Scalar Analysis

Discrete values for both analyses?

{1, 2, 3, ..., infinite numbers}

set of all possible values with inputs

Unscalable!

Discrete Values

o_1 o_2 o_3 ... o_1 o_2 o_3 ...

o_1 o_1 o_2 o_1 o_3 ...
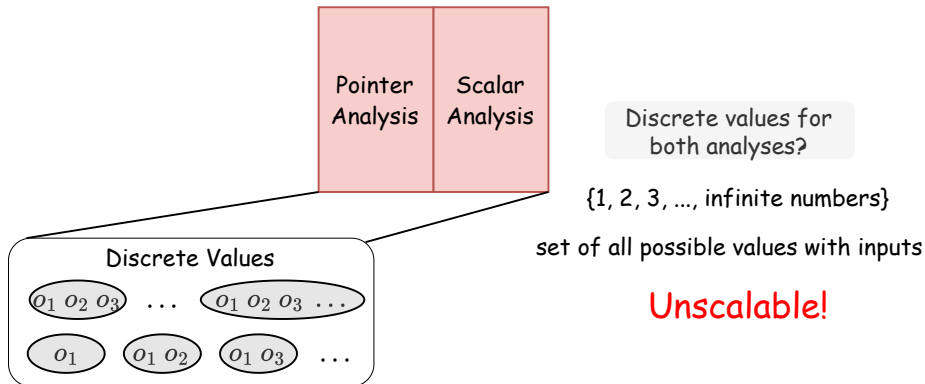
Combined Analysis Using Interval Values

Interval values for both analyses?

Pointer Analysis   Scalar Analysis

$o_1\ o_5$ $\xrightarrow{\text{abstraction}}$ $[o_1, o_5]$

$o_1\ o_2\ o_3\ o_4\ o_5$

discrete value becomes imprecise interval

Imprecise!

Interval Values

$\dots$ $[0, +\infty]$ $\dots$

$\dots$ $[0, 3]$ $[1, 5]$ $\dots$
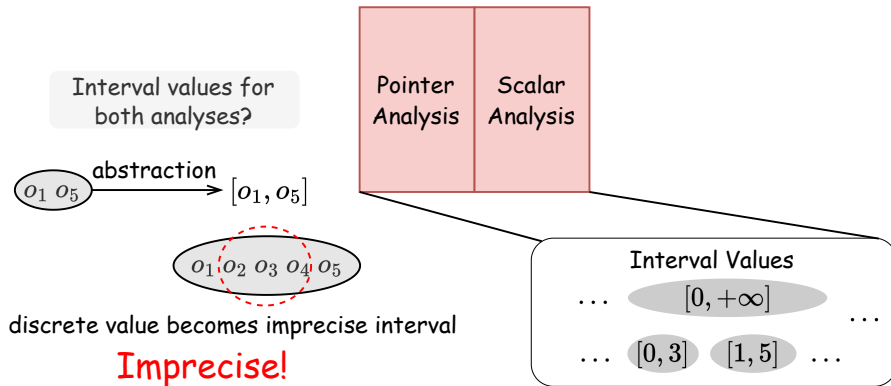
Abstract Interpretation Over a Combined Domain
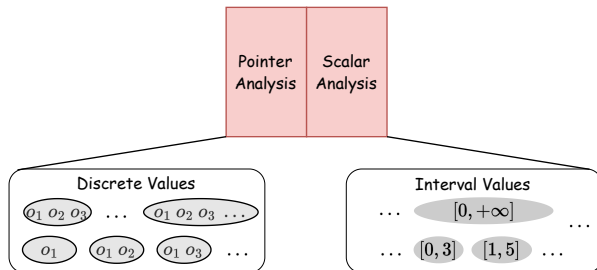

```
p = malloc(m*sizeof(int)); // p points to an array of size m
q = malloc(n*sizeof(int)); // q points to an array of size n
```

```
m = r[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 blue and red denote the discrete values and interval values dependent),

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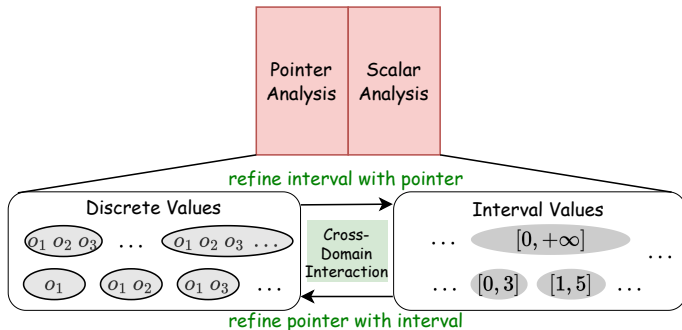
We require **a combination of interval and memory address domains** to precisely and efficiently perform abstract execution on SVFIR in the presence of pointers.

Abstract Interpretation over Interval and MemAddress Domains

A Combined Domain of Intervals and Discrete Memory Addresses
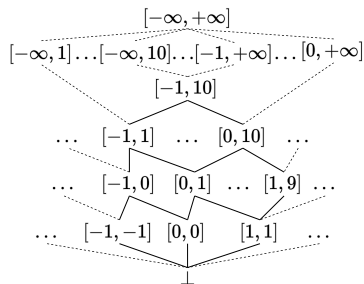
Interval domain for scalar variables
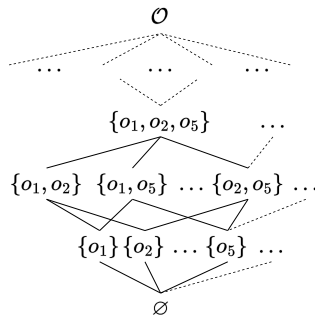
MemAddress domain for discrete memory address values

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	$o \in (\mathbb{S} \cup \mathbb{G} \cup \mathbb{H})$	A single (base) memory object
GepObjVar	$o_i \in (\mathbb{S} \cup \mathbb{G} \cup \mathbb{H}) \times \mathbb{P}$	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 $\sigma \in \mathbb{L} \times \mathbb{P} \to \textit{Interval} \times \textit{MemAddress}$ to track the memory addresses or interval values of these variables.
- For memory objects $\mathbb{O}$, we use $\delta \in \mathbb{L} \times \mathbb{O} \to \textit{Interval} \times \textit{MemAddress}$ to track their abstract values

	Notation	Domain	Data Structure Implementation
Abstract trace	σ	$\mathbb{L} \times \mathbb{P} \to \textit{Interval} \times \textit{MemAddress}$	*preAbsTrace*, *postAbsTrace*
	δ	$\mathbb{L} \times \mathbb{O} \to \textit{Interval} \times \textit{MemAddress}$	
Abstract state	σ_L	$\mathbb{P} \to \textit{Interval} \times \textit{MemAddress}$	*AbstractState.varToAbsVal*
	δ_L	$\mathbb{O} \to \textit{Interval} \times \textit{MemAddress}$	*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** $\mathbb{O}$.

28

Implementation of Abstract Trace and State in Assignment-3

- For a program point *L*, the abstract state is an instance of class *AEState*, consisting of:
  - Top-level variable, *varToAbsVal* : $\sigma_L \in \mathbb{P} \to$ *Interval* $\times$ *MemAddress*
  - Memory object, *addrToAbsVal* : $\delta_L \in$ *MemAddress* $\to$ *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 ℓ, *preAbsTrace*(ℓ) retrieves the abstract state $\langle \sigma_{\overline{\ell}}, \delta_{\overline{\ell}} \rangle$, and *postAbsTrace*($\ell$) represents $\langle \sigma_{\underline{\ell}}, \delta_{\underline{\ell}} \rangle$.
  - For each abstract state $\langle \sigma_{\overline{\ell}}, \delta_{\overline{\ell}} \rangle$ we use as[varId] to operate $\sigma_{\underline{\ell}}$ and use storeValue and loadValue to operate $\delta_{\underline{\ell}}$.
  - 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
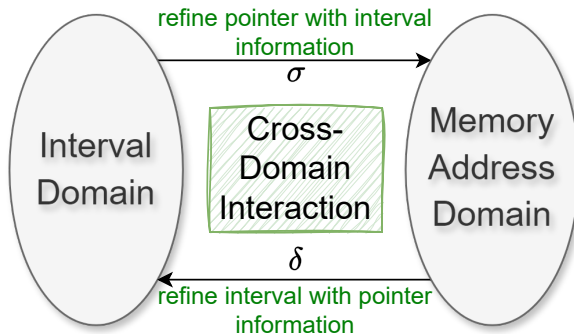
Abstract Execution Rules on SVFIR in the Presence of Pointers

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

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

Abstract Execution on CONSSTMT

SVFSTMT	C-Like form	Abstract Execution Rule
CONSSTMT	$\ell : \text{p} = \text{c}$	$\sigma_{\underline{\ell}}(\text{p}) := \langle [\text{c}, \text{c}], \bot \rangle$
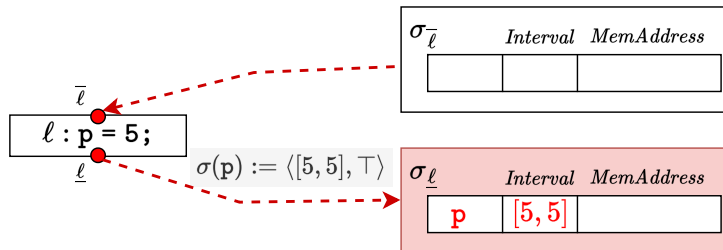
Algorithm 13: Abstract Execution Rule for CONSSTMT

```
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	$\ell : \mathrm{p} = \mathrm{q}$	$\sigma_{\underline{\ell}}(\mathrm{p}) := \sigma_{\overline{\ell}}(\mathrm{q})$
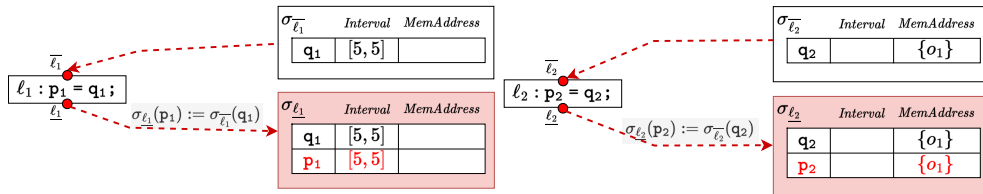
Algorithm 14: Abstract Execution Rule for COPYSTMT

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	$\ell : \mathtt{r} = \mathtt{p} \otimes \mathtt{q}$	$\sigma_{\underline{\ell}}(r) := \sigma_{\overline{\ell}}(p) \hat{\otimes} \sigma_{\overline{\ell}}(q)$

$\sigma_{\overline{\ell}}$

	Interval	MemAddress
p	$[0,0]$	
q	$[5,5]$	

$\ell : \mathtt{r} = \mathtt{p} - \mathtt{q};$

$\sigma_{\underline{\ell}}$

	Interval	MemAddress
p	$[0,0]$	
q	$[5,5]$	
r	$[-5,-5]$	

$\sigma_{\underline{\ell}}(\mathtt{r}) := \sigma_{\overline{\ell}}(\mathtt{p}) - \sigma_{\overline{\ell}}(\mathtt{q})$
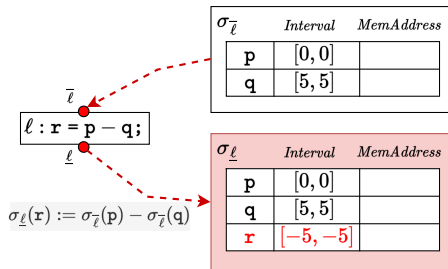
Algorithm 15: Abstract Execution Rule for BINARYSTMT

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

Operands `op0` and `op1` are assumed to be properly initialized (no uninitialized variables or randomization).

Abstract Execution on PHISTMT

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

$\sigma_{\overline{\ell}}$

	Interval	*MemAddress*
$\mathtt{p_1}$	$[0,0]$	
$\mathtt{p_2}$	$[5,5]$	

$\overline{\ell}$

$\ell : \mathtt{r = phi(p_1, p_2)};$

$\underline{\ell}$

$\sigma_{\underline{\ell}}$

	Interval	*MemAddress*
$\mathtt{p_1}$	$[0,0]$	
$\mathtt{p_2}$	$[5,5]$	
$\mathtt{r}$	$[0,5]$	

$\sigma_{\underline{\ell}}(\mathtt{r}) := \sigma_{\overline{\ell}}(\mathtt{p_1}) \sqcup \sigma_{\overline{\ell}}(\mathtt{p_2})$
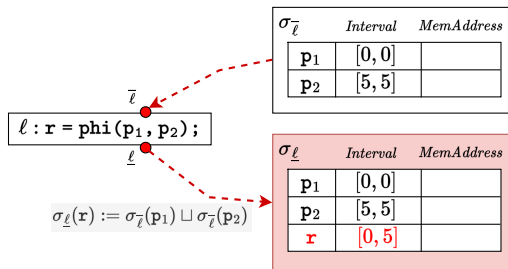
Algorithm 16: Abstract Execution Rule for PHISTMT

1. **Function** *updateStateOnPhi(phi)***:**
2. $\mathtt{node = phi \rightarrow getICFGNode()}$;
3. $\mathtt{as = getAbsStateFromTrace(node)}$;
4. $\mathtt{res = phi \rightarrow getResID()}$;
5. $\mathtt{rhs = AbstractValue()}$;
6.   **for** $\mathtt{i = 0; i < phi \rightarrow getOpVarNum(); i + +}$ **do**
7. $\mathtt{curId = phi \rightarrow getOpVarID(i)}$;
8. $\mathtt{rhs.join\_with(as[curId])}$
9. $\mathtt{as[res] = rhs}$

Abstract Execution on ADDRSTMT

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

$$\sigma_{\underline{\ell}}(\mathtt{p}) := \langle \top, \{o_1\} \rangle$$

Algorithm 17: Abstract Execution Rule for ADDRSTMT

1 Function *updateStateOnAddr(addr)*:
2   node $= $ addr $\rightarrow$ getICFGNode();
3   as $= $ getAbsStateFromTrace(node);
4   initObjVar(as, SVFUtil :: cast$\langle$ObjVar$\rangle$(addr$\rightarrow$getRHSVar()));
5   as[addr$\rightarrow$getLHSVarID()] $= $ as[addr$\rightarrow$getRHSVarID()];

Abstract Execution on GEPSTMT

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

$\sigma_{\overline{\ell}}$

	Interval	MemAddress
q		$\{o_1\}$
i	$[1,2]$	

$\ell : p = \&(q[i]);$

$\sigma_{\underline{\ell}}(p) := \langle \top, \{o_1.\mathtt{fld}_1\} \rangle \sqcup \langle \top, \{o_1.\mathtt{fld}_2\} \rangle$

$\sigma_{\underline{\ell}}$

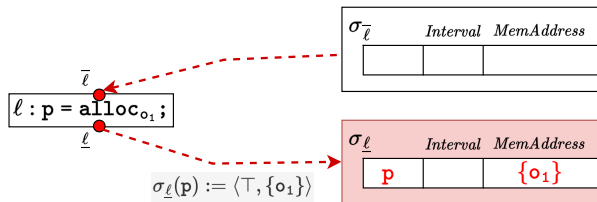|	Interval	MemAddress
q		$\{o_1\}$
i	$[1,2]$	
p		$\{o_1.\mathtt{fld}_1, o_1.\mathtt{fld}_2\}$
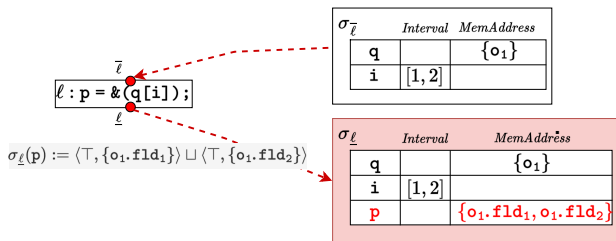
Algorithm 18: Abstract Execution Rule for GEPSTMT
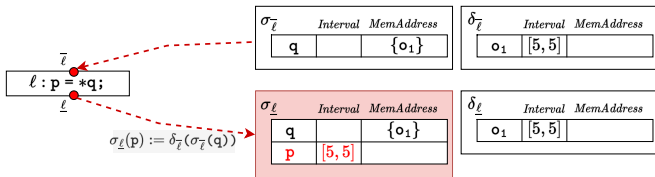
1 **Function** *updateStateOnGep(gep)*:
2   node = gep→getICFGNode();
3   as = getAbsStateFromTrace(node);
4   rhs = gep→getRHSVarID();
5   lhs = gep→getLHSVarID();
6   as[lhs] = as.getGepObjAddrs(rhs, as.getElementIndex(gep));;

Abstract Execution on LOADSTMT

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

$\overline{\ell}$

$\ell : \mathtt{p} = *\mathtt{q};$

$\underline{\ell}$

$\sigma_{\underline{\ell}}(\mathtt{p}) := \delta_{\overline{\ell}}(\sigma_{\overline{\ell}}(\mathtt{q}))$

$\sigma_{\overline{\ell}}$	*Interval*	*MemAddress*
q		$\{o_1\}$

$\delta_{\overline{\ell}}$	*Interval*	*MemAddress*
o_1	$[5,5]$	

$\sigma_{\underline{\ell}}$	*Interval*	*MemAddress*
q		$\{o_1\}$
p	$[5,5]$	

$\delta_{\underline{\ell}}$	*Interval*	*MemAddress*
o_1	$[5,5]$	

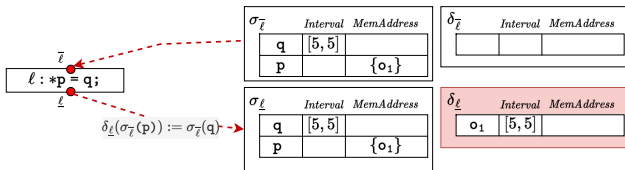**Algorithm 19:** Abstract Execution Rule for LOADSTMT

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 STORESTMT

SVFSTMT	C-Like form	Abstract Execution Rule
STORESTMT	$\ell : *p = q$	$\delta_{\underline{\ell}} := (\{o \mapsto \sigma_{\overline{\ell}}(q) \mid o \in \gamma(\sigma_{\overline{\ell}}(p))\} \sqcup \delta_{\underline{\ell}})$

$\sigma_{\overline{\ell}}$

	Interval	MemAddress
q	$[5,5]$	
p		$\{o_1\}$

$\delta_{\overline{\ell}}$

	Interval	MemAddress

$\overline{\ell}$

$\ell : *p = q;$

$\underline{\ell}$

$\delta_{\underline{\ell}}(\sigma_{\overline{\ell}}(p)) := \sigma_{\overline{\ell}}(q)$

$\sigma_{\underline{\ell}}$

	Interval	MemAddress
q	$[5,5]$	
p		$\{o_1\}$

$\delta_{\underline{\ell}}$

	Interval	MemAddress
o_1	$[5,5]$	

Algorithm 20: 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 in Assignment-3

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
              // Handle svf_assert and OVERFLOW stub function for
              correctness validation;
14            handleStubFunctions(callnode);
15        else
16            // Does not analyze recursive functions in this course;
17            handleCallSite(callnode);
```

Overall Algorithm of Abstract Interpretation in Assignment-3

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 σ for Top-level Variables

```
extern void assert(int);

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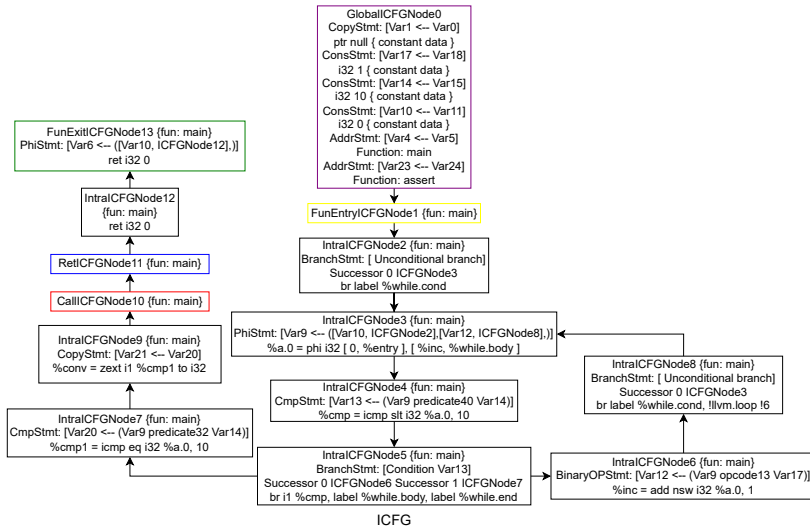
Source Code

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
}
```
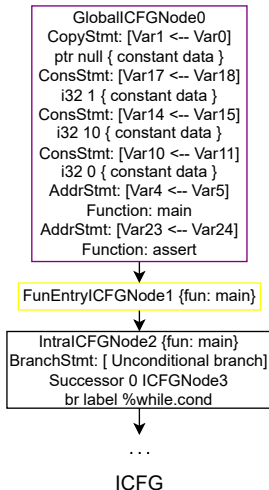
LLVM IR

An Example: Abstract Trace σ for Top-level Variables

ICFG

An Example: Abstract Trace σ for Top-level Variables

GlobalICFGNode0
CopyStmt: [Var1 <-- Var0]
ptr null { constant data }
ConsStmt: [Var17 <-- Var18]
i32 1 { constant data }
ConsStmt: [Var14 <-- Var15]
i32 10 { constant data }
ConsStmt: [Var10 <-- Var11]
i32 0 { constant data }
AddrStmt: [Var4 <-- Var5]
Function: main
AddrStmt: [Var23 <-- Var24]
Function: assert

↓

FunEntryICFGNode1 {fun: main}

↓

IntraICFGNode2 {fun: main}
BranchStmt: [Unconditional branch]
Successor 0 ICFGNode3
br label %while.cond

↓

. . .

ICFG

Algorithm 5: Abstract execution guided by WTO

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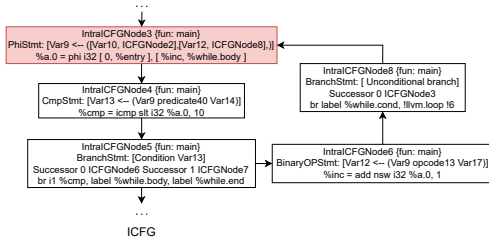

$postAbsTrace[\ell_0].varToAbsVal$:

SVFVar	AbstractValue⟨*Interval*, *MemAddress*⟩
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 σ for Top-level Variables

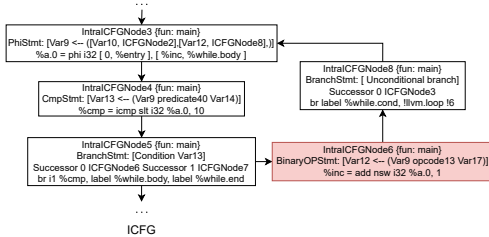**Widen Delay Phase (cur_iter is 0)**

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          // cur_iter ≡ 0,  Options :: WidenDelay() ≡ 2
7          if cur_iter ≥ Options :: WidenDelay() then
8              prev_head_state := postAbsTrace[cycle_head];
9              handleICFGNode(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             handleICFGNode(cycle→head());
22         handleWTOComponents(cycle→getWTOComponents());
23         cur_iter++;
```

An Example: Abstract Trace σ for Top-level Variables

Widen Delay Phase (cur_iter is 0)


```
                        ...
          IntraICFGNode3 {fun: main}
PhiStmt: [Var9 <-- ([Var10, ICFGNode2],[Var12, ICFGNode8],)]
          %a.0 = phi i32 [ 0, %entry ], [ %inc, %while.body ]

          IntraICFGNode4 {fun: main}
CmpStmt: [Var13 <-- (Var9 predicate40 Var14)]
          %cmp = icmp slt i32 %a.0, 10

          IntraICFGNode5 {fun: main}
BranchStmt: [Condition Var13]
Successor 0 ICFGNode6 Successor 1 ICFGNode7
br i1 %cmp, label %while.body, label %while.end
                        ...
                      ICFG
```

```
          IntraICFGNode8 {fun: main}
BranchStmt: [ Unconditional branch]
          Successor 0 ICFGNode3
br label %while.cond, !llvm.loop !6

          IntraICFGNode6 {fun: main}
BinaryOPStmt: [Var12 <-- (Var9 opcode13 Var17)]
          %inc = add nsw i32 %a.0, 1
```

$postAbsTrace[\ell_6].varToAbsVal$:

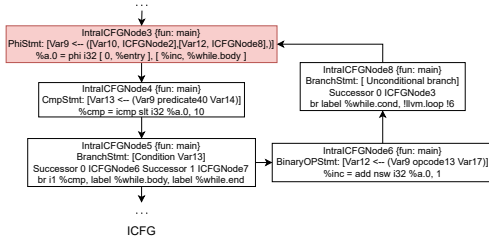| SVFVar	AbstractValue$\langle Interval, MemAddress \rangle$
...	
Var10	$\langle [0,0], \perp \rangle$
Var9	$\langle [0,0], \perp \rangle$
Var12	$\langle [1,1], \perp \rangle$
...	

Algorithm 12: Handle Cycle WTO

```
1  Function handleCycleWTO(cycle):
2      cycle_head := cycle→head()→node() ;
3      increasing := true ;
4      cur_iter := 0 ;
5      while true do
6          // cur_iter ≡ 0, Options :: WidenDelay() ≡ 2;
7          if cur_iter ≥ Options :: WidenDelay() then
8              prev_head_state := postAbsTrace[cycle_head];
9              handleICFGNode(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             handleICFGNode(cycle→head());
22         handleWTOComponents(cycle→getWTOComponents());
23         cur_iter++;
```

An Example: Abstract Trace σ for Top-level Variables

Widen Delay Phase (cur_iter is 1)


```
...
IntraICFGNode3 {fun: main}
PhiStmt: [Var4 <-- ([Var10, ICFGNode2],[Var12, ICFGNode8],)]
%a.0 = phi i32 [ 0, %entry ], [ %inc, %while.body ]

IntraICFGNode4 {fun: main}
CmpStmt: [Var13 <-- (Var9 predicate40 Var14)]
%cmp = icmp slt i32 %a.0, 10

IntraICFGNode5 {fun: main}
BranchStmt: [Condition Var13]
Successor 0 ICFGNode6 Successor 1 ICFGNode7
br i1 %cmp, label %while.body, label %while.end

IntraICFGNode8 {fun: main}
BranchStmt: [ Unconditional branch]
Successor 0 ICFGNode3
br label %while.cond, !llvm.loop !6

IntraICFGNode6 {fun: main}
BinaryOPStmt: [Var12 <-- (Var9 opcode13 Var17)]
%inc = add nsw i32 %a.0, 1
...
ICFG
```

$postAbsTrace[\ell_3].varToAbsVal$:

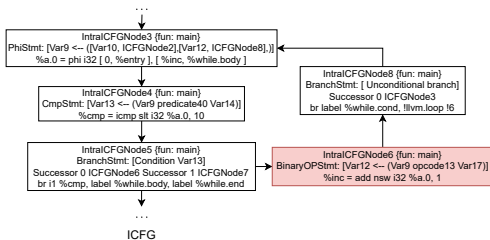| SVFVar	AbstractValue⟨Interval, MemAddress⟩
...	
Var9	⟨[0, 1], ⊥⟩
Var12	⟨[1, 1], ⊥⟩
...	

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         // cur_iter ≡ 1,  Options :: WidenDelay() ≡ 2;
7         if cur_iter ≥ Options :: WidenDelay() then
8             prev_head_state := postAbsTrace[cycle_head];
9             handleICFGNode(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            handleICFGNode(cycle→head());
22        handleWTOComponents(cycle→getWTOComponents());
23        cur_iter++ ;
```

An Example: Abstract Trace σ for Top-level Variables

Widen Delay Phase (cur_iter is 1)

ICFG

$postAbsTrace[\ell_6].varToAbsVal$:

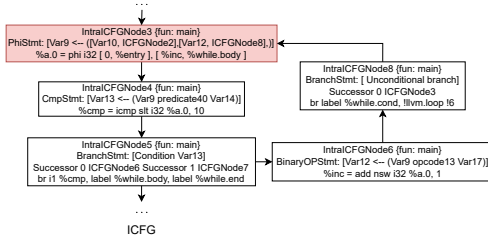| SVFVar	AbstractValue$\langle$*Interval*, *MemAddress*$\rangle$
...	
Var9	$\langle[0, 1], \perp\rangle$
Var12	$\langle[1, 2], \perp\rangle$
...	

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          // cur_iter ≡ 1,  Options :: WidenDelay() ≡ 2;
 7          if cur_iter ≥ Options :: WidenDelay() then
 8              prev_head_state := postAbsTrace[cycle_head] ;
 9              handleICFGNode(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              handleICFGNode(cycle→head()) ;
22          handleWTOComponents(cycle→getWTOComponents()) ;
23          cur_iter++ ;
```

Widen Phase (cur_iter is 2)

An Example: Abstract Trace σ for Top-level Variables

Widen Phase (cur_iter is 2)

ICFG

$postAbsTrace[\ell_6].varToAbsVal$:

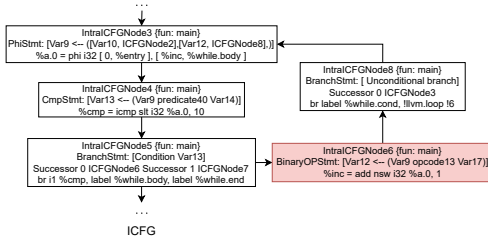| SVFVar	AbstractValue⟨*Interval*, *MemAddress*⟩
	...
Var9	⟨[0, 9], ⊥⟩
Var12	⟨[1, 10], ⊥⟩
	...

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          // cur_iter ≡ 2, Options :: WidenDelay() ≡ 2
7          if cur_iter ≥ Options :: WidenDelay() then
8              prev_head_state := postAbsTrace[cycle_head];
9              handleICFGNode(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             handleICFGNode(cycle→head());
22         handleWTOComponents(cycle→getWTOComponents())
23         cur_iter++ ;
```

An Example: Abstract Trace σ for Top-level Variables
Widen Phase Fixed Point

...

IntraICFGNode3 {fun: main}
PhiStmt: [Var9 <-- ([Var10, ICFGNode2],[Var12, ICFGNode8],)]
%a.0 = phi i32 [0, %entry], [%inc, %while.body]

IntraICFGNode4 {fun: main}
CmpStmt: [Var13 <-- (Var9 predicate40 Var14)]
%cmp = icmp slt i32 %a.0, 10

IntraICFGNode5 {fun: main}
BranchStmt: [Condition Var13]
Successor 0 ICFGNode6 Successor 1 ICFGNode7
br i1 %cmp, label %while.body, label %while.end

IntraICFGNode8 {fun: main}
BranchStmt: [Unconditional branch]
Successor 0 ICFGNode3
br label %while.cond, !llvm.loop !6

IntraICFGNode6 {fun: main}
BinaryOPStmt: [Var12 <-- (Var9 opcode13 Var17)]
%inc = add nsw i32 %a.0, 1

...

ICFG

$postAbsTrace[\ell_3].varToAbsVal$:

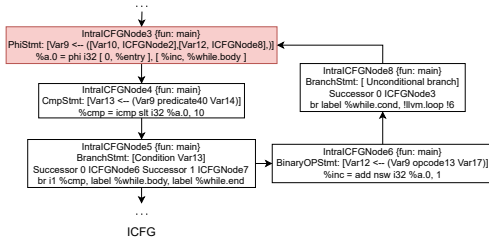| SVFVar	$\langle Interval, MemAddress \rangle$
...	
Var9	$\langle [0, +\infty], \bot \rangle$
Var12	$\langle [1, +\infty], \bot \rangle$
...	

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          // cur_iter ≡ 3, Options :: WidenDelay() ≡ 2
7          if cur_iter > Options :: WidenDelay() then
8              prev_head_state := postAbsTrace[cycle_head];
9              handleICFGNode(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             handleICFGNode(cycle→head());
22         handleWTOComponents(cycle→getWTOComponents());
23         cur_iter++ ;
```

ICFG

$postAbsTrace[\ell_3].varToAbsVal$:

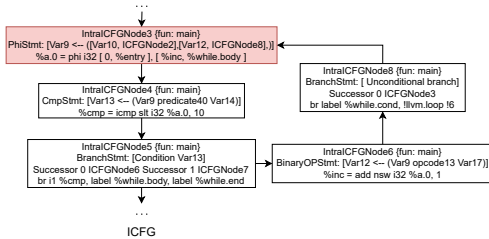| SVFVar	$\langle Interval, MemAddress \rangle$
...	
Var9	$\langle [0, 10], \bot \rangle$
Var12	$\langle [1, 10], \bot \rangle$
...	

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          // cur_iter ≡ 3, Options :: WidenDelay() ≡ 2
7          if cur_iter ≥ Options :: WidenDelay() then
8              prev_head_state := postAbsTrace[cycle_head];
9              handleICFGNode(cycle→head()) // increasing ≡ false;
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             handleICFGNode(cycle→head());
22         handleWTOComponents(cycle→getWTOComponents());
23         cur_iter++ ;
```

An Example: Abstract Trace σ for Top-level Variables
Narrow Phase

...

IntraICFGNode3 {fun: main}
PhiStmt: [Var9 <-- ([Var10, ICFGNode2],[Var12, ICFGNode8],)]
%a.0 = phi i32 [0, %entry], [%inc, %while.body]

IntraICFGNode4 {fun: main}
CmpStmt: [Var13 <-- (Var9 predicate40 Var14)]
%cmp = icmp slt i32 %a.0, 10

IntraICFGNode5 {fun: main}
BranchStmt: [Condition Var13]
Successor 0 ICFGNode6 Successor 1 ICFGNode7
br i1 %cmp, label %while.body, label %while.end

IntraICFGNode8 {fun: main}
BranchStmt: [Unconditional branch]
Successor 0 ICFGNode3
br label %while.cond, !llvm.loop !6

IntraICFGNode6 {fun: main}
BinaryOPStmt: [Var12 <-- (Var9 opcode13 Var17)]
%inc = add nsw i32 %a.0, 1

...

ICFG

$postAbsTrace[\ell_6].varToAbsVal$:

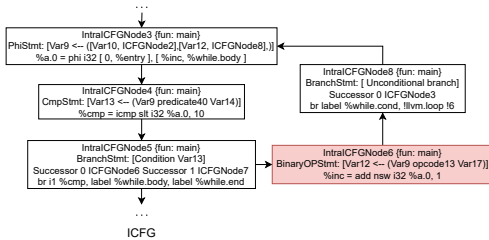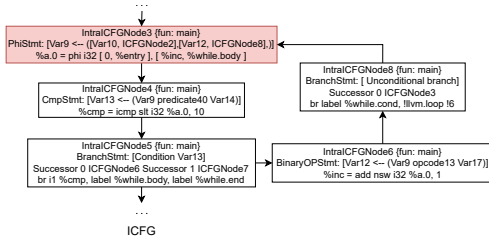| SVFVar	$\langle Interval, MemAddress \rangle$
...	...
Var9	$\langle [0,9], \perp \rangle$
Var12	$\langle [1,10], \perp \rangle$
...	...

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   // cur_iter $\equiv 3$, Options :: WidenDelay() $\equiv 2$
7   **if** cur_iter $\geq$ Options :: WidenDelay() **then**
8   prev_head_state := postAbsTrace[cycle_head] ;
9   handleICFGNode(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   handleICFGNode(cycle→head());
22   handleWTOComponents(cycle→getWTOComponents()) ;
23   cur_iter++;

An Example: Abstract Trace σ for Top-level Variables
Narrow Phase Fixed Point

...

IntraICFGNode3 {fun: main}
PhiStmt: [Var9 <-- ([Var10, ICFGNode2],[Var12, ICFGNode8],)]
%a.0 = phi i32 [0, %entry], [%inc, %while.body]

IntraICFGNode4 {fun: main}
CmpStmt: [Var13 <-- (Var9 predicate40 Var14)]
%cmp = icmp slt i32 %a.0, 10

IntraICFGNode5 {fun: main}
BranchStmt: [Condition Var13]
Successor 0 ICFGNode6 Successor 1 ICFGNode7
br i1 %cmp, label %while.body, label %while.end

IntraICFGNode8 {fun: main}
BranchStmt: [Unconditional branch]
Successor 0 ICFGNode3
br label %while.cond, !llvm.loop !6

IntraICFGNode6 {fun: main}
BinaryOPStmt: [Var12 <-- (Var9 opcode13 Var17)]
%inc = add nsw i32 %a.0, 1

ICFG

$postAbsTrace[\ell_3].varToAbsVal$:

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

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         // cur_iter ≡ 4,  Options :: WidenDelay() ≡ 2
7         if cur_iter ≥ Options :: WidenDelay() then
8             prev_head_state := postAbsTrace[cycle_head];
9             handleICFGNode(cycle→head()) // increasing ≡ false;
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            handleICFGNode(cycle→head());
22        handleWTOComponents(cycle→getWTOComponents());
23        cur_iter++ ;
```

An Example: Abstract Trace σ for Top-level Variables

FunExitICFGNode13 {fun: main}
PhiStmt: [Var6 <-- ([Var10, ICFGNode12],)]
ret i32 0

IntraICFGNode12
{fun: main}
ret i32 0

RetICFGNode11 {fun: main}

CallICFGNode10 {fun: main}

IntraICFGNode9 {fun: main}
CopyStmt: [Var21 <-- Var20]
%conv = zext i1 %cmp1 to i32

IntraICFGNode7 {fun: main}
CmpStmt: [Var20 <-- (Var9 predicate32 Var14)]
%cmp1 = icmp eq i32 %a.0, 10

. . .

ICFG

Algorithm 13: Abstract execution guided by WTO

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

$postAbsTrace[\ell_7].varToAbsVal$:

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