

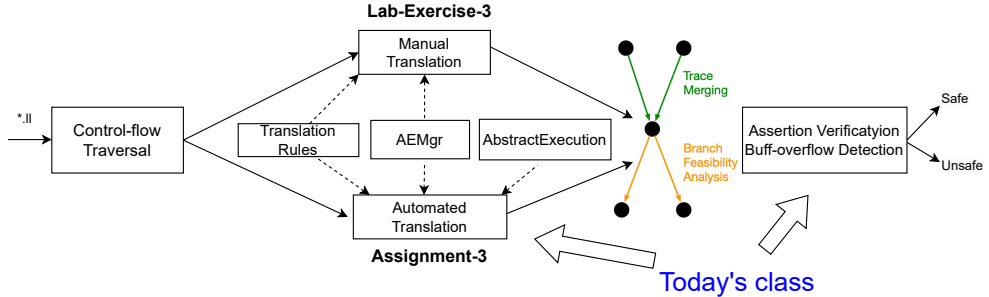
Abstract Interpretation for Code Analysis and Verification

(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

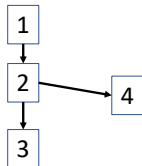
? How to analyze a program **free of loop**?

✓ 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

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.
 - We will practice loop handling using WTO in Assignment-3. Function recursions will not be handled in this Assignment.

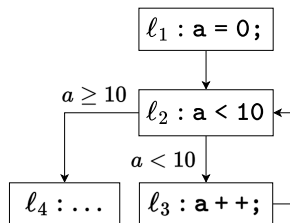
Weak Topological Order

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

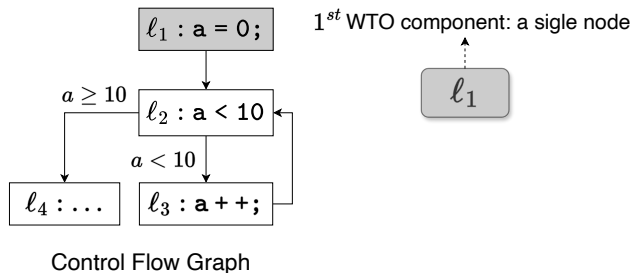
Weak Topological Order

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?



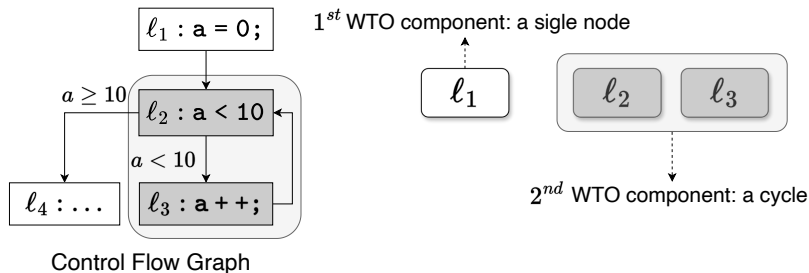
Weak Topological Order

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?



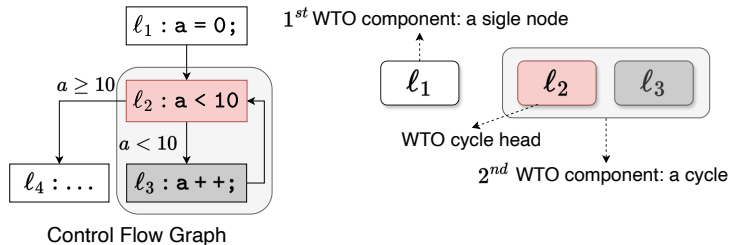
Weak Topological Order

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?



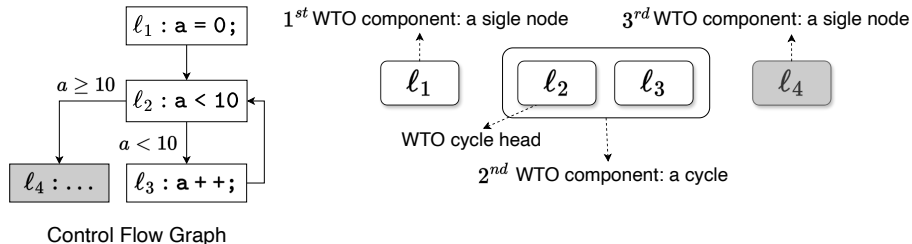
Weak Topological Order

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?

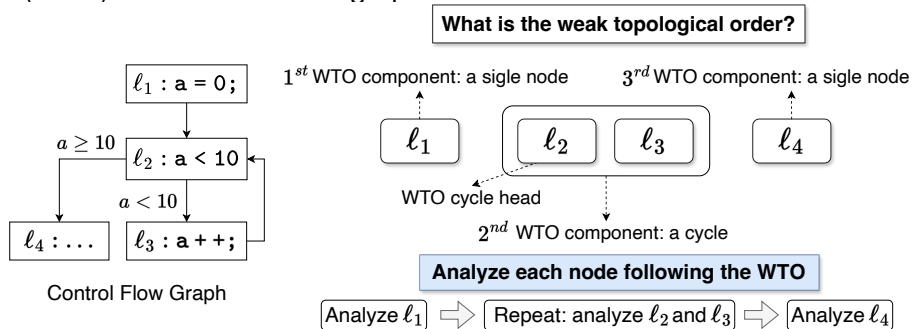


Weak Topological Order

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.

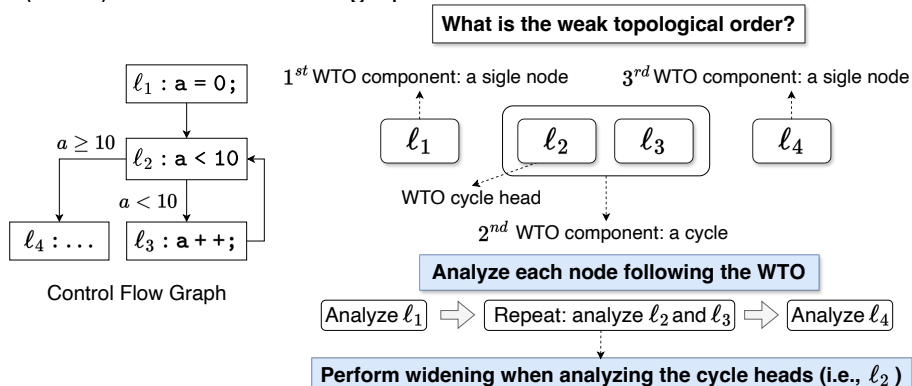


Weak Topological Order

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 (WTO)?

- 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)

Revisit the Notations and Data Structure

- An **abstract trace** $\sigma \in \mathbb{L} \times \mathcal{V} \rightarrow \mathbb{A}$ represents a list of abstract states before $(\bar{\ell})$ and after $(\underline{\ell})$ each program statement ℓ (`preAbsTrace` and `postAbsTrace`).
- An **abstract state** (`AbstractState` in Lab-3 and Assignment-3) is defined as a map $AS : \mathcal{V} \rightarrow \mathbb{A}$ associating program variables \mathcal{V} with an abstract value in \mathbb{A} , approximating the runtime states of program variables.

Revisit the Notations and Data Structure

- An **abstract trace** $\sigma \in \mathbb{L} \times \mathcal{V} \rightarrow \mathbb{A}$ represents a list of abstract states before ($\bar{\ell}$) and after ($\underline{\ell}$) each program statement ℓ (preAbsTrace and postAbsTrace).
- An **abstract state** (AbstractState in Lab-3 and Assignment-3) is defined as a map $AS : \mathcal{V} \rightarrow \mathbb{A}$ associating program variables \mathcal{V} with an abstract value in \mathbb{A} , approximating the runtime states of program variables.
- An **abstract value** can be either an interval or a memory address.

	Domain	Notation	SSE Data Structure
Abstract trace	$\mathbb{L} \times \mathcal{V} \rightarrow \mathbb{A}$	σ	preAbsTrace: trace before ICFGNodes postAbsTrace: trace after ICFGNodes
Abstract state at $L \in \mathbb{L}$	$\mathcal{V} \rightarrow \mathbb{A}$	$\sigma_{\bar{\ell}}$ $\sigma_{\underline{\ell}}$	preAbsTrace[node]: state before node ℓ postAbsTrace[node]: state after node ℓ
Abstract value of varId at $L \in \mathbb{L}$	\mathbb{A}	$\sigma_{\underline{\ell}}(\text{varId})$	as = postAbsTrace[node] as[VarID]: value of varId after node ℓ

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   handleFunction(mainFun);
```

Algorithm 2: Handle Function

```
1 Function handleFunction(fun):
2   worklist := [funEntryICFGNode] while worklist  $\neq \emptyset$  do
3     n := worklist.pop-front();
4     if n is a cycle head then
5       cycle := cycle.head.to-cycle[n];
6       handleICFGCycle(cycle); // Assignment-3
7       foreach n'  $\in$  getNextNodesOfCycle(cycle) do
8         worklist.push-back(n');
9     else
10      if handleICFGNode(n) == false then
11        foreach n'  $\in$  getNextNodes(n) do
12          worklist.push-back(n');
```

Algorithm 3: Handle ICFG Node

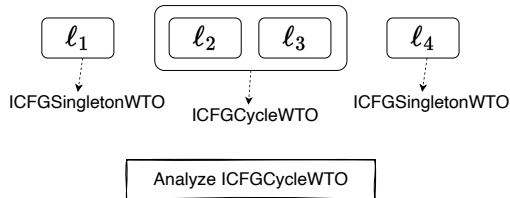
```
1 Function handleICFGNode(n):
2   feasible, aspre := mergeStatesFromPredecessors(node);
3   if !feasible then
4     return false;
5   aslast :=  $\sigma_n$ ;
6    $\sigma_n$  := aspre;
7   foreach stmt  $\in$  n  $\rightarrow$  getSVFStmts() do
8     updateAbsState(stmt); // Assignment-3
9     bufOverflowDetection(stmt); // Assignment-3
10
11  if n is CallICFGNode then
12    // Handle stub function and external call;
13    // Skip recursive call, not handling recursive functions in
    Assignment-3;
14    // Handle normal call;
15  if  $\sigma_n \equiv as_{last}$  then
16    return false; // state not changed
17  return true; // state changed
```

Handling Loop in Assignment-3

Algorithm 4: Handle ICFG Cycle

```
1 Function handleICFGCycle (cycle):
2    $\ell := \text{cycle.getHead().getICFGNode}();$  // cycle head ICFGNode  $\ell$ 
3   increasing := true;
4   i := 0; // analysis iteration for the loop
5   while true do
6      $\text{as}_{\text{pre}} := \sigma_{\ell};$  // abstract state in the last iteration
7     handleICFGNode( $\ell$ );
8      $\text{as}_{\text{cur}} := \sigma_{\ell};$  // abstract state in the current iteration
9     if  $i \geq \text{Options.WidenDelay}()$  then
10      if increasing then
11         $\sigma_{\ell} := \text{as}_{\text{pre}} \nabla \text{as}_{\text{cur}};$  // widening
12        if  $\sigma_{\ell} \equiv \text{as}_{\text{pre}}$  then
13          increasing := false;
14          continue;
15      else
16         $\sigma_{\ell} := \text{as}_{\text{pre}} \Delta \text{as}_{\text{cur}};$  // narrowing
17        if  $\sigma_{\ell} \equiv \text{as}_{\text{pre}}$  then
18          break;
19      // analyze remaining cycle components after two fixed-points
20      foreach  $\text{comp} \in \text{cycle.getWTOComponents}()$  do
21        if comp is Singleton then
22          handleICFGNode( $\text{comp.getICFGNode}()$ )
23        else if comp is Cycle then
24          handleICFGCycle(comp);
25      i++;
26  return;
```

Widening and Narrowing



ICFGWTOComp has two subclasses:

`ICFGSingletonWTO` and `ICFGCycleWTO`.

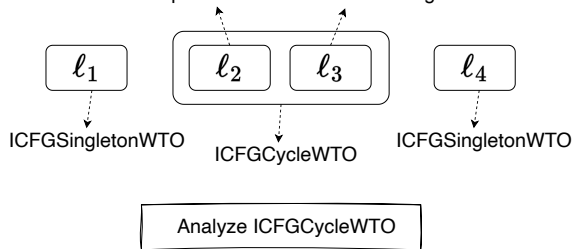
Use `dyn_cast` to identify the subclass of an instance.

Algorithm 12: Handle ICFG Cycle

```
1 Function handleICFGCycle (cycle):
2    $\ell := \text{cycle.getHead().getICFGNode}();$  // cycle head ICFGNode  $\ell$ 
3   increasing := true;
4    $i := 0;$  // analysis iteration for the loop
5   while true do
6      $\text{as}_{\text{pre}} := \sigma_{\ell};$  // abstract state in the last iteration
7     handleICFGNode( $\ell$ );
8      $\text{as}_{\text{cur}} := \sigma_{\ell};$  // abstract state in the current iteration
9     if  $i \geq \text{Options.WidenDelay}()$  then
10       if increasing then
11          $\sigma_{\ell} := \text{as}_{\text{pre}} \nabla \text{as}_{\text{cur}};$  // widening
12         if  $\sigma_{\ell} \equiv \text{as}_{\text{pre}}$  then
13           increasing := false;
14           continue;
15       else
16          $\sigma_{\ell} := \text{as}_{\text{pre}} \Delta \text{as}_{\text{cur}};$  // narrowing
17         if  $\sigma_{\ell} \equiv \text{as}_{\text{pre}}$  then
18           break;
19       // analyze remaining cycle components after two fixed-points
20       foreach  $\text{comp} \in \text{cycle.getWTOComponents}()$  do
21         if comp is Singleton then
22           handleICFGNode(comp.getICFGNode());
23         else if comp is Cycle then
24           handleICFGCycle(comp);
25        $i++;$ 
26   return;
```

Widening and Narrowing

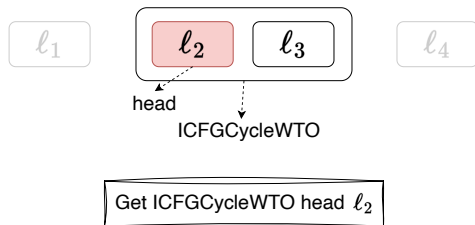
Sub WTO Components: each is an ICFGSingletonWTO



Algorithm 12: Handle ICFG Cycle

```
1 Function handleICFGCycle (cycle):
2    $\ell := \text{cycle.getHead().getICFGNode}();$  // cycle head ICFGNode  $\ell$ 
3   increasing := true;
4   i := 0; // analysis iteration for the loop
5   while true do
6      $\text{as}_{\text{pre}} := \sigma_{\ell};$  // abstract state in the last iteration
7     handleICFGNode( $\ell$ );
8      $\text{as}_{\text{cur}} := \sigma_{\ell};$  // abstract state in the current iteration
9     if i  $\geq$  Options.WidenDelay() then
10      if increasing then
11         $\sigma_{\ell} := \text{as}_{\text{pre}} \nabla \text{as}_{\text{cur}};$  // widening
12        if  $\sigma_{\ell} \equiv \text{as}_{\text{pre}}$  then
13          increasing := false;
14          continue;
15      else
16         $\sigma_{\ell} := \text{as}_{\text{pre}} \Delta \text{as}_{\text{cur}};$  // narrowing
17        if  $\sigma_{\ell} \equiv \text{as}_{\text{pre}}$  then
18          break;
19      // analyze remaining cycle components after two fixed-points
20      foreach comp  $\in$  cycle.getWTOComponents() do
21        if comp is Singleton then
22          handleICFGNode(comp.getICFGNode())
23        else if comp is Cycle then
24          handleICFGCycle(comp);
25      i++;
26  return;
```

Weak Topological Order

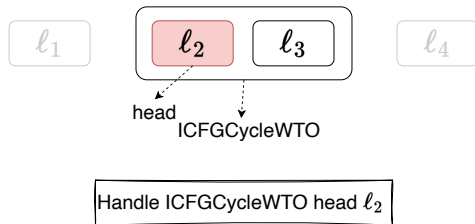


Algorithm 12: Handle ICFG Cycle

```

1 Function handleICFGCycle (cycle):
2    $\ell := \text{cycle.getHead().getICFGNode()};$  // cycle head ICFGNode  $\ell$ 
3   increasing := true;
4   i := 0; // analysis iteration for the loop
5   while true do
6      $\text{as}_{\text{pre}} := \sigma_{\ell};$  // abstract state in the last iteration
7     handleICFGNode( $\ell$ );
8      $\text{as}_{\text{cur}} := \sigma_{\ell};$  // abstract state in the current iteration
9     if i  $\geq$  Options.WidenDelay() then
10      if increasing then
11         $\sigma_{\ell} := \text{as}_{\text{pre}} \nabla \text{as}_{\text{cur}};$  // widening
12        if  $\sigma_{\ell} \equiv \text{as}_{\text{pre}}$  then
13          increasing := false;
14          continue;
15      else
16         $\sigma_{\ell} := \text{as}_{\text{pre}} \Delta \text{as}_{\text{cur}};$  // narrowing
17        if  $\sigma_{\ell} \equiv \text{as}_{\text{pre}}$  then
18          break;
19      // analyze remaining cycle components after two fixed-points
20      foreach comp  $\in$  cycle.getWTOComponents() do
21        if comp is Singleton then
22          handleICFGNode(comp.getICFGNode())
23        else if comp is Cycle then
24          handleICFGCycle(comp);
25      i++;
26  return;
  
```

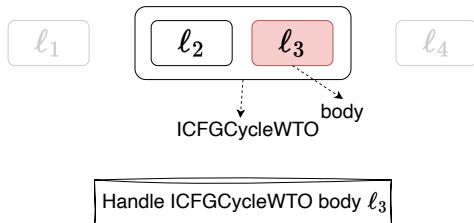
Weak Topological Order



Algorithm 12: Handle ICFG Cycle

```
1 Function handleICFGCycle(cycle):  
2    $\ell := \text{cycle.getHead().getICFGNode()};$  // cycle head ICFGNode  $\ell$   
3    $\text{increasing} := \text{true};$   
4    $i := 0;$  // analysis iteration for the loop  
5   while true do  
6      $\text{as}_{\text{pre}} := \sigma_{\ell};$  // abstract state in the last iteration  
7     handleICFGNode( $\ell$ );  
8      $\text{as}_{\text{cur}} := \sigma_{\ell};$  // abstract state in the current iteration  
9     if  $i \geq \text{Options.WidenDelay}()$  then  
10      if increasing then  
11         $\sigma_{\ell} := \text{as}_{\text{pre}} \nabla \text{as}_{\text{cur}};$  // widening  
12        if  $\sigma_{\ell} \equiv \text{as}_{\text{pre}}$  then  
13           $\text{increasing} := \text{false};$   
14          continue;  
15      else  
16         $\sigma_{\ell} := \text{as}_{\text{pre}} \Delta \text{as}_{\text{cur}};$  // narrowing  
17        if  $\sigma_{\ell} \equiv \text{as}_{\text{pre}}$  then  
18          break;  
19      // analyze remaining cycle components after two fixed-points  
20      foreach  $\text{comp} \in \text{cycle.getWTOComponents}()$  do  
21        if comp is Singleton then  
22          handleICFGNode( $\text{comp.getICFGNode}()$ )  
23        else if comp is Cycle then  
24          handleICFGCycle( $\text{comp}$ );  
25       $i++;$   
26      return;
```

Widening and Narrowing



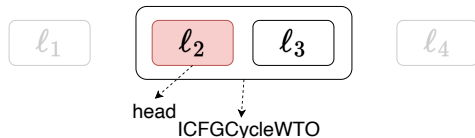
Algorithm 12: Handle ICFG Cycle

```

1 Function handleICFGCycle (cycle):
2    $\ell := \text{cycle.getHead().getICFGNode}()$ ; // cycle head ICFGNode  $\ell$ 
3   increasing := true;
4   i := 0; // analysis iteration for the loop
5   while true do
6      $\text{as}_{\text{pre}} := \sigma_{\ell}$ ; // abstract state in the last iteration
7     handleICFGNode(h);
8      $\text{as}_{\text{cur}} := \sigma_{\ell}$ ; // abstract state in the current iteration
9     if i  $\geq$  Options.WidenDelay() then
10       if increasing then
11          $\sigma_{\ell} := \text{as}_{\text{pre}} \nabla \text{as}_{\text{cur}}$ ; // widening
12         if  $\sigma_{\ell} \equiv \text{as}_{\text{pre}}$  then
13           increasing := false;
14           continue;
15       else
16          $\sigma_{\ell} := \text{as}_{\text{pre}} \Delta \text{as}_{\text{cur}}$ ; // narrowing
17         if  $\sigma_{\ell} \equiv \text{as}_{\text{pre}}$  then
18           break;
19       // analyze remaining cycle components after two fixed-points
20       foreach comp  $\in$  cycle.getWTOComponents() do
21         if comp is Singleton then
22           handleICFGNode(comp.getICFGNode())
23         else if comp is Cycle then
24           handleICFGCycle(comp);
25       i++;
26   return;
  
```

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

Widening and Narrowing

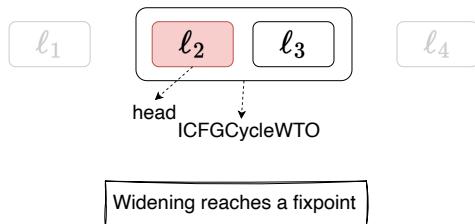


When $cur_iter \geq Options :: WidenDelay()$
perform widening on l_2

Algorithm 12: Handle ICFG Cycle

```
1 Function handleICFGCycle (cycle):  
2    $\ell := cycle.getHead().getICFGNode();$  // cycle head ICFGNode  $\ell$   
3    $increasing := true;$   
4    $i := 0;$  // analysis iteration for the loop  
5   while  $true$  do  
6      $as_{pre} := \sigma_{\ell};$  // abstract state in the last iteration  
7     handleICFGNode( $\ell$ );  
8      $as_{cur} := \sigma_{\ell};$  // abstract state in the current iteration  
9     if  $i \geq Options.WidenDelay()$  then  
10      if  $increasing$  then  
11         $\sigma_{\ell} := as_{pre} \nabla as_{cur};$  // widening  
12        if  $\sigma_{\ell} \equiv as_{pre}$  then  
13           $increasing := false;$   
14          continue;  
15        else  
16           $\sigma_{\ell} := as_{pre} \Delta as_{cur};$  // narrowing  
17          if  $\sigma_{\ell} \equiv as_{pre}$  then  
18            break;  
19      // analyze remaining cycle components after two fixed-points  
20      foreach  $comp \in cycle.getWTOComponents()$  do  
21        if  $comp$  is Singleton then  
22          handleICFGNode( $comp.getICFGNode()$ )  
23        else if  $comp$  is Cycle then  
24          handleICFGCycle( $comp$ );  
25       $i++;$   
26      return;
```

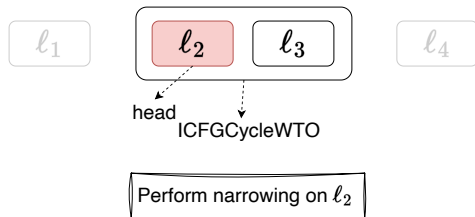
Widening and Narrowing



Algorithm 12: Handle ICFG Cycle

```
1 Function handleICFGCycle (cycle):
2    $\ell := \text{cycle.getHead().getICFGNode()};$  // cycle head ICFGNode  $\ell$ 
3   increasing := true;
4    $i := 0;$  // analysis iteration for the loop
5   while true do
6      $\text{as}_{\text{pre}} := \sigma_{\ell};$  // abstract state in the last iteration
7     handleICFGNode( $\ell$ );
8      $\text{as}_{\text{cur}} := \sigma_{\ell};$  // abstract state in the current iteration
9     if  $i \geq \text{Options.WidenDelay}()$  then
10      if increasing then
11         $\sigma_{\ell} := \text{as}_{\text{pre}} \nabla \text{as}_{\text{cur}};$  // widening
12        if  $\sigma_{\ell} \equiv \text{as}_{\text{pre}}$  then
13          increasing := false;
14          continue;
15      else
16         $\sigma_{\ell} := \text{as}_{\text{pre}} \Delta \text{as}_{\text{cur}};$  // narrowing
17        if  $\sigma_{\ell} \equiv \text{as}_{\text{pre}}$  then
18          break;
19      // analyze remaining cycle components after two fixed-points
20      foreach  $\text{comp} \in \text{cycle.getWTOComponents}()$  do
21        if comp is Singleton then
22          handleICFGNode(comp.getICFGNode())
23        else if comp is Cycle then
24          handleICFGCycle(comp);
25       $i++;$ 
26  return;
```


Widening and Narrowing

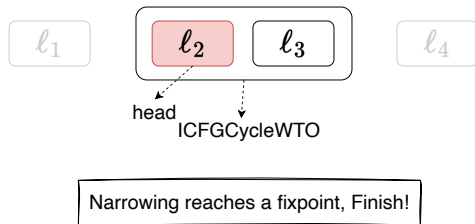


Algorithm 12: Handle ICFCycle

```

1 Function handleICFCycle (cycle):
2    $\ell := \text{cycle.getHead().getICFGNode()};$  // cycle head ICFGNode  $\ell$ 
3   increasing := true;
4    $i := 0;$  // analysis iteration for the loop
5   while true do
6      $as_{pre} := \sigma_{\ell};$  // abstract state in the last iteration
7     handleICFGNode( $\ell$ );
8      $as_{cur} := \sigma_{\ell};$  // abstract state in the current iteration
9     if  $i \geq \text{Options.WidenDelay}()$  then
10      if increasing then
11         $\sigma_{\ell} := as_{pre} \nabla as_{cur};$  // widening
12        if  $\sigma_{\ell} \equiv as_{pre}$  then
13          increasing := false;
14          continue;
15      else
16         $\sigma_{\ell} := as_{pre} \Delta as_{cur};$  // narrowing
17        if  $\sigma_{\ell} \equiv as_{pre}$  then
18          break;
19      // analyze remaining cycle components after two fixed-points
20      foreach comp  $\in \text{cycle.getWTOComponents}()$  do
21        if comp is Singleton then
22          handleICFGNode(comp.getICFGNode())
23        else if comp is Cycle then
24          handleICFCycle(comp);
25       $i++;$ 
26   return;
  
```

Widening and Narrowing



Algorithm 12: Handle ICFCycle

```
1 Function handleICFCycle (cycle):  
2    $\ell := \text{cycle.getHead().getICFGNode()};$  // cycle head ICFGNode  $\ell$   
3   increasing := true;  
4    $i := 0;$  // analysis iteration for the loop  
5   while true do  
6      $as_{pre} := \sigma_{\ell};$  // abstract state in the last iteration  
7     handleICFGNode( $\ell$ );  
8      $as_{cur} := \sigma_{\ell};$  // abstract state in the current iteration  
9     if  $i \geq \text{Options.WidenDelay()}$  then  
10      if increasing then  
11         $\sigma_{\ell} := as_{pre} \nabla as_{cur};$  // widening  
12        if  $\sigma_{\ell} \equiv as_{pre}$  then  
13          increasing := false;  
14          continue;  
15      else  
16         $\sigma_{\ell} := as_{pre} \Delta as_{cur};$  // narrowing  
17        if  $\sigma_{\ell} \equiv as_{pre}$  then  
18          break;  
19      // analyze remaining cycle components after two fixed-points  
20      foreach comp  $\in \text{cycle.getWTOComponents()}$  do  
21        if comp is Singleton then  
22          handleICFGNode(comp.getICFGNode())  
23        else if comp is Cycle then  
24          handleICFCycle(comp);  
25       $i++;$   
26      return;
```

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

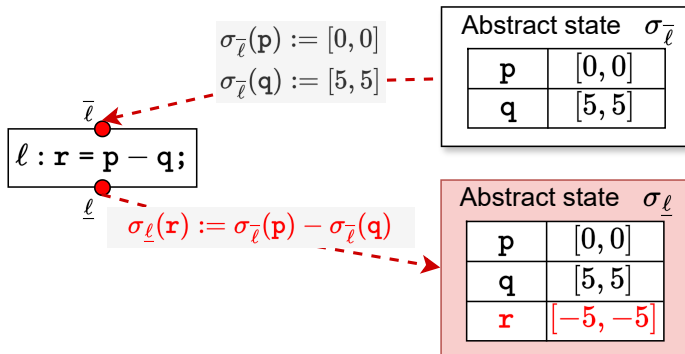
Interval Domain

Let's use the *Interval* abstract domain to update σ 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	$\ell : r = p \otimes q$	$\sigma_{\underline{\ell}}(r) := \sigma_{\bar{\ell}}(p) \hat{\otimes} \sigma_{\bar{\ell}}(q)$

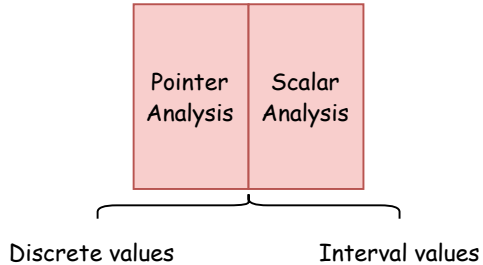


Abstract Interpretation in the Presence of Pointers

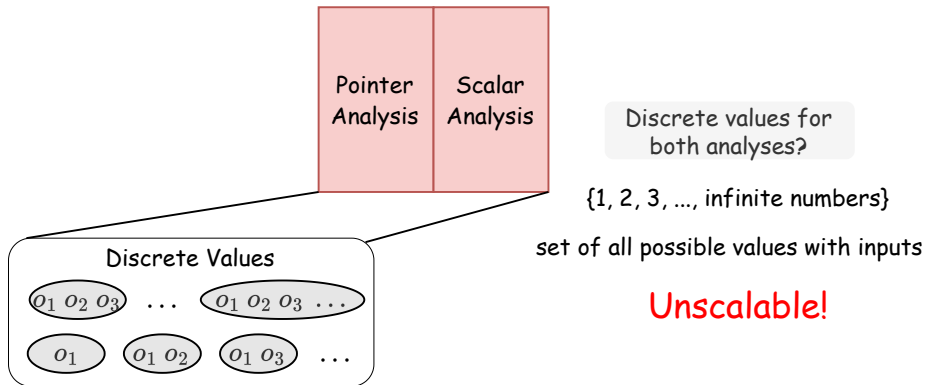
- SVFIR in the presence of pointers contain pointer-related statements including ADDRSTMT, GEPSTMT, LOADSTMT and STORESTMT.
- Abstract interpretation needs to be performed on **a combined domain of intervals and addresses**.

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	$\ell : *p = q$

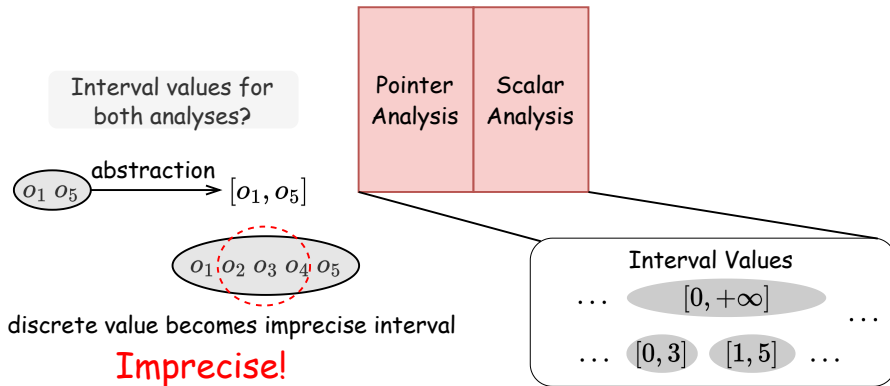
Combined Analysis



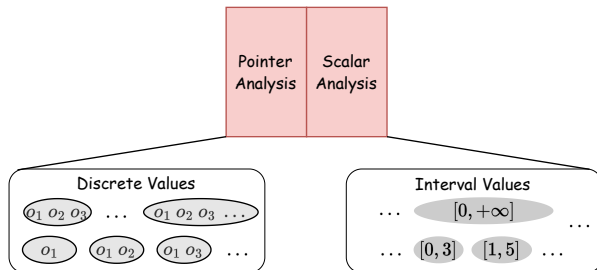
Combined Analysis Using Discrete Values



Combined Analysis Using Interval Values



Abstract Interpretation Over a Combined Domain



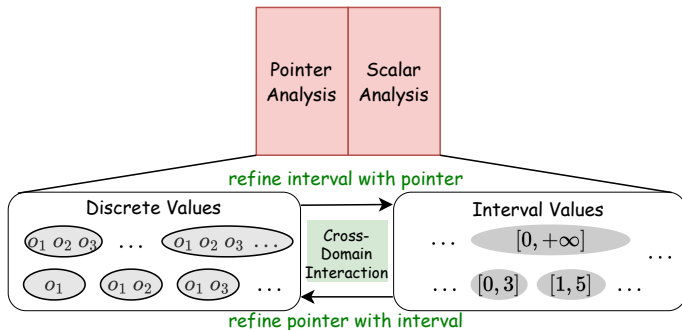
$p = \text{malloc}(\textcolor{red}{m} * \text{sizeof}(\text{int}));$ // p points to an array of size m

$q = \text{malloc}(n * \text{sizeof}(\text{int}));$ // 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

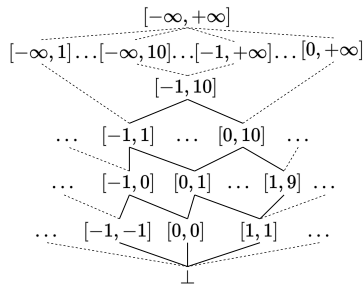


We require **a combination of interval and memory address domains** to precisely and efficiently perform abstract execution on SVFIR in the presence of pointers.

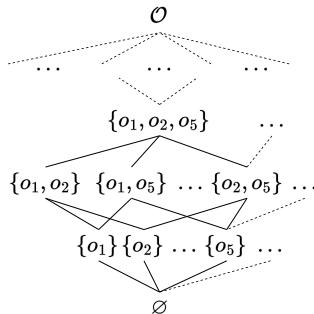
Precise Sparse Abstract Execution via Cross-Domain Interaction, ICSE 2024

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

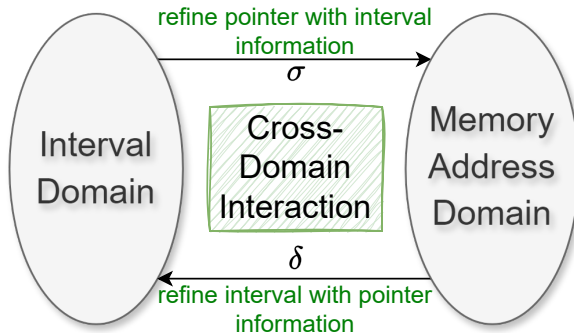
	Notation	Domain	Data Structure Implementation
Abstract trace	σ	$\mathbb{L} \times \mathbb{P} \rightarrow \text{Interval} \times \text{MemAddress}$	$preAbsTrace, postAbsTrace$
	δ	$\mathbb{L} \times \mathbb{O} \rightarrow \text{Interval} \times \text{MemAddress}$	
Abstract state	σ_L	$\mathbb{P} \rightarrow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

- *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 Trace and State in Assignment-3

- For a program point L , $AEState$ consists of:
 - Top-level variable, $varToAbsVal : \sigma_L \in \mathbb{P} \rightarrow Interval \times MemAddress$
 - Memory object, $addrToAbsVal : \delta_L \in \mathbb{O} \rightarrow Interval \times MemAddress$
- The abstract trace has two maps, $preAbsTrace$ and $postAbsTrace$, which maintains abstract states before and after each `ICFGNode` respectively.
 - For an `ICFGNode` ℓ , $preAbsTrace(\ell)$ 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 σ_{ℓ} and use `storeValue` and `loadValue` to operate δ_{ℓ} .
 - Each variable's `AbstractValue` (e.g., `as[VarId]`) is initialized as \perp in an `AbstractState` before assigned a new value. An `uninitialized variable` is assigned with \top for over-approximation.
 - 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



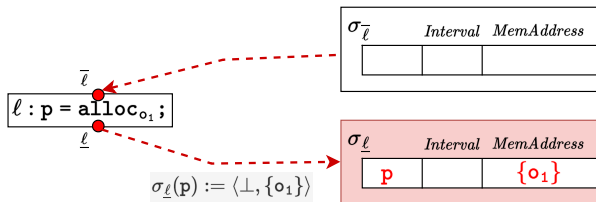
Abstract Execution Rules on SVFIR in the Presence of Pointers

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

Abstract Interpretation on ADDRSTMT

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



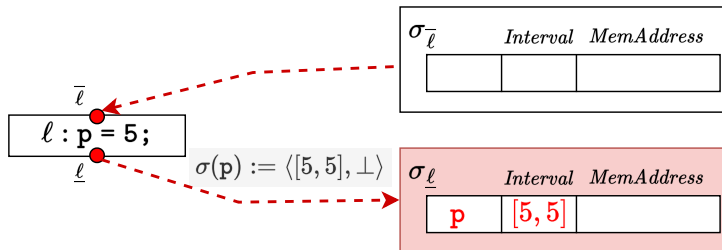
Algorithm 13: Abstract Execution Rule for ADDRSTMT

```

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 Interpretation on CONSTMT

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



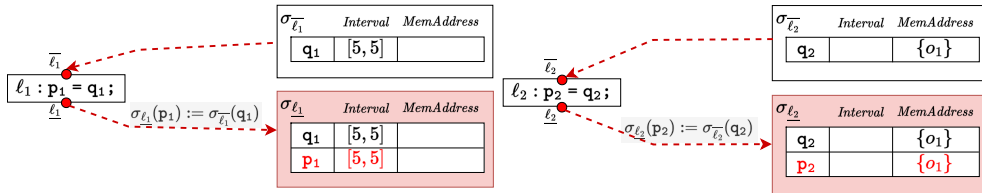
Algorithm 14: Abstract Execution Rule for CONSTMT

```

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 Interpretation on COPYSTMT

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



Algorithm 15: Abstract Execution Rule for COPYSTMT

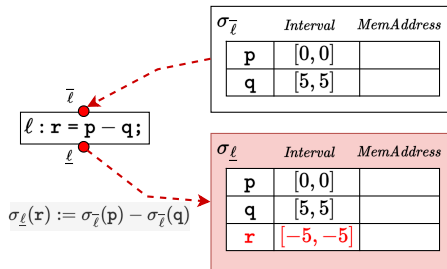
```

1 Function updateStateOnCopy(copy):
2   // Retrieve ICFGNode  $\ell$ ;
3   // Retrieve the abstract state at  $\underline{\ell}$ ;
4   // Assign RHS's abstract value to LHS;

```

Abstract Interpretation 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 16: Abstract Execution Rule for BINARYSTMT

```

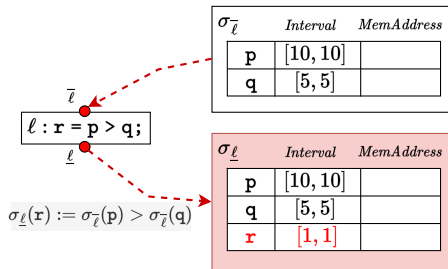
1 Function updateStateOnBinary(binary):
2   // Retrieve ICFGNode  $\ell$ ;
3   // Retrieve the abstract state at  $\underline{\ell}$ ;
4   // Assign the results after the binary
   operation of the two operands  $\mathbf{op0}$  and  $\mathbf{op1}$ ;

```

Operands $\mathbf{op0}$ and $\mathbf{op1}$ are assumed to be properly initialized (no uninitialized variables or randomization).

Abstract Interpretation on CMPSTMT

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



Algorithm 17: Abstract Execution Rule for CMPSTMT

```

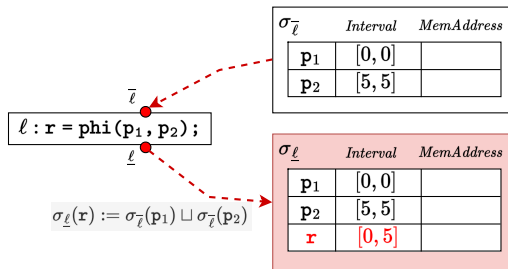
1 Function updateStateOnCmp(cmp):
2   // Retrieve ICFGNode  $\ell$ ;
3   // Retrieve the abstract state at  $\underline{\ell}$ ;
4   // Assign the results after the
   // comparison operation of the two operands;

```

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

Abstract Interpretation 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 18: Abstract Execution Rule for PHISTMT

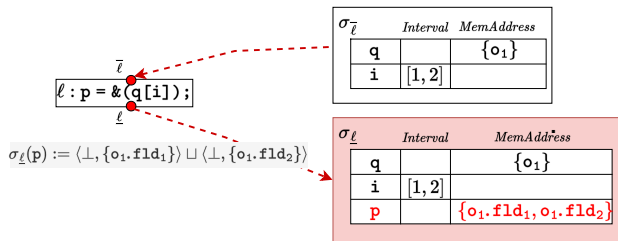
```

1 Function updateStateOnPhi(phi):
2   // Retrieve ICFGNode  $\ell$ ;
3   // Retrieve the abstract state at  $\bar{\ell}$ ;
4   // Join the abstract values of all n
   // operands retrieved from  $\bar{\ell}$  or from the
   // ICFGNode where each operand is defined.;
5   // Assign the joined values to the result
   // operand.;
6   //  $\sigma_{\underline{\ell}}(r) := \bigsqcup_{i=1}^n \sigma_{\bar{\ell}}(p_i)$ 

```


Abstract Interpretation on GEPSTMT

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



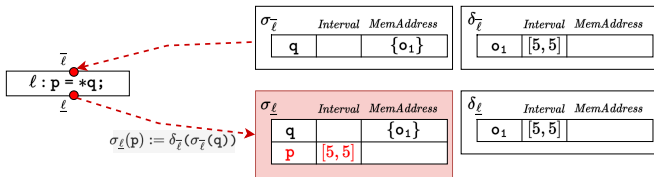
Algorithm 19: Abstract Execution Rule for GEPSTMT

```

1 Function updateStateOnGep(gep):
2   // Retrieve ICFGNode  $\ell$ ;
3   // Retrieve the abstract state as at  $\underline{\ell}$ ;
4   // Retrieve the field index or array index
   i given as as.getElementIndex(gep);
5   // Retrieve the memory address value via
   as.getGepObjAddr(rhs, i) and assign it to
   LHS
    
```

Abstract Interpretation 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 20: Abstract Execution Rule for LOADSTMT

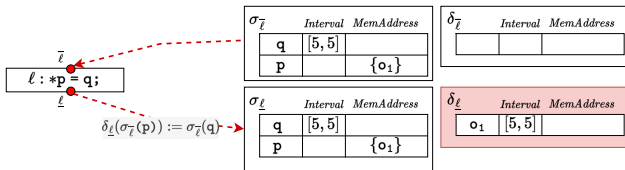
```

1 Function updateStateOnLoad(load):
2   // Retrieve ICFGNode  $\bar{\ell}$ ;
3   // Retrieve the abstract state as at  $\bar{\ell}$ ;
4   // Load the value from RHS via
   as.loadValue(rhs) and assign it to LHS;

```

Abstract Interpretation on STORESTMT

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



Algorithm 21: Abstract Execution Rule for STORESTMT

```

1 Function updateStateOnStore(store):
2   // Retrieve ICFGNode  $\ell$ ;
3   // Retrieve the abstract state as at  $\underline{\ell}$ ;
4   // Store RHS value to LHS via as.storeValue;

```

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

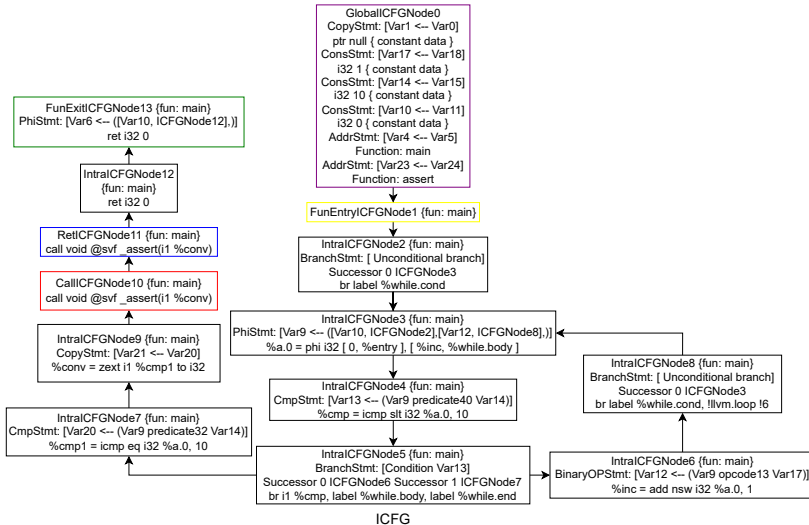
Compile to LLVM IR



```
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

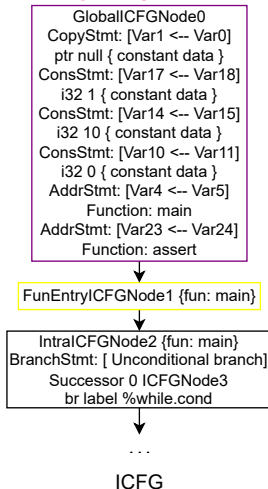
An Example: Abstract Trace σ for Top-level Variables



ICFG

An Example: Abstract Trace σ for Top-level Variables

Before Entering Loop



Algorithm 22: Abstract execution guided by WTO

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

$postAbsTrace[ICFGNode0].varToAbsVal$:

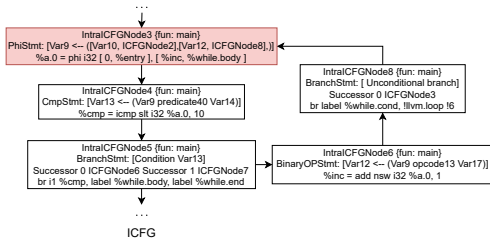
SVFVar	AbstractValue(<i>Interval</i> , <i>MemAddress</i>)
Var0	$\langle \perp, \{0x7f00\} \rangle$
Var1	$\langle \perp, \{0x7f00\} \rangle$
Var18	$\langle [1, 1], \perp \rangle$
Var17	$\langle [1, 1], \perp \rangle$
Var14	$\langle [10, 10], \perp \rangle$
Var15	$\langle [10, 10], \perp \rangle$
Var10	$\langle [0, 0], \perp \rangle$
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)



$postAbsTrace[ICFGNode3].varToAbsVal :$

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

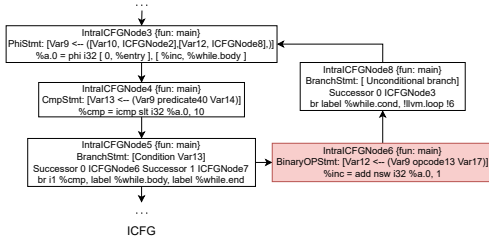
Algorithm 12: Handle ICFG Cycle

```

1 Function handleICFGCycle (cycle):
2    $\ell := cycle.getHead().getICFGNode();$  // cycle head ICFGNode  $\ell$ 
3   increasing := true;
4   i := 0; // analysis iteration for the loop
5   while true do
6      $as_{pre} := \sigma_{\ell};$  // abstract state in the last iteration
7     handleICFGNode( $\ell$ );
8      $as_{cur} := \sigma_{\ell};$  // abstract state in the current iteration
9     if  $i \geq Options.WidenDelay()$  then
10      if increasing then
11         $\sigma_{\ell} := as_{pre} \nabla as_{cur};$  // widening
12        if  $\sigma_{\ell} \equiv as_{pre}$  then
13          increasing := false;
14          continue;
15      else
16         $\sigma_{\ell} := as_{pre} \Delta as_{cur};$  // narrowing
17        if  $\sigma_{\ell} \equiv as_{pre}$  then
18          break;
19      // analyze remaining cycle components after two fixed-points
20      foreach comp  $\in cycle.getWTOComponents()$  do
21        if comp is Singleton then
22          handleICFGNode(comp.getICFGNode())
23        else if comp is Cycle then
24          handleICFGCycle(comp);
25      i++;
26  return;
  
```

An Example: Abstract Trace σ for Top-level Variables

Widen Delay Phase (cur_iter is 0)



$postAbsTrace[ICFGNode6].varToAbsVal$:

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

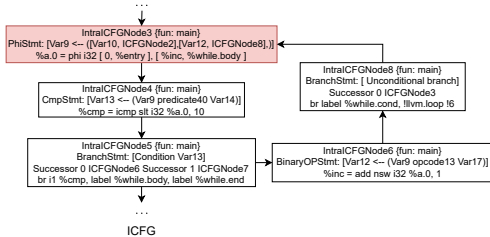
Algorithm 12: Handle ICFG Cycle

```

1 Function handleICFGCycle (cycle):
2    $\ell := cycle.getHead().getICFGNode();$  // cycle head ICFGNode  $\ell$ 
3   increasing := true;
4   i := 0; // analysis iteration for the loop
5   while true do
6      $as_{pre} := \sigma_{\ell};$  // abstract state in the last iteration
7     handleICFGNode( $\ell$ );
8      $as_{cur} := \sigma_{\ell};$  // abstract state in the current iteration
9     if  $i \geq Options.WidenDelay()$  then
10      if increasing then
11         $\sigma_{\ell} := as_{pre} \nabla as_{cur};$  // widening
12        if  $\sigma_{\ell} \equiv as_{pre}$  then
13          increasing := false;
14          continue;
15      else
16         $\sigma_{\ell} := as_{pre} \Delta as_{cur};$  // narrowing
17        if  $\sigma_{\ell} \equiv as_{pre}$  then
18          break;
19      // analyze remaining cycle components after two fixed-points
20      foreach comp  $\in cycle.getWTOComponents()$  do
21        if comp is Singleton then
22          handleICFGNode(comp.getICFGNode())
23        else if comp is Cycle then
24          handleICFGCycle(comp);
25      i++;
26   return;
  
```


An Example: Abstract Trace σ for Top-level Variables

Widen Delay Phase (cur_iter is 1)



$postAbsTrace[ICFGNode3].varToAbsVal :$

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

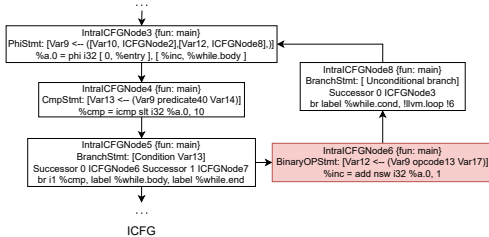
Algorithm 12: Handle ICFG Cycle

```

1 Function handleICFGCycle (cycle):
2    $\ell := cycle.getHead().getICFGNode();$  // cycle head ICFGNode  $\ell$ 
3   increasing := true;
4   i := 0; // analysis iteration for the loop
5   while true do
6      $as_{pre} := \sigma_{\ell};$  // abstract state in the last iteration
7     handleICFGNode( $\ell$ );
8      $as_{cur} := \sigma_{\ell};$  // abstract state in the current iteration
9     if  $i \geq Options.WidenDelay()$  then
10      if increasing then
11         $\sigma_{\ell} := as_{pre} \nabla as_{cur};$  // widening
12        if  $\sigma_{\ell} \equiv as_{pre}$  then
13          increasing := false;
14          continue;
15      else
16         $\sigma_{\ell} := as_{pre} \Delta as_{cur};$  // narrowing
17        if  $\sigma_{\ell} \equiv as_{pre}$  then
18          break;
19      // analyze remaining cycle components after two fixed-points
20      foreach comp  $\in cycle.getWTOComponents()$  do
21        if comp is Singleton then
22          handleICFGNode(comp.getICFGNode())
23        else if comp is Cycle then
24          handleICFGCycle(comp);
25      i++;
26  return;
  
```

An Example: Abstract Trace σ for Top-level Variables

Widen Delay Phase (cur_iter is 1)



$postAbsTrace[ICFGNode6].varToAbsVal :$

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

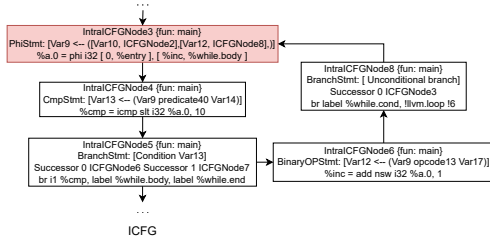
Algorithm 12: Handle ICFG Cycle

```

1 Function handleICFGCycle (cycle):
2    $\ell := cycle.getHead().getICFGNode();$  // cycle head ICFGNode  $\ell$ 
3   increasing := true;
4   i := 0; // analysis iteration for the loop
5   while true do
6      $as_{pre} := \sigma_{\ell};$  // abstract state in the last iteration
7     handleICFGNode( $\ell$ );
8      $as_{cur} := \sigma_{\ell};$  // abstract state in the current iteration
9     if  $i \geq Options.WidenDelay()$  then
10      if increasing then
11         $\sigma_{\ell} := as_{pre} \nabla as_{cur};$  // widening
12        if  $\sigma_{\ell} \equiv as_{pre}$  then
13          increasing := false;
14          continue;
15      else
16         $\sigma_{\ell} := as_{pre} \Delta as_{cur};$  // narrowing
17        if  $\sigma_{\ell} \equiv as_{pre}$  then
18          break;
19      // analyze remaining cycle components after two fixed-points
20      foreach comp  $\in cycle.getWTOComponents()$  do
21        if comp is Singleton then
22          handleICFGNode(comp.getICFGNode())
23        else if comp is Cycle then
24          handleICFGCycle(comp);
25      i++;
26  return;
  
```

An Example: Abstract Trace σ for Top-level Variables

Widen Phase (cur_iter is 2)



ICFG
 $postAbsTrace[ICFGNode3].varToAbsVal :$

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

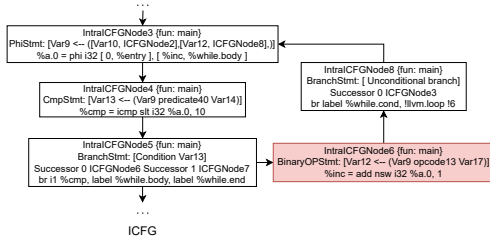
Algorithm 12: Handle ICFG Cycle

```

1 Function handleICFGCycle (cycle):
2    $\ell := cycle.getHead().getICFGNode();$  // cycle head ICFGNode  $\ell$ 
3   increasing := true;
4   i := 0; // analysis iteration for the loop
5   while true do
6      $as_{pre} := \sigma_{\ell};$  // abstract state in the last iteration
7     handleICFGNode( $\ell$ );
8      $as_{cur} := \sigma_{\ell};$  // abstract state in the current iteration
9     if  $i \geq Options.WidenDelay()$  then
10      if increasing then
11         $\sigma_{\ell} := as_{pre} \nabla as_{cur};$  // widening
12        if  $\sigma_{\ell} \equiv as_{pre}$  then
13          increasing := false;
14          continue;
15        else
16           $\sigma_{\ell} := as_{pre} \Delta as_{cur};$  // narrowing
17          if  $\sigma_{\ell} \equiv as_{pre}$  then
18            break;
19      // analyze remaining cycle components after two fixed-points
20      foreach comp  $\in cycle.getWTOComponents()$  do
21        if comp is Singleton then
22          handleICFGNode(comp.getICFGNode())
23        else if comp is Cycle then
24          handleICFGCycle(comp);
25      i++;
26   return;
  
```

An Example: Abstract Trace σ for Top-level Variables

Widen Phase (cur_iter is 2)



$postAbsTrace[ICFGNode6].varToAbsVal :$

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

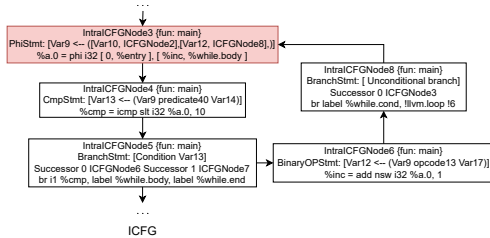
Algorithm 12: Handle ICFG Cycle

```

1 Function handleICFGCycle (cycle):
2    $\ell := cycle.getHead().getICFGNode();$  // cycle head ICFGNode  $\ell$ 
3   increasing := true;
4   i := 0; // analysis iteration for the loop
5   while true do
6      $as_{pre} := \sigma_{\ell};$  // abstract state in the last iteration
7     handleICFGNode( $\ell$ );
8      $as_{cur} := \sigma_{\ell};$  // abstract state in the current iteration
9     if  $i \geq Options.WidenDelay()$  then
10      if increasing then
11         $\sigma_{\ell} := as_{pre} \nabla as_{cur};$  // widening
12        if  $\sigma_{\ell} \equiv as_{pre}$  then
13          increasing := false;
14          continue;
15      else
16         $\sigma_{\ell} := as_{pre} \Delta as_{cur};$  // narrowing
17        if  $\sigma_{\ell} \equiv as_{pre}$  then
18          break;
19      // analyze remaining cycle components after two fixed-points
20      foreach comp  $\in cycle.getWTOComponents()$  do
21        if comp is Singleton then
22          handleICFGNode(comp.getICFGNode())
23        else if comp is Cycle then
24          handleICFGCycle(comp);
25      i++;
26  return;
  
```

An Example: Abstract Trace σ for Top-level Variables

Widen Phase Fixed Point



$postAbsTrace[ICFGNode3].varToAbsVal :$

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

Algorithm 12: Handle ICFG Cycle

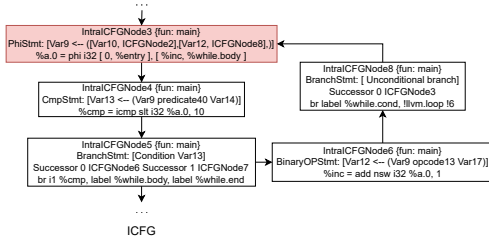
```

1 Function handleICFGCycle (cycle):
2    $\ell := cycle.getHead().getICFGNode();$  // cycle head ICFGNode  $\ell$ 
3    $increasing := true;$ 
4    $i := 0;$  // analysis iteration for the loop
5   while  $true$  do
6      $as_{pre} := \sigma_{\ell};$  // abstract state in the last iteration
7      $handleICFGNode(\ell);$ 
8      $as_{cur} := \sigma_{\ell};$  // abstract state in the current iteration
9     if  $i \geq Options.WidenDelay()$  then
10      if  $increasing$  then
11         $\sigma_{\ell} := as_{pre} \nabla as_{cur};$  // widening
12        if  $\sigma_{\ell} \equiv as_{pre}$  then
13           $increasing := false;$ 
14          continue;
15      else
16         $\sigma_{\ell} := as_{pre} \Delta as_{cur};$  // narrowing
17        if  $\sigma_{\ell} \equiv as_{pre}$  then
18          break;
19      // analyze remaining cycle components after two fixed-points
20      foreach  $comp \in cycle.getWTOComponents()$  do
21        if  $comp$  is Singleton then
22           $handleICFGNode(comp.getICFGNode());$ 
23        else if  $comp$  is Cycle then
24           $handleICFGCycle(comp);$ 
25       $i++;$ 
26   return;

```

An Example: Abstract Trace σ for Top-level Variables

Narrow Phase



$postAbsTrace[ICFGNode3].varToAbsVal :$

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

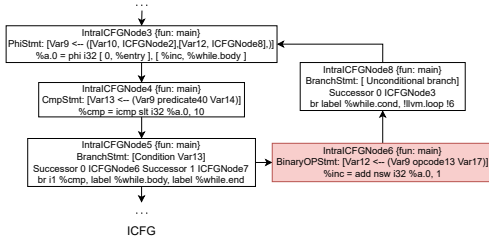
Algorithm 12: Handle ICFG Cycle

```

1 Function handleICFGCycle (cycle):
2    $\ell := cycle.getHead().getICFGNode();$  // cycle head ICFGNode  $\ell$ 
3   increasing := true;
4   i := 0; // analysis iteration for the loop
5   while true do
6      $as_{pre} := \sigma_{\ell};$  // abstract state in the last iteration
7     handleICFGNode( $\ell$ );
8      $as_{cur} := \sigma_{\ell};$  // abstract state in the current iteration
9     if  $i \geq Options.WidenDelay()$  then
10      if increasing then
11         $\sigma_{\ell} := as_{pre} \nabla as_{cur};$  // widening
12        if  $\sigma_{\ell} \equiv as_{pre}$  then
13          increasing := false;
14          continue;
15      else
16         $\sigma_{\ell} := as_{pre} \Delta as_{cur};$  // narrowing
17        if  $\sigma_{\ell} \equiv as_{pre}$  then
18          break;
19    // analyze remaining cycle components after two fixed-points
20    foreach comp  $\in cycle.getWTOComponents()$  do
21      if comp is Singleton then
22        handleICFGNode(comp.getICFGNode())
23      else if comp is Cycle then
24        handleICFGCycle(comp);
25    i++;
26  return;
  
```

An Example: Abstract Trace σ for Top-level Variables

Narrow Phase



$postAbsTrace[ICFGNode6].varToAbsVal :$

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

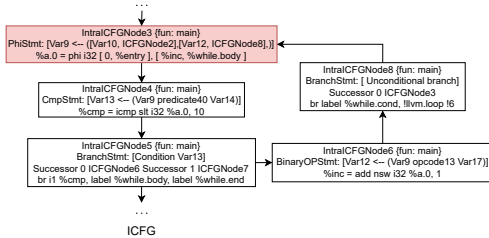
Algorithm 12: Handle ICFG Cycle

```

1 Function handleICFGCycle (cycle):
2    $\ell := cycle.getHead().getICFGNode();$  // cycle head ICFGNode  $\ell$ 
3   increasing := true;
4   i := 0; // analysis iteration for the loop
5   while true do
6      $as_{pre} := \sigma_{\ell};$  // abstract state in the last iteration
7     handleICFGNode( $\ell$ );
8      $as_{cur} := \sigma_{\ell};$  // abstract state in the current iteration
9     if  $i \geq Options.WidenDelay()$  then
10      if increasing then
11         $\sigma_{\ell} := as_{pre} \nabla as_{cur};$  // widening
12        if  $\sigma_{\ell} \equiv as_{pre}$  then
13          increasing := false;
14          continue;
15      else
16         $\sigma_{\ell} := as_{pre} \Delta as_{cur};$  // narrowing
17        if  $\sigma_{\ell} \equiv as_{pre}$  then
18          break;
19      // analyze remaining cycle components after two fixed-points
20      foreach comp  $\in cycle.getWTOComponents()$  do
21        if comp is Singleton then
22          handleICFGNode(comp.getICFGNode())
23        else if comp is Cycle then
24          handleICFGCycle(comp);
25      i++;
26  return;
  
```

An Example: Abstract Trace σ for Top-level Variables

Narrow Phase Fixed Point



$postAbsTrace[ICFGNode3].varToAbsVal :$

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

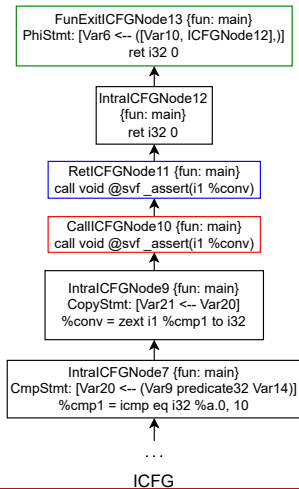
Algorithm 12: Handle ICFG Cycle

```

1 Function handleICFGCycle (cycle):
2    $\ell := cycle.getHead().getICFGNode();$  // cycle head ICFGNode  $\ell$ 
3   increasing := true;
4   i := 0; // analysis iteration for the loop
5   while true do
6      $as_{pre} := \sigma_{\ell};$  // abstract state in the last iteration
7     handleICFGNode( $\ell$ );
8      $as_{cur} := \sigma_{\ell};$  // abstract state in the current iteration
9     if  $i \geq Options.WidenDelay()$  then
10       if increasing then
11          $\sigma_{\ell} := as_{pre} \nabla as_{cur};$  // widening
12         if  $\sigma_{\ell} \equiv as_{pre}$  then
13           increasing := false;
14           continue;
15       else
16          $\sigma_{\ell} := as_{pre} \Delta as_{cur};$  // narrowing
17         if  $\sigma_{\ell} \equiv as_{pre}$  then
18           break;
19       // analyze remaining cycle components after two fixed-points
20       foreach comp  $\in cycle.getWTOComponents()$  do
21         if comp is Singleton then
22           handleICFGNode(comp.getICFGNode())
23         else if comp is Cycle then
24           handleICFGCycle(comp);
25   i++;
26   return;
  
```


An Example: Abstract Trace σ for Top-level Variables

After Exiting Loop



Algorithm 13: Abstract execution guided by WTO

```

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

$postAbsTrace[ICFGNode7].varToAbsVal$:

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