# Abstract Interpretation for Code Analysis and Verification

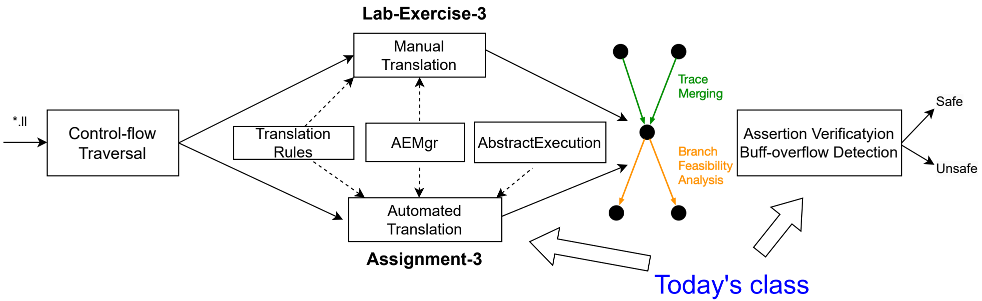
## (Week 9)

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

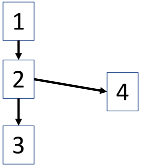
? How to analyze a program **free of loop**?

✓ 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 $a \to b$, 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.

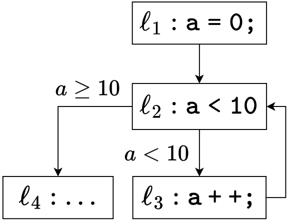
# Weak Topological Order

## 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

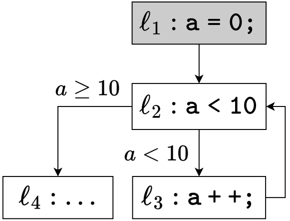
# Weak Topological Order

## Analysis Order of Nodes on Control-Flow Graph

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

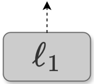
✔ We can analyze a program containing loops adhering to the **weak topological order** (WTO) on its control flow graph.

**What is the weak topological order?**




Control Flow Graph

$1^{st}$ WTO component: a sigle node



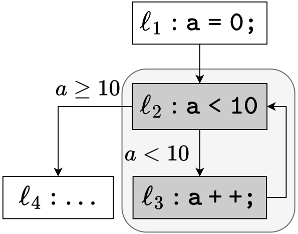
# Weak Topological Order

## Analysis Order of Nodes on Control-Flow Graph

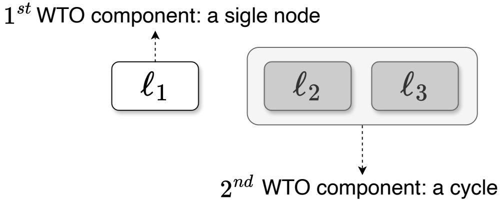
❓ How to analyze a program **containing loops**?

✅ 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



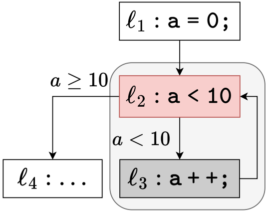
# Weak Topological Order

## Analysis Order of Nodes on Control-Flow Graph

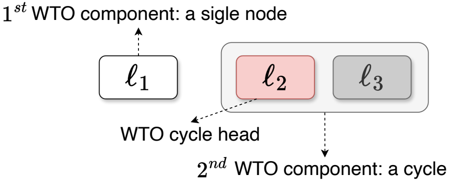
? How to analyze a program **containing loops**?

✔ 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?**



Control Flow Graph



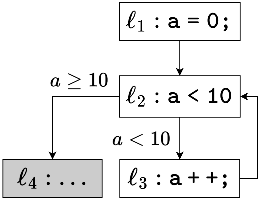
# Weak Topological Order

## Analysis Order of Nodes on Control-Flow Graph

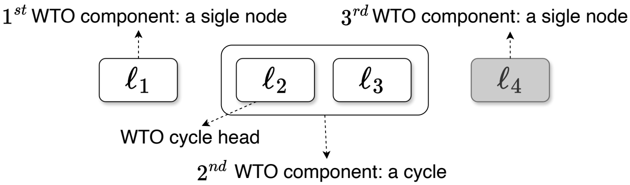
? How to analyze a program **containing loops**?

✔ 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?**




Control Flow Graph



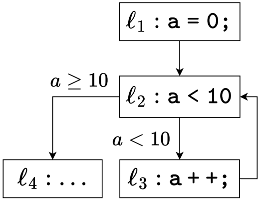
# Weak Topological Order

## Analysis Order of Nodes on Control-Flow Graph

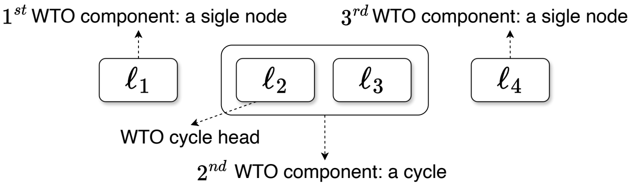
❓ How to analyze a program **containing loops**?

✔ 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?**




Control Flow Graph




**Analyze each node following the WTO**

Analyze $\ell_1$ ⇨ Repeat: analyze $\ell_2$ and $\ell_3$ ⇨ Analyze $\ell_4$

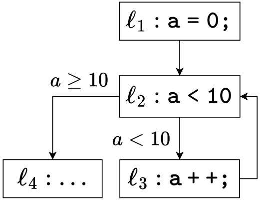
# Weak Topological Order

## Analysis Order of Nodes on Control-Flow Graph

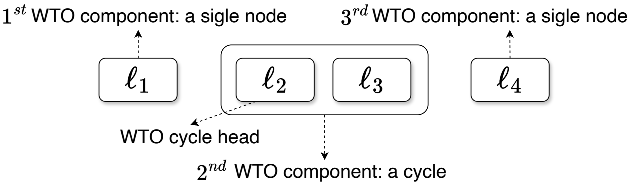
❓ How to analyze a program **containing loops**?

✔ 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




**Analyze each node following the WTO**

Analyze $\ell_1$ ⇨ Repeat: analyze $\ell_2$ and $\ell_3$ ⇨ Analyze $\ell_4$

**Perform widening when analyzing the cycle heads (i.e., $\ell_2$ )**

# 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 $\mathtt{x = y + z}$, and $\mathtt{y} \in [1, 5]$ and $\mathtt{z} \in [2, 3]$, then $\mathtt{x} \in [3, 8]$. 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 $\ell$ (`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** $\sigma \in \mathbb{L} \times \mathcal{V} \to \mathbb{A}$ represents a list of abstract states before ($\overline{\ell}$) and after ($\underline{\ell}$) each program statement $\ell$ (`preAbsTrace` and `postAbsTrace`).
- An **abstract state** (`AbstractState` in Lab-3 and Assignment-3) is defined as a map $AS : \mathcal{V} \to \mathbb{A}$ 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.

| | Notation | Domain | SSE Data Structure |
|---|---|---|---|
| Abstract **trace** | $\mathbb{L} \times \mathcal{V} \to \mathbb{A}$ | $\sigma$ | `preAbsTrace`: **trace** before `ICFGNodes`<br>`postAbsTrace`: **trace** after `ICFGNodes` |
| Abstract **state** at $L \in \mathbb{L}$ | $\mathcal{V} \to \mathbb{A}$ | $\sigma_{\overline{\ell}}$<br>$\sigma_{\underline{\ell}}$ | `preAbsTrace[node]`: **state** before `node` $\ell$<br>`postAbsTrace[node]`: **state** after `node` $\ell$ |
| Abstract **value** of varId at $L \in \mathbb{L}$ | $\mathbb{A}$ | $\sigma_{\underline{\ell}}$(varId) | `as` $=$ `postAbsTrace[node]`<br>`as[VarID]`: **value** of varId after `node` $\ell$ |

# Overall Algorithm of Abstract Interpretation in Assignment-3

**Algorithm 1:** Analyse from main function

```
Function analyse() // driver function to start the analysis:
    initWTO();
    handleGlobalNode();
    handleFunction(mainFun);
```

**Algorithm 2:** Handle Function

```
Function handleFunction(fun):
    worklist := [funEntryICFGNode] while worklist ≠ ∅ do
        n := worklist.pop_front() ;
        if n is a cycle head then
            cycle := cycle_head_to_cycle[n] ;
            handleICFGCycle(cycle); ;                         // Assignment-3
            foreach n′ ∈ getNextNodesOfCycle(cycle) do
                worklist.push_back(n′) ;
        else
            if handleICFGNode(n) == false then
                foreach n′ ∈ getNextNodes(n) do
                    worklist.push_back(n′) ;
```

**Algorithm 3:** Handle ICFG Node

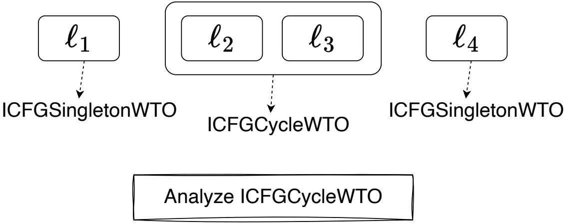
```
Function handleICFGNode(n):
    feasible, as_pre := mergeStatesFromPredecessors(node);
    if !feasible then
        return false;
    as_last := σ_n;
    σ_n := as_pre;
    foreach stmt ∈ n→getSVFStmts() do
        updateAbsState(stmt),; ;                    // Assignment-3
        bufOverflowDetection(stmt); ;               // Assignment-3

    if n is CallICFGNode then
        // Handle stub function and external call;
        // Skip recursive call, not handling recursive functions in
        Assignment-3;
        // Handle normal call;
    if σ_n ≡ as_last then
        return false; // state not changed
    return true; // state changed
```

# Handling Loop in Assignment-3

**Algorithm 4:** Handle ICFG Cycle

```
Function handleICFGCycle (cycle):
    ℓ := cycle.getHead().getICFGNode();    // cycle head ICFGNode ℓ
    increasing := true;
    i := 0;    // analysis iteration for the loop
    while true do
        as_pre := σ_ℓ;    // abstract state in the last iteration
        handleICFGNode(ℓ);
        as_cur := σ_ℓ;    // abstract state in the current iteration
        if i ≥ Options.WidenDelay() then
            if increasing then
                σ_ℓ := as_pre ∇ as_cur;    // widening
                if σ_ℓ ≡ as_pre then
                    increasing := false;
                    continue;
            else
                σ_ℓ := as_pre Δ as_cur;    // narrowing
                if σ_ℓ ≡ as_pre then
                    break;
        // analyze remaining cycle components after two fixed-points
        foreach comp ∈ cycle.getWTOComponents() do
            if comp is Singleton then
                handleICFGNode(comp.getICFGNode())
            else if comp is Cycle then
                handleICFGCycle(comp);
        i++;
    return;
```

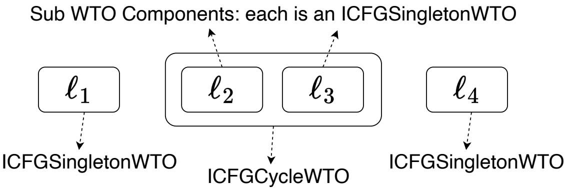
# Widening and Narrowing




**Algorithm 12:** Handle ICFG Cycle

```
Function handleICFGCycle (cycle):
    ℓ := cycle.getHead().getICFGNode();   // cycle head ICFGNode ℓ
    increasing := true;
    i := 0;   // analysis iteration for the loop
    while true do
        as_pre := σ_ℓ;   // abstract state in the last iteration
        handleICFGNode(ℓ);
        as_cur := σ_ℓ;   // abstract state in the current iteration
        if i ≥ Options.WidenDelay() then
            if increasing then
                σ_ℓ := as_pre ∇ as_cur;   // widening
                if σ_ℓ ≡ as_pre then
                    increasing := false;
                    continue;
            else
                σ_ℓ := as_pre Δ as_cur;   // narrowing
                if σ_ℓ ≡ as_pre then
                    break;
        // analyze remaining cycle components after two fixed-points
        foreach comp ∈ cycle.getWTOComponents() do
            if comp is Singleton then
                handleICFGNode(comp.getICFGNode())
            else if comp is Cycle then
                handleICFGCycle(comp);
        i++;
    return;
```

# Widening and Narrowing

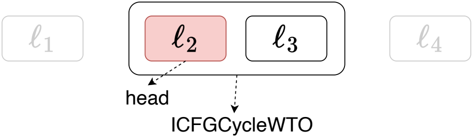



Analyze ICFGCycleWTO

**Algorithm 12:** Handle ICFG Cycle

```
Function handleICFGCycle (cycle):
    ℓ := cycle.getHead().getICFGNode();    // cycle head ICFGNode ℓ
    increasing := true;
    i := 0;    // analysis iteration for the loop
    while true do
        as_pre := σ_ℓ;    // abstract state in the last iteration
        handleICFGNode(ℓ);
        as_cur := σ_ℓ;    // abstract state in the current iteration
        if i ≥ Options.WidenDelay() then
            if increasing then
                σ_ℓ := as_pre ∇ as_cur;    // widening
                if σ_ℓ ≡ as_pre then
                    increasing := false;
                    continue;
            else
                σ_ℓ := as_pre Δ as_cur;    // narrowing
                if σ_ℓ ≡ as_pre then
                    break;
        // analyze remaining cycle components after two fixed-points
        foreach comp ∈ cycle.getWTOComponents() do
            if comp is Singleton then
                handleICFGNode(comp.getICFGNode())
            else if comp is Cycle then
                handleICFGCycle(comp);
        i++;
    return;
```

# Weak Topological Order

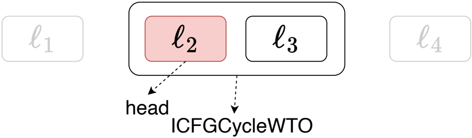



Get ICFGCycleWTO head $\ell_2$

**Algorithm 12:** Handle ICFG Cycle

```
Function handleICFGCycle (cycle):
    ℓ := cycle.getHead().getICFGNode();   // cycle head ICFGNode ℓ
    increasing := true;
    i := 0;   // analysis iteration for the loop
    while true do
        as_pre := σ_ℓ;   // abstract state in the last iteration
        handleICFGNode(ℓ);
        as_cur := σ_ℓ;   // abstract state in the current iteration
        if i ≥ Options.WidenDelay() then
            if increasing then
                σ_ℓ := as_pre ∇ as_cur;   // widening
                if σ_ℓ ≡ as_pre then
                    increasing := false;
                    continue;
            else
                σ_ℓ := as_pre Δ as_cur;   // narrowing
                if σ_ℓ ≡ as_pre then
                    break;
        // analyze remaining cycle components after two fixed-points
        foreach comp ∈ cycle.getWTOComponents() do
            if comp is Singleton then
                handleICFGNode(comp.getICFGNode())
            else if comp is Cycle then
                handleICFGCycle(comp);
        i++;
    return;
```

# Weak Topological Order

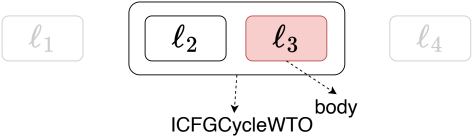



Handle ICFGCycleWTO head $\ell_2$

**Algorithm 12:** Handle ICFG Cycle

```
Function handleICFGCycle (cycle):
    ℓ := cycle.getHead().getICFGNode();   // cycle head ICFGNode ℓ
    increasing := true;
    i := 0;   // analysis iteration for the loop
    while true do
        as_pre := σ_ℓ;   // abstract state in the last iteration
        handleICFGNode(ℓ);
        as_cur := σ_ℓ;   // abstract state in the current iteration
        if i ≥ Options.WidenDelay() then
            if increasing then
                σ_ℓ := as_pre ∇ as_cur;   // widening
                if σ_ℓ ≡ as_pre then
                    increasing := false;
                    continue;
            else
                σ_ℓ := as_pre Δ as_cur;   // narrowing
                if σ_ℓ ≡ as_pre then
                    break;
        // analyze remaining cycle components after two fixed-points
        foreach comp ∈ cycle.getWTOComponents() do
            if comp is Singleton then
                handleICFGNode(comp.getICFGNode())
            else if comp is Cycle then
                handleICFGCycle(comp);
        i++;
    return;
```

# Widening and Narrowing



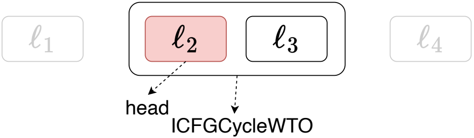

Handle ICFGCycleWTO body $\ell_3$

**Algorithm 12:** Handle ICFG Cycle

```
Function handleICFGCycle (cycle):
    ℓ := cycle.getHead().getICFGNode();   // cycle head ICFGNode ℓ
    increasing := true;
    i := 0;   // analysis iteration for the loop
    while true do
        as_pre := σ_ℓ;   // abstract state in the last iteration
        handleICFGNode(h);
        as_cur := σ_ℓ;   // abstract state in the current iteration
        if i ≥ Options.WidenDelay() then
            if increasing then
                σ_ℓ := as_pre ∇ as_cur;   // widening
                if σ_ℓ ≡ as_pre then
                    increasing := false;
                    continue;
            else
                σ_ℓ := as_pre Δ as_cur;   // narrowing
                if σ_ℓ ≡ as_pre then
                    break;
        // analyze remaining cycle components after two fixed-points
        foreach comp ∈ cycle.getWTOComponents() do
            if comp is Singleton then
                handleICFGNode(comp.getICFGNode())
            else if comp is Cycle then
                handleICFGCycle(comp);
        i++;
    return;
```

**Note**: `getIWTOcomponents` returns Cycle WTO body, i.e., $\ell_3$

# Widening and Narrowing



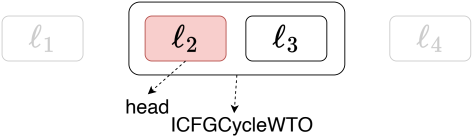

When $cur\_iter \geq Options :: WidenDelay()$

perform widening on $\ell_2$

**Algorithm 12:** Handle ICFG Cycle

```
Function handleICFGCycle (cycle):
    ℓ := cycle.getHead().getICFGNode();   // cycle head ICFGNode ℓ
    increasing := true;
    i := 0;   // analysis iteration for the loop
    while true do
        as_pre := σ_ℓ;   // abstract state in the last iteration
        handleICFGNode(ℓ);
        as_cur := σ_ℓ;   // abstract state in the current iteration
        if i ≥ Options.WidenDelay() then
            if increasing then
                σ_ℓ := as_pre ∇ as_cur;   // widening
                if σ_ℓ ≡ as_pre then
                    increasing := false;
                    continue;
            else
                σ_ℓ := as_pre Δ as_cur;   // narrowing
                if σ_ℓ ≡ as_pre then
                    break;
        // analyze remaining cycle components after two fixed-points
        foreach comp ∈ cycle.getWTOComponents() do
            if comp is Singleton then
                handleICFGNode(comp.getICFGNode())
            else if comp is Cycle then
                handleICFGCycle(comp);
        i++;
    return;
```

# Widening and Narrowing

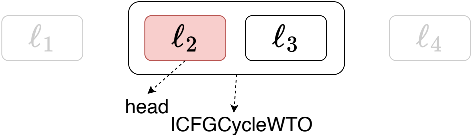



Widening reaches a fixpoint

**Algorithm 12:** Handle ICFG Cycle

```
Function handleICFGCycle (cycle):
    ℓ := cycle.getHead().getICFGNode();    // cycle head ICFGNode ℓ
    increasing := true;
    i := 0;    // analysis iteration for the loop
    while true do
        as_pre := σ_ℓ;    // abstract state in the last iteration
        handleICFGNode(ℓ);
        as_cur := σ_ℓ;    // abstract state in the current iteration
        if i ≥ Options.WidenDelay() then
            if increasing then
                σ_ℓ := as_pre ∇ as_cur;    // widening
                if σ_ℓ ≡ as_pre then
                    increasing := false;
                    continue;
            else
                σ_ℓ := as_pre Δ as_cur;    // narrowing
                if σ_ℓ ≡ as_pre then
                    break;
        // analyze remaining cycle components after two fixed-points
        foreach comp ∈ cycle.getWTOComponents() do
            if comp is Singleton then
                handleICFGNode(comp.getICFGNode())
            else if comp is Cycle then
                handleICFGCycle(comp);
        i++;
    return;
```

# Widening and Narrowing

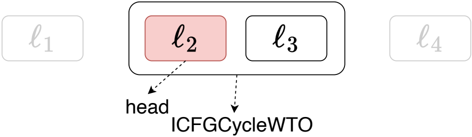



Perform narrowing on $\ell_2$

**Algorithm 12:** Handle ICFG Cycle

```
Function handleICFGCycle (cycle):
    ℓ := cycle.getHead().getICFGNode();    // cycle head ICFGNode ℓ
    increasing := true;
    i := 0;    // analysis iteration for the loop
    while true do
        as_pre := σ_ℓ;    // abstract state in the last iteration
        handleICFGNode(ℓ);
        as_cur := σ_ℓ;    // abstract state in the current iteration
        if i ≥ Options.WidenDelay() then
            if increasing then
                σ_ℓ := as_pre ∇ as_cur;    // widening
                if σ_ℓ ≡ as_pre then
                    increasing := false;
                    continue;
            else
                σ_ℓ := as_pre Δ as_cur;    // narrowing
                if σ_ℓ ≡ as_pre then
                    break;
        // analyze remaining cycle components after two fixed-points
        foreach comp ∈ cycle.getWTOComponents() do
            if comp is Singleton then
                handleICFGNode(comp.getICFGNode())
            else if comp is Cycle then
                handleICFGCycle(comp);
        i++;
    return;
```

# Widening and Narrowing




Narrowing reaches a fixpoint, Finish!

**Algorithm 12:** Handle ICFG Cycle

```
Function handleICFGCycle (cycle):
    ℓ := cycle.getHead().getICFGNode();    // cycle head ICFGNode ℓ
    increasing := true;
    i := 0;    // analysis iteration for the loop
    while true do
        as_pre := σ_ℓ;    // abstract state in the last iteration
        handleICFGNode(ℓ);
        as_cur := σ_ℓ;    // abstract state in the current iteration
        if i ≥ Options.WidenDelay() then
            if increasing then
                σ_ℓ := as_pre ∇ as_cur;    // widening
                if σ_ℓ ≡ as_pre then
                    increasing := false;
                    continue;
            else
                σ_ℓ := as_pre Δ as_cur;    // narrowing
                if σ_ℓ ≡ as_pre then
                    break;
        // analyze remaining cycle components after two fixed-points
        foreach comp ∈ cycle.getWTOComponents() do
            if comp is Singleton then
                handleICFGNode(comp.getICFGNode())
            else if comp is Cycle then
                handleICFGCycle(comp);
        i++;
    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 $\texttt{p} = \texttt{alloc}_{\texttt{o}}$) 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 | $\ell : \texttt{p} = \texttt{c}$ |
| COPYSTMT | $\ell : \texttt{p} = \texttt{q}$ |
| BINARYSTMT | $\ell : \texttt{r} = \texttt{p} \otimes \texttt{q}$ |
| PHISTMT | $\ell : \texttt{r} = \texttt{phi}(\texttt{p}_1, \texttt{p}_2, \ldots, \texttt{p}_n)$ |
| SEQUENCE | $\ell_1 ; \ell_2$ |
| BRANCHSTMT | $\ell_1$ : if($x < c$) then $\ell_2$ else $\ell_3$ |

# Abstract Interpretation on Pointer-Free SVFIR

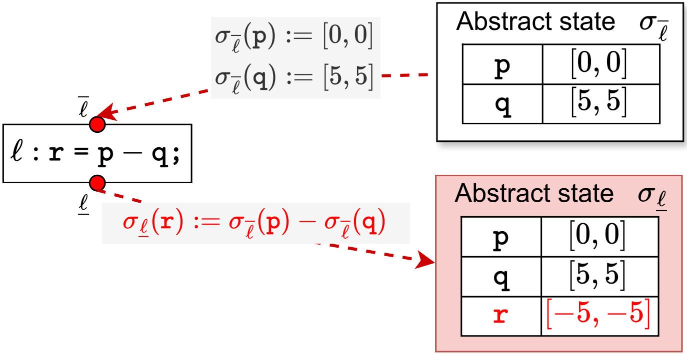
## Interval Domain

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

| SVFSTMT | C-Like form | Abstract Execution Rule |
|---|---|---|
| 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 $\ell_2$ else $\ell_3$ | $\sigma_{\overline{\ell_2}}(x) := \sigma_{\underline{\ell_1}}(x) \sqcap [-\infty, c-1]$, if $\sigma_{\underline{\ell_1}}(x) \sqcap [-\infty, c-1] \neq \perp$<br>$\sigma_{\overline{\ell_3}}(x) := \sigma_{\underline{\ell_1}}(x) \sqcap [c, +\infty]$, if $\sigma_{\underline{\ell_1}}(x) \sqcap [c, +\infty] \neq \perp$ |

# Abstract Interpretation on BinaryStmt

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

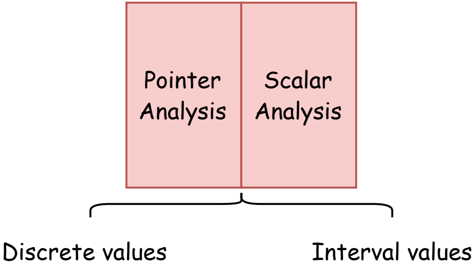


# Abstract Interpretation in the Presence of Pointers

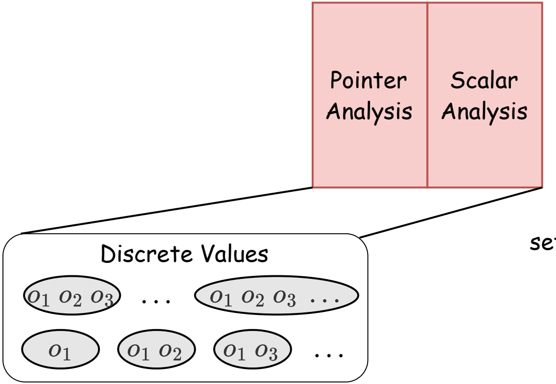
- SVFIR in the presence of pointers contain pointer-related statements including AddrStmt, GepStmt, LoadStmt and StoreStmt.
- Abstract interpretation needs to be performed on **a combined domain of intervals and addresses.**

| **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 | $\ell : \mathtt{r} = \mathtt{phi}(\mathtt{p}_1, \mathtt{p}_2, \ldots, \mathtt{p}_n)$ |
| Sequence | $\ell_1 ; \ell_2$ |
| BranchStmt | $\ell_1$ : if$(x < c)$ then $\ell_2$ else $\ell_3$ |
| AddrStmt | $\ell : \mathtt{p} = \mathtt{alloc}$ |
| GepStmt | $\ell : \mathtt{p} = \&(\mathtt{q} \rightarrow \mathtt{i})$ or $\mathtt{p} = \&\mathtt{q}[\mathtt{i}]$ |
| LoadStmt | $\ell : \mathtt{p} = *\mathtt{q}$ |
| StoreStmt | $\ell : *\mathtt{p} = \mathtt{q}$ |

# Combined Analysis



# Combined Analysis Using Discrete Values



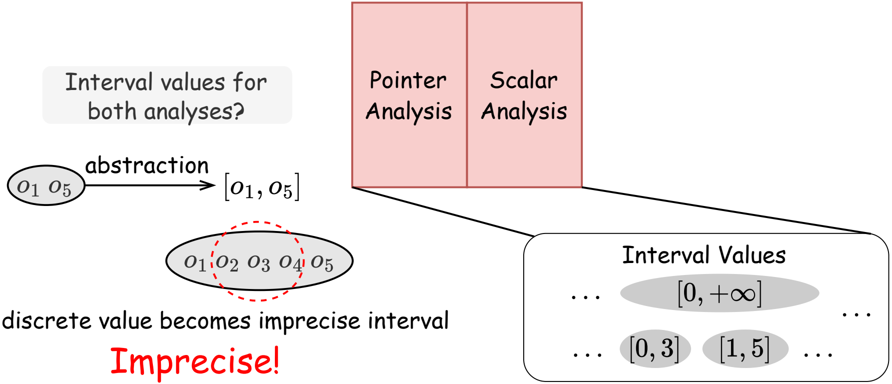

Discrete values for both analyses?

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

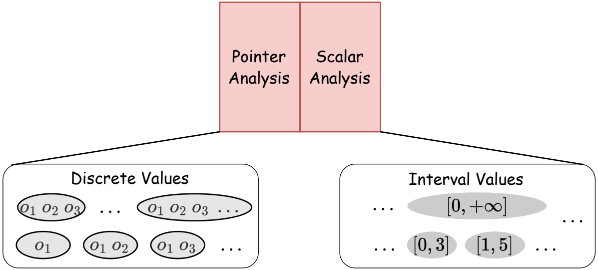
set of all possible values with inputs

Unscalable!

# Combined Analysis Using Interval Values



# Abstract Interpretation Over a Combined Domain

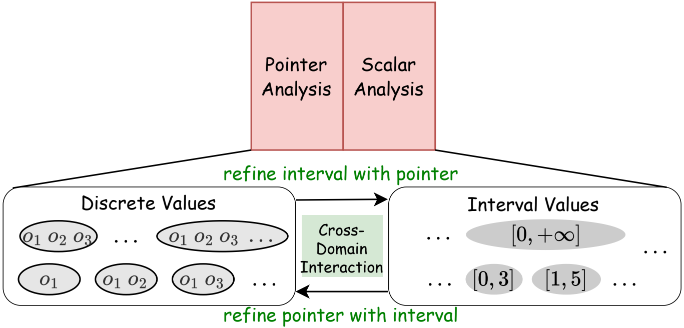



p = malloc(m*sizeof(int)); // p points to an array of size m
q = malloc(n*sizeof(int)); // q points to an array of size n

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

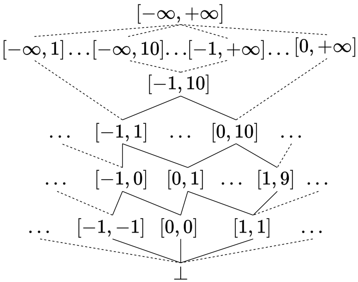



We require **a combination of interval and memory address domains** to precisely and efficiently perform abstract execution on SVFIR in the presence of pointers.

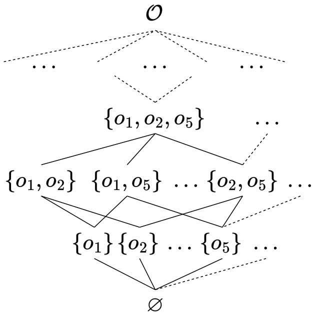
*Precise Sparse Abstract Execution via Cross-Domain Interaction, ICSE 2024*

# 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` | $o \in (\mathbb{S} \cup \mathbb{G} \cup \mathbb{H})$ | 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 $\sigma \in \mathbb{L} \times \mathbb{P} \rightarrow \mathit{Interval} \times \mathit{MemAddress}$ to track the memory addresses or interval values of these variables.
- For memory objects $\mathbb{O}$, we use $\delta \in \mathbb{L} \times \mathbb{O} \rightarrow \mathit{Interval} \times \mathit{MemAddress}$ to track their abstract values

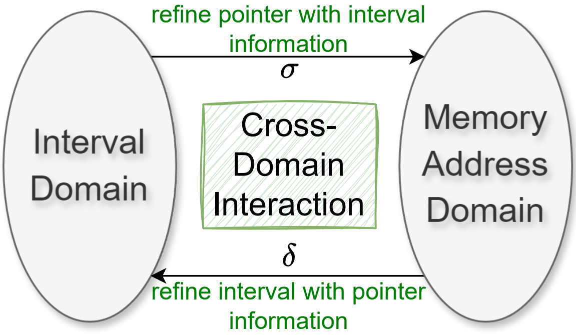
| | Notation | Domain | Data Structure Implementation |
|---|---|---|---|
| Abstract trace | $\sigma$ | $\mathbb{L} \times \mathbb{P} \rightarrow \mathit{Interval} \times \mathit{MemAddress}$ | *preAbsTrace*, *postAbsTrace* |
| | $\delta$ | $\mathbb{L} \times \mathbb{O} \rightarrow \mathit{Interval}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



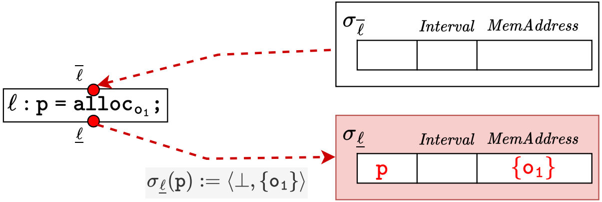
# Abstract Execution Rules on SVFIR in the Presence of Pointers

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

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

# Abstract Interpretation on ADDRSTMT

| SVFSTMT | C-Like form | Abstract Execution Rule |
|---|---|---|
| ADDRSTMT | $\ell : \texttt{p} = \texttt{alloc}_{\texttt{o}_\texttt{i}}$ | $\sigma_{\underline{\ell}}(\texttt{p}) := \langle \bot, \{o_i\} \rangle$ |



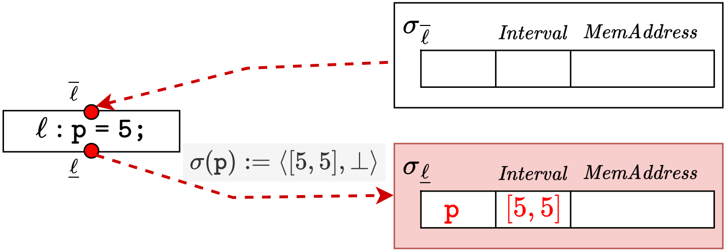

**Algorithm 13:** Abstract Execution Rule for ADDRSTMT

**Function** *updateStateOnAddr(addr)*:

```
node = addr→getICFGNode();
as = getAbsStateFromTrace(node);
initObjVar(as, SVFUtil :: cast⟨ObjVar⟩(addr→getRHSVar()));
as[addr→getLHSVarID()] = as[addr→getRHSVarID()];
```

# Abstract Interpretation on ConsStmt

| SVFStmt | C-Like form | Abstract Execution Rule |
|---|---|---|
| ConsStmt | $\ell : \mathtt{p} = \mathtt{c}$ | $\sigma_{\underline{\ell}}(\mathtt{p}) := \langle [\mathtt{c}, \mathtt{c}], \bot \rangle$ |



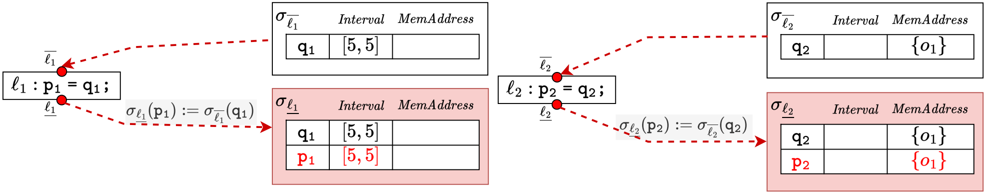

**Algorithm 14:** Abstract Execution Rule for ConsStmt

**Function** *updateStateOnAddr(addr)*:

```
node = addr→getICFGNode();
as = getAbsStateFromTrace(node);
initObjVar(as, SVFUtil :: cast⟨ObjVar⟩(addr→getRHSVar()));
as[addr→getLHSVarID()] = as[addr→getRHSVarID()];
```

# Abstract Interpretation on CopyStmt

| SVFStmt | C-Like form | Abstract Execution Rule |
|---|---|---|
| CopyStmt | $\ell : \mathtt{p} = \mathtt{q}$ | $\sigma_{\underline{\ell}}(\mathtt{p}) := \sigma_{\overline{\ell}}(\mathtt{q})$ |

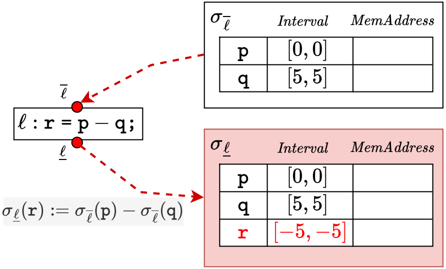



**Algorithm 15:** Abstract Execution Rule for CopyStmt

```
Function updateStateOnCopy(copy):
    // Retrieve ICFGNode ℓ;
    // Retrieve the abstract state at ℓ;
    // Assign RHS's abstract value to LHS;
```

# Abstract Interpretation on BINARYSTMT

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




**Algorithm 16:** Abstract Execution Rule for BINARYSTMT

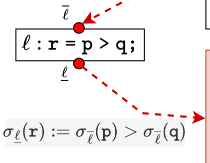
**Function** *updateStateOnBinary(binary)*:

```
// Retrieve ICFGNode ℓ;
// Retrieve the abstract state at ℓ;
// Assign the results after the binary
operation of the two operands op0 and op1;
```

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

# Abstract Interpretation on CmpStmt

| SVFStmt | C-Like form | Abstract Execution Rule |
|---|---|---|
| CmpStmt | $\ell : \mathtt{r} = \mathtt{p} \odot \mathtt{q}$ | $\sigma_{\underline{\ell}}(r) := \sigma_{\overline{\ell}}(p) \hat{\otimes} \sigma_{\overline{\ell}}(q)$ |




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

| | *Interval* | *MemAddress* |
|---|---|---|
| p | $[10, 10]$ | |
| q | $[5, 5]$ | |

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

| | *Interval* | *MemAddress* |
|---|---|---|
| p | $[10, 10]$ | |
| q | $[5, 5]$ | |
| r | $[1, 1]$ | |

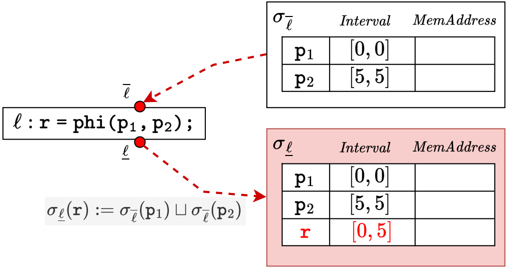
**Algorithm 17:** Abstract Execution Rule for CmpStmt

```
Function updateStateOnCmp(cmp):
    // Retrieve ICFGNode ℓ;
    // Retrieve the abstract state at ℓ;
    // 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 | $\ell : \mathtt{r} = \mathtt{phi}(\mathtt{p_1}, \mathtt{p_2}, \ldots, \mathtt{p_n})$ | $\sigma_{\underline{\ell}}(r) := \bigsqcup_{i=1}^{n} \sigma_{\overline{\ell}}(p_i)$ |




**Algorithm 18:** Abstract Execution Rule for PHISTMT

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

# Abstract Interpretation on GEPSTMT

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

$\overline{\ell}$

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

$\underline{\ell}$

$\sigma_{\underline{\ell}}(\texttt{p}) := \langle \perp, \{\texttt{o}_1.\texttt{fld}_1\} \rangle \sqcup \langle \perp, \{\texttt{o}_1.\texttt{fld}_2\} \rangle$

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

| | *Interval* | *MemAddress* |
|---|---|---|
| q | | $\{\texttt{o}_1\}$ |
| i | $[1, 2]$ | |

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

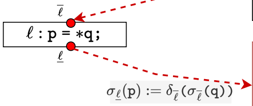
| | *Interval* | *MemAddress* |
|---|---|---|
| q | | $\{\texttt{o}_1\}$ |
| i | $[1, 2]$ | |
| p | | $\{\texttt{o}_1.\texttt{fld}_1, \texttt{o}_1.\texttt{fld}_2\}$ |

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

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

# Abstract Interpretation on LoadStmt

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




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

| | *Interval* | *MemAddress* |
|---|---|---|
| q | | $\{o_1\}$ |

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

| | *Interval* | *MemAddress* |
|---|---|---|
| $o_1$ | $[5, 5]$ | |

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

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

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

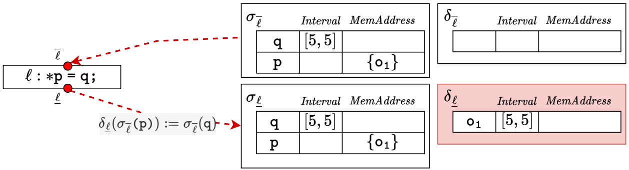
| | *Interval* | *MemAddress* |
|---|---|---|
| $o_1$ | $[5, 5]$ | |

**Algorithm 20:** Abstract Execution Rule for LoadStmt

```
Function updateStateOnLoad(load):
    // Retrieve ICFGNode ℓ;
    // Retrieve the abstract state as at ℓ;
    // 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 | $\ell : *\mathtt{p} = \mathtt{q}$ | $\delta_{\underline{\ell}} := (\{o \mapsto \sigma_{\overline{\ell}}(\mathtt{q}) \mid o \in \gamma(\sigma_{\overline{\ell}}(\mathtt{p}))\} \sqcup \delta_{\underline{\ell}})$ |




**Algorithm 21:** Abstract Execution Rule for STORESTMT

```
Function updateStateOnStore(store):
    // Retrieve ICFGNode ℓ;
    // Retrieve the abstract state as at ℓ;
    // Store RHS value to LHS via as.storeValue;
```

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

```c
extern void assert(int);

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

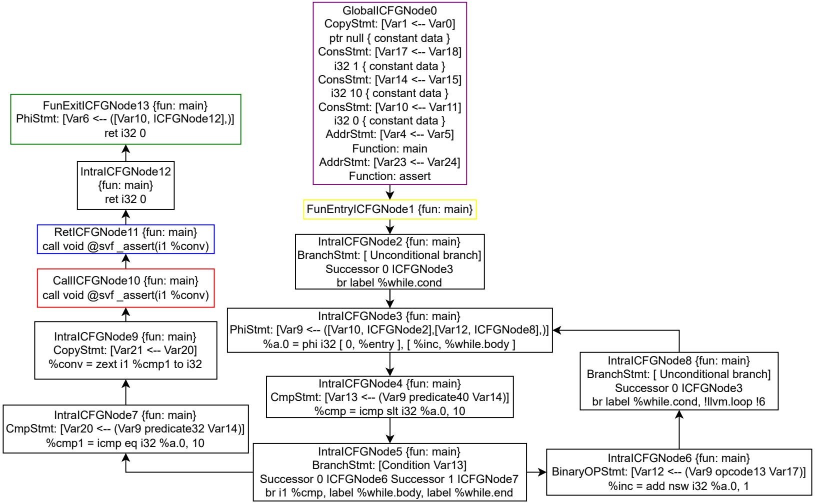
Source Code

Compile to LLVM IR

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

LLVM IR

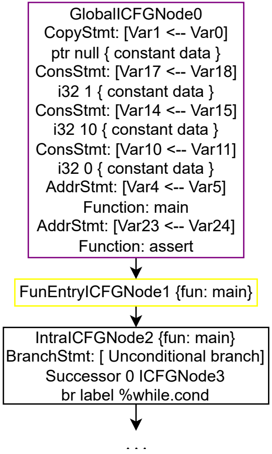
# An Example: Abstract Trace $\sigma$ for Top-level Variables




ICFG

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

## Before Entering Loop




ICFG

**Algorithm 22:** Abstract execution guided by WTO

```
Function handleStatement(ℓ):
    tmpAS := preAbsTrace[ℓ];
    if ℓ is ConsStmt or AddrStmt then
        updateStateOnAddr(ℓ);
    else if ℓ is CopyStmt then
        updateStateOnCopy(ℓ);
    ...;
```

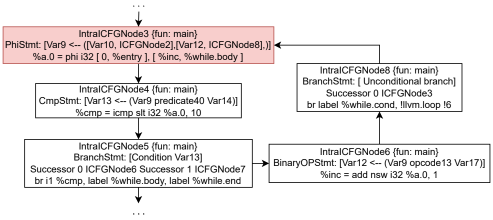
*postAbsTrace*[*ICFGNode0*].*varToAbsVal* :

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

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

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

## Widen Delay Phase (cur_iter is 0)




ICFG

*postAbsTrace*[*ICFGNode3*].*varToAbsVal* :

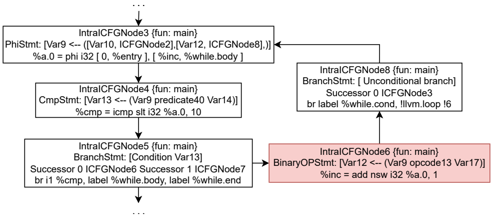
| SVFVar | AbstractValue⟨*Interval*, *MemAddress*⟩ |
|---|---|
| | ... |
| *Var10* | $\langle[0,0],\perp\rangle$ |
| *Var9* | $\langle[0,0],\perp\rangle$ |
| | ... |

**Algorithm 12:** Handle ICFG Cycle

```
Function handleICFGCycle (cycle):
    ℓ := cycle.getHead().getICFGNode();   // cycle head ICFGNode ℓ
    increasing := true;
    i := 0;   // analysis iteration for the loop
    while true do
        as_pre := σ_ℓ;   // abstract state in the last iteration
        handleICFGNode(ℓ);
        as_cur := σ_ℓ;   // abstract state in the current iteration
        if i ≥ Options.WidenDelay() then
            if increasing then
                σ_ℓ := as_pre ∇ as_cur;   // widening
                if σ_ℓ ≡ as_pre then
                    increasing := false;
                    continue;
            else
                σ_ℓ := as_pre Δ as_cur;   // narrowing
                if σ_ℓ ≡ as_pre then
                    break;
        // analyze remaining cycle components after two fixed-points
        foreach comp ∈ cycle.getWTOComponents() do
            if comp is Singleton then
                handleICFGNode(comp.getICFGNode())
            else if comp is Cycle then
                handleICFGCycle(comp);
        i++;
    return;
```

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

## Widen Delay Phase (cur_iter is 0)




ICFG

$postAbsTrace[ICFGNode6].varToAbsVal$ :

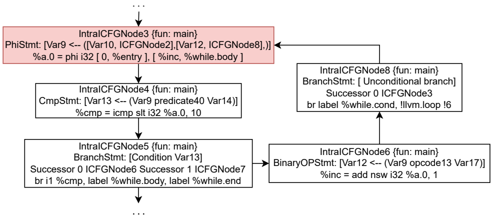
| SVFVar | AbstractValue⟨*Interval*, *MemAddress*⟩ |
|---|---|
| | ... |
| *Var10* | $\langle[0,0],\bot\rangle$ |
| *Var9* | $\langle[0,0],\bot\rangle$ |
| *Var12* | $\langle[1,1],\bot\rangle$ |
| | ... |

**Algorithm 12:** Handle ICFG Cycle

```
Function handleICFGCycle (cycle):
    ℓ := cycle.getHead().getICFGNode();   // cycle head ICFGNode ℓ
    increasing := true;
    i := 0;   // analysis iteration for the loop
    while true do
        as_pre := σ_ℓ;   // abstract state in the last iteration
        handleICFGNode(ℓ);
        as_cur := σ_ℓ;   // abstract state in the current iteration
        if i ≥ Options.WidenDelay() then
            if increasing then
                σ_ℓ := as_pre ∇ as_cur;   // widening
                if σ_ℓ ≡ as_pre then
                    increasing := false;
                    continue;
            else
                σ_ℓ := as_pre Δ as_cur;   // narrowing
                if σ_ℓ ≡ as_pre then
                    break;
        // analyze remaining cycle components after two fixed-points
        foreach comp ∈ cycle.getWTOComponents() do
            if comp is Singleton then
                handleICFGNode(comp.getICFGNode())
            else if comp is Cycle then
                handleICFGCycle(comp);
        i++;
    return;
```

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

## Widen Delay Phase (cur_iter is 1)




ICFG

*postAbsTrace*[*ICFGNode3*].*varToAbsVal* :

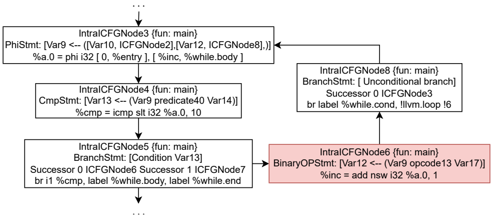
| SVFVar | AbstractValue⟨*Interval*, *MemAddress*⟩ |
|---|---|
| | ... |
| *Var9* | $\langle [0, 1], \bot \rangle$ |
| *Var12* | $\langle [1, 1], \bot \rangle$ |
| | ... |

**Algorithm 12:** Handle ICFG Cycle

```
Function handleICFGCycle (cycle):
    ℓ := cycle.getHead().getICFGNode();   // cycle head ICFGNode ℓ
    increasing := true;
    i := 0;   // analysis iteration for the loop
    while true do
        as_pre := σ_ℓ;   // abstract state in the last iteration
        handleICFGNode(ℓ);
        as_cur := σ_ℓ;   // abstract state in the current iteration
        if i ≥ Options.WidenDelay() then
            if increasing then
                σ_ℓ := as_pre ∇ as_cur;   // widening
                if σ_ℓ ≡ as_pre then
                    increasing := false;
                    continue;
            else
                σ_ℓ := as_pre Δ as_cur;   // narrowing
                if σ_ℓ ≡ as_pre then
                    break;
        // analyze remaining cycle components after two fixed-points
        foreach comp ∈ cycle.getWTOComponents() do
            if comp is Singleton then
                handleICFGNode(comp.getICFGNode())
            else if comp is Cycle then
                handleICFGCycle(comp);
        i++;
    return;
```

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

## Widen Delay Phase (cur_iter is 1)




ICFG

$postAbsTrace[ICFGNode6].varToAbsVal$ :

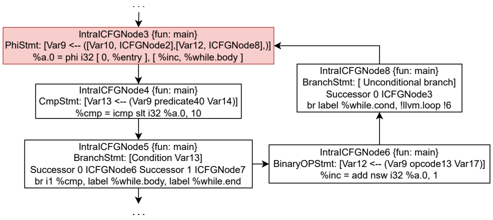
| SVFVar | AbstractValue⟨*Interval*, *MemAddress*⟩ |
|---|---|
| ... | |
| *Var9* | $\langle [0,1], \perp \rangle$ |
| *Var12* | $\langle [1,2], \perp \rangle$ |
| ... | |

**Algorithm 12:** Handle ICFG Cycle

```
Function handleICFGCycle (cycle):
    ℓ := cycle.getHead().getICFGNode();   // cycle head ICFGNode ℓ
    increasing := true;
    i := 0;   // analysis iteration for the loop
    while true do
        as_pre := σ_ℓ;   // abstract state in the last iteration
        handleICFGNode(ℓ);
        as_cur := σ_ℓ;   // abstract state in the current iteration
        if i ≥ Options.WidenDelay() then
            if increasing then
                σ_ℓ := as_pre ∇ as_cur;   // widening
                if σ_ℓ ≡ as_pre then
                    increasing := false;
                    continue;
            else
                σ_ℓ := as_pre Δ as_cur;   // narrowing
                if σ_ℓ ≡ as_pre then
                    break;
        // analyze remaining cycle components after two fixed-points
        foreach comp ∈ cycle.getWTOComponents() do
            if comp is Singleton then
                handleICFGNode(comp.getICFGNode())
            else if comp is Cycle then
                handleICFGCycle(comp);
        i++;
    return;
```

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

## Widen Phase (cur_iter is 2)




ICFG

$postAbsTrace[ICFGNode3].varToAbsVal$ :

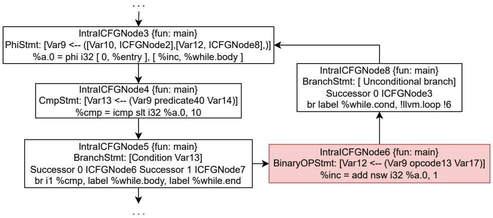
| SVFVar | $\langle Interval, MemAddress \rangle$ |
|---|---|
| | ... |
| *Var9* | $\langle [0, +\infty], \perp \rangle$ |
| *Var12* | $\langle [1, +\infty], \perp \rangle$ |
| | ... |

**Algorithm 12:** Handle ICFG Cycle

```
Function handleICFGCycle (cycle):
    ℓ := cycle.getHead().getICFGNode();   // cycle head ICFGNode ℓ
    increasing := true;
    i := 0;   // analysis iteration for the loop
    while true do
        as_pre := σ_ℓ;   // abstract state in the last iteration
        handleICFGNode(ℓ);
        as_cur := σ_ℓ;   // abstract state in the current iteration
        if i ≥ Options.WidenDelay() then
            if increasing then
                σ_ℓ := as_pre ∇ as_cur;   // widening
                if σ_ℓ ≡ as_pre then
                    increasing := false;
                    continue;
            else
                σ_ℓ := as_pre Δ as_cur;   // narrowing
                if σ_ℓ ≡ as_pre then
                    break;
        // analyze remaining cycle components after two fixed-points
        foreach comp ∈ cycle.getWTOComponents() do
            if comp is Singleton then
                handleICFGNode(comp.getICFGNode())
            else if comp is Cycle then
                handleICFGCycle(comp);
        i++;
    return;
```

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

## Widen Phase (cur_iter is 2)




ICFG

*postAbsTrace[ICFGNode6].varToAbsVal* :

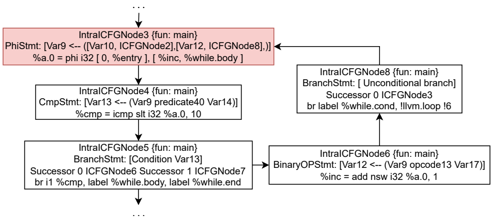
| SVFVar | AbstractValue⟨*Interval*, *MemAddress*⟩ |
|---|---|
| | ... |
| *Var9* | $\langle [0, 9], \bot \rangle$ |
| *Var12* | $\langle [1, 10], \bot \rangle$ |
| | ... |

**Algorithm 12:** Handle ICFG Cycle

```
Function handleICFGCycle (cycle):
    ℓ := cycle.getHead().getICFGNode();   // cycle head ICFGNode ℓ
    increasing := true;
    i := 0;   // analysis iteration for the loop
    while true do
        as_pre := σ_ℓ;   // abstract state in the last iteration
        handleICFGNode(ℓ);
        as_cur := σ_ℓ;   // abstract state in the current iteration
        if i ≥ Options.WidenDelay() then
            if increasing then
                σ_ℓ := as_pre ∇ as_cur;   // widening
                if σ_ℓ ≡ as_pre then
                    increasing := false;
                    continue;
            else
                σ_ℓ := as_pre Δ as_cur;   // narrowing
                if σ_ℓ ≡ as_pre then
                    break;
        // analyze remaining cycle components after two fixed-points
        foreach comp ∈ cycle.getWTOComponents() do
            if comp is Singleton then
                handleICFGNode(comp.getICFGNode())
            else if comp is Cycle then
                handleICFGCycle(comp);
        i++;
    return;
```

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

## Widen Phase Fixed Point




ICFG

$postAbsTrace[ICFGNode3].varToAbsVal$ :

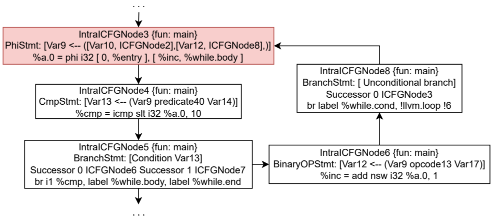
| SVFVar | $\langle Interval, MemAddress \rangle$ |
|---|---|
| | ... |
| *Var9* | $\langle [0, +\infty], \perp \rangle$ |
| *Var12* | $\langle [1, +\infty], \perp \rangle$ |
| | ... |

**Algorithm 12:** Handle ICFG Cycle

```
Function handleICFGCycle (cycle):
    ℓ := cycle.getHead().getICFGNode();   // cycle head ICFGNode ℓ
    increasing := true;
    i := 0;   // analysis iteration for the loop
    while true do
        as_pre := σ_ℓ;   // abstract state in the last iteration
        handleICFGNode(ℓ);
        as_cur := σ_ℓ;   // abstract state in the current iteration
        if i ≥ Options.WidenDelay() then
            if increasing then
                σ_ℓ := as_pre ∇ as_cur;   // widening
                if σ_ℓ ≡ as_pre then
                    increasing := false;
                    continue;
            else
                σ_ℓ := as_pre Δ as_cur;   // narrowing
                if σ_ℓ ≡ as_pre then
                    break;
        // analyze remaining cycle components after two fixed-points
        foreach comp ∈ cycle.getWTOComponents() do
            if comp is Singleton then
                handleICFGNode(comp.getICFGNode())
            else if comp is Cycle then
                handleICFGCycle(comp);
        i++;
    return;
```

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

## Narrow Phase




ICFG

$postAbsTrace[ICFGNode3].varToAbsVal$ :

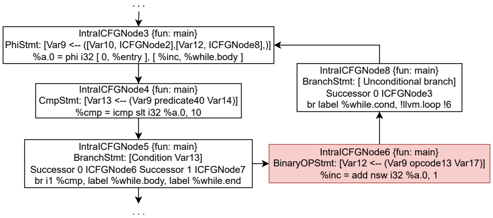
| SVFVar | $\langle Interval, MemAddress \rangle$ |
|---|---|
| | ... |
| $Var9$ | $\langle [0, 10], \perp \rangle$ |
| $Var12$ | $\langle [1, 10], \perp \rangle$ |
| | ... |

**Algorithm 12:** Handle ICFG Cycle

**Function** handleICFGCycle *(cycle)*:
- $\ell$ := cycle.getHead().getICFGNode(); // cycle head ICFGNode $\ell$
- increasing := true;
- i := 0; // analysis iteration for the loop
- **while** *true* **do**
  - $as_{pre} := \sigma_\ell$; // abstract state in the last iteration
  - handleICFGNode($\ell$);
  - $as_{cur} := \sigma_\ell$; // abstract state in the current iteration
  - **if** i $\geq$ Options.WidenDelay() **then**
    - **if** increasing **then**
      - $\sigma_\ell := as_{pre} \nabla as_{cur}$; // widening
      - **if** $\sigma_\ell \equiv as_{pre}$ **then**
        - increasing := false;
        - **continue**;
    - **else**
      - $\sigma_\ell := as_{pre} \Delta as_{cur}$; // narrowing
      - **if** $\sigma_\ell \equiv as_{pre}$ **then**
        - **break**;
  - // analyze remaining cycle components after two fixed-points
  - **foreach** comp $\in$ cycle.getWTOComponents() **do**
    - **if** comp *is Singleton* **then**
      - handleICFGNode(comp.getICFGNode())
    - **else if** comp *is Cycle* **then**
      - handleICFGCycle(comp);
  - i++;
- **return**;

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

## Narrow Phase




ICFG

$postAbsTrace[ICFGNode6].varToAbsVal$ :

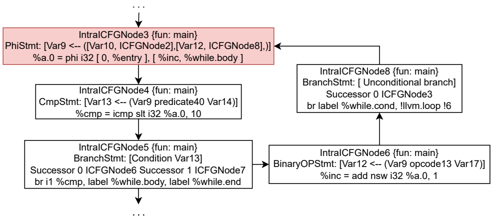
| SVFVar | $\langle Interval, MemAddress \rangle$ |
|---|---|
| | ... |
| *Var9* | $\langle [0, 9], \bot \rangle$ |
| *Var12* | $\langle [1, 10], \bot \rangle$ |
| | ... |

**Algorithm 12:** Handle ICFG Cycle

```
Function handleICFGCycle(cycle):
    ℓ := cycle.getHead().getICFGNode();   // cycle head ICFGNode ℓ
    increasing := true;
    i := 0;   // analysis iteration for the loop
    while true do
        as_pre := σ_ℓ;   // abstract state in the last iteration
        handleICFGNode(ℓ);
        as_cur := σ_ℓ;   // abstract state in the current iteration
        if i ≥ Options.WidenDelay() then
            if increasing then
                σ_ℓ := as_pre ∇ as_cur;   // widening
                if σ_ℓ ≡ as_pre then
                    increasing := false;
                    continue;
            else
                σ_ℓ := as_pre Δ as_cur;   // narrowing
                if σ_ℓ ≡ as_pre then
                    break;
        // analyze remaining cycle components after two fixed-points
        foreach comp ∈ cycle.getWTOComponents() do
            if comp is Singleton then
                handleICFGNode(comp.getICFGNode())
            else if comp is Cycle then
                handleICFGCycle(comp);
        i++;
    return;
```

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

## Narrow Phase Fixed Point




ICFG

*postAbsTrace*[*ICFGNode3*].*varToAbsVal* :

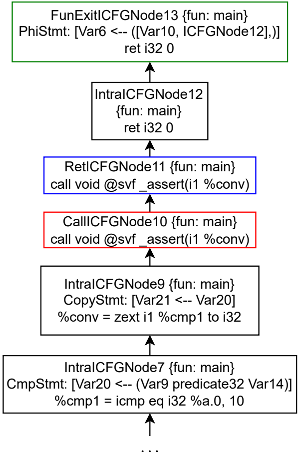
| SVFVar | $\langle Interval, MemAddress \rangle$ |
|---|---|
| | ... |
| *Var9* | $\langle [0, 10], \perp \rangle$ |
| *Var12* | $\langle [1, 10], \perp \rangle$ |
| | ... |

**Algorithm 12:** Handle ICFG Cycle

```
Function handleICFGCycle (cycle):
    ℓ := cycle.getHead().getICFGNode();   // cycle head ICFGNode ℓ
    increasing := true;
    i := 0;   // analysis iteration for the loop
    while true do
        as_pre := σ_ℓ;   // abstract state in the last iteration
        handleICFGNode(ℓ);
        as_cur := σ_ℓ;   // abstract state in the current iteration
        if i ≥ Options.WidenDelay() then
            if increasing then
                σ_ℓ := as_pre ∇ as_cur;   // widening
                if σ_ℓ ≡ as_pre then
                    increasing := false;
                    continue;
            else
                σ_ℓ := as_pre Δ as_cur;   // narrowing
                if σ_ℓ ≡ as_pre then
                    break;
        // analyze remaining cycle components after two fixed-points
        foreach comp ∈ cycle.getWTOComponents() do
            if comp is Singleton then
                handleICFGNode(comp.getICFGNode())
            else if comp is Cycle then
                handleICFGCycle(comp);
        i++;
    return;
```

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

## After Exiting Loop




ICFG

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

**Function** `handleStatement`($\ell$)**:**
  $tmpAS := preAbsTrace[\ell]$;
  **if** $\ell$ *is* CmpStmt **then**
    `updateStateOnCmp`($\ell$);
  $\ldots$;

$postAbsTrace[ICFGNode7].varToAbsVal$ :

| SVFVar | $\langle Interval, MemAddress \rangle$ |
|---|---|
| | $\ldots$ |
| *Var9* | $\langle [10, 10], \perp \rangle$ |
| *Var20* | $\langle [1, 1], \perp \rangle$ |
| | $\ldots$ |