Abstract Interpretation for Code Analysis and Verification

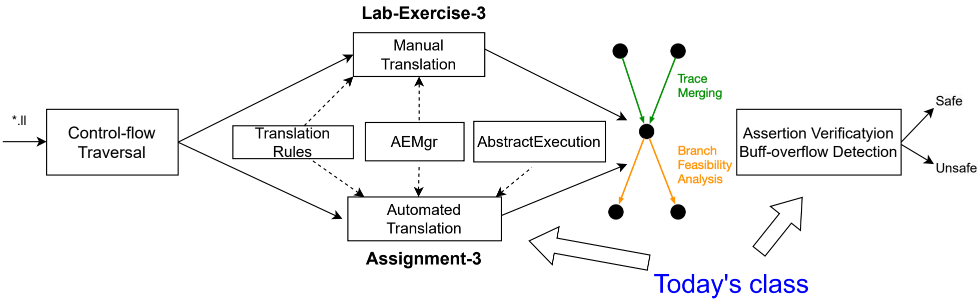
(Week 9)

Yulei Sui

School of Computer Science and Engineering

University of New South Wales, Australia

Today’s class



Topological Order

Analysis Order of Nodes on Control-Flow Graph

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

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

Topological Order

Analysis Order of Nodes on Control-Flow Graph

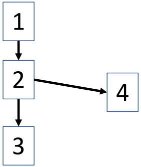
? How to analyze a program **free of loop**?

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

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

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

Example of topological order:



acyclic graph G

1 2 3 4 ✓

1 2 4 3 ✓

1 3 2 4 ✗

Valid/invalid topological order

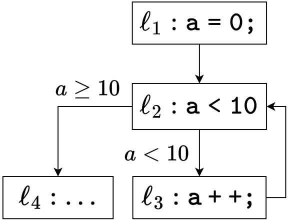
Weak Topological Order

Analysis Order of Nodes on Control-Flow Graph

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

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

What is the weak topological order?



Control Flow Graph

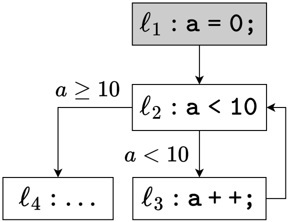
Weak Topological Order

Analysis Order of Nodes on Control-Flow Graph

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

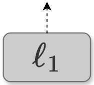
✓ We can analyze a program containing loops adhering to the **weak topological order** (WTO) on its control flow graph.

What is the weak topological order?



Control Flow Graph

1^{st} WTO component: a sigle node



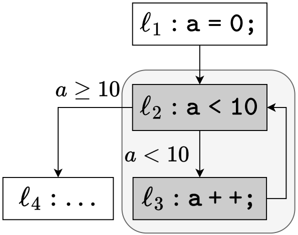
Weak Topological Order

Analysis Order of Nodes on Control-Flow Graph

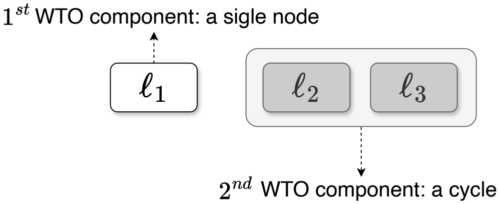
? How to analyze a program **containing loops**?

✓ 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



1^{st} WTO component: a sigle node
ℓ_1
ℓ_2
ℓ_3
2^{nd} WTO component: a cycle

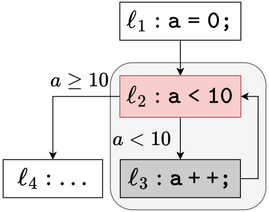
Weak Topological Order

Analysis Order of Nodes on Control-Flow Graph

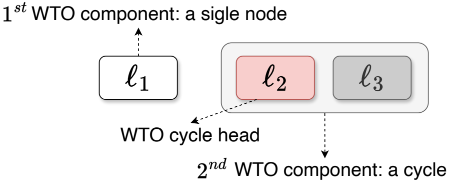
? How to analyze a program **containing loops**?

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

What is the weak topological order?



Control Flow Graph



1^{st} WTO component: a sigle node
ℓ_1
ℓ_2
ℓ_3
WTO cycle head
2^{nd} WTO component: a cycle

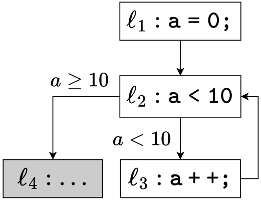
Weak Topological Order

Analysis Order of Nodes on Control-Flow Graph

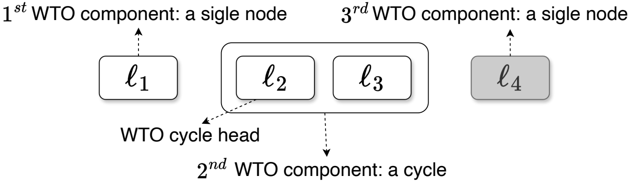
? How to analyze a program **containing loops**?

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

What is the weak topological order?



Control Flow Graph



1^{st} WTO component: a sigle node
3^{rd} WTO component: a sigle node
ℓ_1
ℓ_2
ℓ_3
ℓ_4
WTO cycle head
2^{nd} WTO component: a cycle

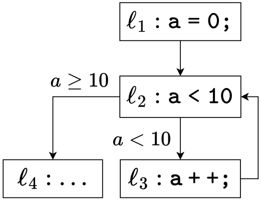
Weak Topological Order

Analysis Order of Nodes on Control-Flow Graph

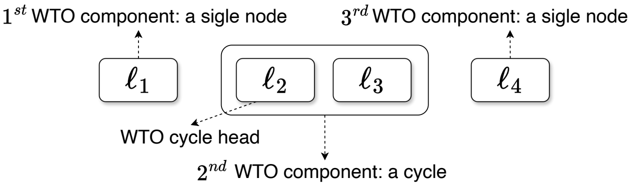
? How to analyze a program **containing loops**?

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

What is the weak topological order?



Control Flow Graph



1^{st} WTO component: a sigle node
3^{rd} WTO component: a sigle node
ℓ_1
ℓ_2
ℓ_3
ℓ_4
WTO cycle head
2^{nd} WTO component: a cycle

Analyze each node following the WTO

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

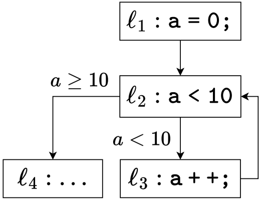
Weak Topological Order

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?



ℓ_1 : a = 0;

$a \geq 10$

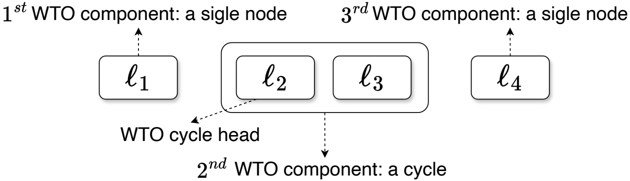
ℓ_2 : a < 10

$a < 10$

ℓ_4 : . . .

ℓ_3 : a++;

Control Flow Graph



1^{st} WTO component: a sigle node

3^{rd} WTO component: a sigle node

ℓ_1

ℓ_2

ℓ_3

ℓ_4

WTO cycle head

2^{nd} WTO component: a cycle

Analyze each node following the WTO

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

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

WTO, Widening and Narrowing

Why Weak Topological Order?

- Handling cyclic dependencies
- Efficient fixed-point computation

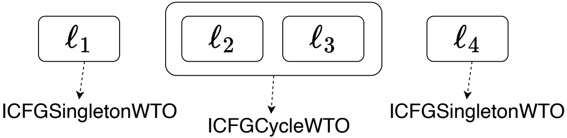
Why Widening?

- Over-approximation
- Prevent non-termination

Why Narrowing?

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

Widening and Narrowing

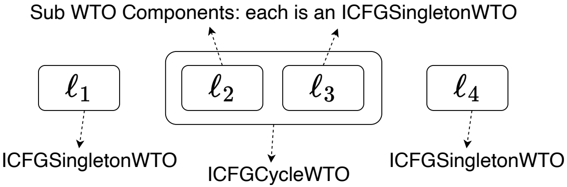


Analyze ICFGCycleWTO

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

```
Function handleCycleWTO(cycle):
    cycle_head := cycle→head()→node() ;
    increasing := true ;
    cur_iter := 0 ;
    while true do
        cur_iter++;
        if $cur\_iter \geq Options :: WidenDelay()$ then
            prev_head_state := postAbsTrace[cycle_head];
            handleSingletonWTO(cycle→head());
            cur_head_state := postAbsTrace[cycle_head];
            if increasing then
                postAbsTrace[cycle_head] := prev_head_state.widen(cur_head_state) ;
                if postAbsTrace[cycle_head] == prev_head_state then
                    increasing := false;
                    continue;
            else
                postAbsTrace[cycle_head] := prev_head_state.narrow(cur_head_state) ;
                if postAbsTrace[cycle_head] == prev_head_state then
                    break ;
        else
            handleSingletonWTO(cycle→head());
        handleWTOComponents(cycle→getWTOComponents());
```

Widening and Narrowing

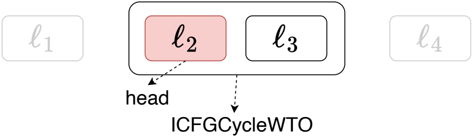


Analyze ICFGCycleWTO

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

```
Function handleCycleWTO(cycle):
    cycle_head := cycle→head()→node() ;
    increasing := true ;
    cur_iter := 0 ;
    while true do
        cur_iter++;
        if cur_iter ≥ Options :: WidenDelay() then
            prev_head_state := postAbsTrace[cycle_head];
            handleSingletonWTO(cycle→head());
            cur_head_state := postAbsTrace[cycle_head];
            if increasing then
                postAbsTrace[cycle_head] := prev_head_state.widen(cur_head_state) ;
                if postAbsTrace[cycle_head] == prev_head_state then
                    increasing := false;
                    continue;
            else
                postAbsTrace[cycle_head] := prev_head_state.narrow(cur_head_state) ;
                if postAbsTrace[cycle_head] == prev_head_state then
                    break ;
        else
            handleSingletonWTO(cycle→head());
        handleWTOComponents(cycle→getWTOComponents());
```

Weak Topological Order

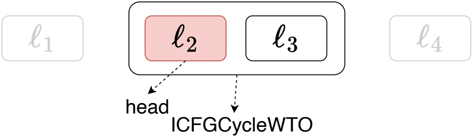


Get ICFGCycleWTO head ℓ_2

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

```
Function handleCycleWTO(cycle):
    cycle_head := cycle→head()→node() ;
    increasing := true ;
    cur_iter := 0 ;
    while true do
        cur_iter++;
        if cur_iter ≥ Options :: WidenDelay() then
            prev_head_state := postAbsTrace[cycle_head];
            handleSingletonWTO(cycle→head());
            cur_head_state := postAbsTrace[cycle_head];
            if increasing then
                postAbsTrace[cycle_head] := prev_head_state.widen(cur_head_state) ;
                if postAbsTrace[cycle_head] == prev_head_state then
                    increasing := false;
                    continue;
            else
                postAbsTrace[cycle_head] := prev_head_state.narrow(cur_head_state) ;
                if postAbsTrace[cycle_head] == prev_head_state then
                    break ;
        else
            handleSingletonWTO(cycle→head());
        handleWTOComponents(cycle→getWTOComponents());
```

Weak Topological Order

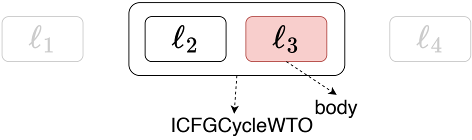


Handle ICFGCycleWTO head ℓ_2

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

```
Function handleCycleWTO(cycle):
    cycle_head := cycle→head()→node() ;
    increasing := true ;
    cur_iter := 0 ;
    while true do
        cur_iter++;
        if cur_iter ≥ Options :: WidenDelay() then
            prev_head_state := postAbsTrace[cycle_head];
            handleSingletonWTO(cycle→head());
            cur_head_state := postAbsTrace[cycle_head];
            if increasing then
                postAbsTrace[cycle_head] := prev_head_state.widen(cur_head_state) ;
                if postAbsTrace[cycle_head] == prev_head_state then
                    increasing := false;
                    continue;
            else
                postAbsTrace[cycle_head] := prev_head_state.narrow(cur_head_state) ;
                if postAbsTrace[cycle_head] == prev_head_state then
                    break ;
        else
            handleSingletonWTO(cycle→head());
        handleWTOComponents(cycle→getWTOComponents());
```

Widening and Narrowing



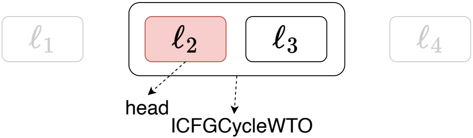
Handle ICFGCycleWTO body ℓ_3

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

```
Function handleCycleWTO(cycle):
    cycle_head := cycle→head()→node() ;
    increasing := true ;
    cur_iter := 0 ;
    while true do
        cur_iter++;
        if cur_iter ≥ Options :: WidenDelay() then
            prev_head_state := postAbsTrace[cycle_head];
            handleSingletonWTO(cycle→head());
            cur_head_state := postAbsTrace[cycle_head];
            if increasing then
                postAbsTrace[cycle_head] := prev_head_state.widen(cur_head_state) ;
                if postAbsTrace[cycle_head] == prev_head_state then
                    increasing := false;
                    continue;
            else
                postAbsTrace[cycle_head] := prev_head_state.narrow(cur_head_state) ;
                if postAbsTrace[cycle_head] == prev_head_state then
                    break ;
        else
            handleSingletonWTO(cycle→head());
        handleWTOComponents(cycle→getWTOComponents());
```

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

Widening and Narrowing



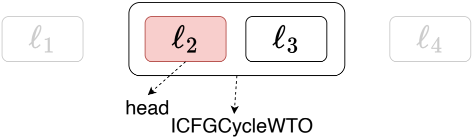
When $cur\_iter \geq Options :: WidenDelay()$

perform widening on ℓ_2

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

```
Function handleCycleWTO(cycle):
    cycle_head := cycle→head()→node() ;
    increasing := true ;
    cur_iter := 0 ;
    while true do
        cur_iter++;
        if $cur\_iter \geq Options :: WidenDelay()$ then
            $prev\_head\_state := postAbsTrace[cycle\_head]$;
            handleSingletonWTO(cycle→head());
            $cur\_head\_state := postAbsTrace[cycle\_head]$;
            if increasing then
                $postAbsTrace[cycle\_head] := prev\_head\_state.widen(cur\_head\_state)$ ;
                if $postAbsTrace[cycle\_head] == prev\_head\_state$ then
                    increasing := false;
                    continue;
            else
                $postAbsTrace[cycle\_head] := prev\_head\_state.narrow(cur\_head\_state)$ ;
                if $postAbsTrace[cycle\_head] == prev\_head\_state$ then
                    break ;
        else
            handleSingletonWTO(cycle→head());
        handleWTOComponents(cycle→getWTOComponents());
```

Widening and Narrowing

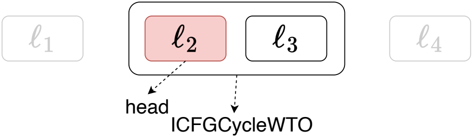


Widening reaches a fixpoint

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

```
Function handleCycleWTO(cycle):
    cycle_head := cycle→head()→node() ;
    increasing := true ;
    cur_iter := 0 ;
    while true do
        cur_iter++;
        if cur_iter ≥ Options :: WidenDelay() then
            prev_head_state := postAbsTrace[cycle_head];
            handleSingletonWTO(cycle→head());
            cur_head_state := postAbsTrace[cycle_head];
            if increasing then
                postAbsTrace[cycle_head] := prev_head_state.widen(cur_head_state) ;
                if postAbsTrace[cycle_head] == prev_head_state then
                    increasing := false;
                    continue;
            else
                postAbsTrace[cycle_head] := prev_head_state.narrow(cur_head_state) ;
                if postAbsTrace[cycle_head] == prev_head_state then
                    break ;
        else
            handleSingletonWTO(cycle→head());
        handleWTOComponents(cycle→getWTOComponents());
```

Widening and Narrowing

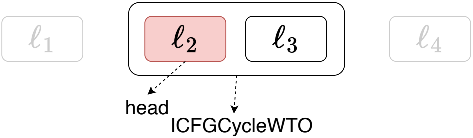


Perform narrowing on ℓ_2

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

```
Function handleCycleWTO(cycle):
    cycle_head := cycle→head()→node() ;
    increasing := true ;
    cur_iter := 0 ;
    while true do
        cur_iter++;
        if cur_iter ≥ Options :: WidenDelay() then
            prev_head_state := postAbsTrace[cycle_head];
            handleSingletonWTO(cycle→head());
            cur_head_state := postAbsTrace[cycle_head];
            if increasing then
                postAbsTrace[cycle_head] := prev_head_state.widen(cur_head_state) ;
                if postAbsTrace[cycle_head] == prev_head_state then
                    increasing := false;
                    continue;
            else
                postAbsTrace[cycle_head] := prev_head_state.narrow(cur_head_state) ;
                if postAbsTrace[cycle_head] == prev_head_state then
                    break ;
        else
            handleSingletonWTO(cycle→head());
        handleWTOComponents(cycle→getWTOComponents());
```

Widening and Narrowing



Narrowing reaches a fixpoint, Finish!

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

```
Function handleCycleWTO(cycle):
    cycle_head := cycle→head()→node() ;
    increasing := true ;
    cur_iter := 0 ;
    while true do
        cur_iter++;
        if cur_iter ≥ Options :: WidenDelay() then
            prev_head_state := postAbsTrace[cycle_head];
            handleSingletonWTO(cycle→head());
            cur_head_state := postAbsTrace[cycle_head];
            if increasing then
                postAbsTrace[cycle_head] := prev_head_state.widen(cur_head_state) ;
                if postAbsTrace[cycle_head] == prev_head_state then
                    increasing := false;
                    continue;
            else
                postAbsTrace[cycle_head] := prev_head_state.narrow(cur_head_state) ;
                if postAbsTrace[cycle_head] == prev_head_state then
                    break ;
        else
            handleSingletonWTO(cycle→head());
        handleWTOComponents(cycle→getWTOComponents());
```

Abstract Interpretation on SVFIR

Week 9

Yulei Sui

School of Computer Science and Engineering

University of New South Wales, Australia

Abstract Interpretation on Pointer-Free SVFIR

Interval Domain

- For simplicity, let's first consider abstract execution on a pointer-free language.
- This means there are no operations for memory allocation (like $\texttt{p} = \texttt{alloc}_\texttt{o}$) 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	$\ell : \texttt{r} = \texttt{phi}(\texttt{p}_1, \texttt{p}_2, \ldots, \texttt{p}_\texttt{n})$
Sequence	$\ell_1 ; \ell_2$
BranchStmt	ℓ_1 : if$(x < c)$ then ℓ_2 else ℓ_3

Abstract Interpretation on Pointer-Free SVFIR

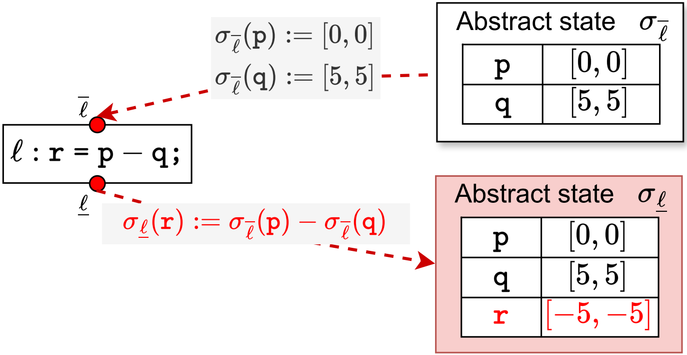
Interval Domain

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

SVFSTMT	**C-Like form**	**Abstract Execution Rule**
CONSSTMT	$\ell : \mathtt{p} = \mathtt{c}$	$\sigma_{\underline{\ell}}(\mathtt{p}) := [\mathtt{c}, \mathtt{c}]$
COPYSTMT	$\ell : \mathtt{p} = \mathtt{q}$	$\sigma_{\underline{\ell}}(\mathtt{p}) := \sigma_{\overline{\ell}}(\mathtt{q})$
BINARYSTMT	$\ell : \mathtt{r} = \mathtt{p} \otimes \mathtt{q}$	$\sigma_{\underline{\ell}}(r) := \sigma_{\overline{\ell}}(p) \hat{\otimes} \sigma_{\overline{\ell}}(q)$
PHISTMT	$\ell : \mathtt{r} = \mathtt{phi}(\mathtt{p_1}, \mathtt{p_2}, \ldots, \mathtt{p_n})$	$\sigma_{\underline{\ell}}(r) := \bigsqcup_{i=1}^{n} \sigma_{\overline{\ell}}(p_i)$
SEQUENCE	$\ell_1 ; \ell_2$	$\forall v \in \mathbb{V}, \sigma_{\overline{\ell_2}}(v) \sqsupseteq \sigma_{\underline{\ell_1}}(v)$
BRANCHSTMT	ℓ_1 : if$(x < c)$ then ℓ_2 else ℓ_3	$\sigma_{\overline{\ell_2}}(x) := \sigma_{\underline{\ell_1}}(x) \sqcap [-\infty, c-1],$ if $\sigma_{\underline{\ell_1}}(x) \sqcap [-\infty, c-1] \neq \perp$ $\sigma_{\overline{\ell_3}}(x) := \sigma_{\underline{\ell_1}}(x) \sqcap [c, +\infty],$ if $\sigma_{\underline{\ell_1}}(x) \sqcap [c, +\infty] \neq \perp$

Abstract Interpretation on BinaryStmt

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



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

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

$\overline{\ell}$

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

$\underline{\ell}$

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

Abstract state $\sigma_{\overline{\ell}}$

$\mathtt{p}$	$[0, 0]$
$\mathtt{q}$	$[5, 5]$

Abstract state $\sigma_{\underline{\ell}}$

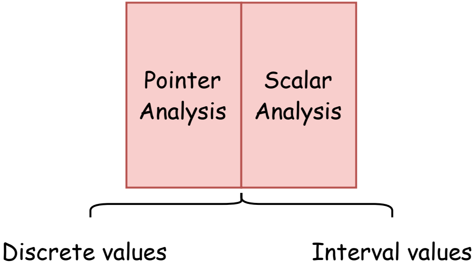
$\mathtt{p}$	$[0, 0]$
$\mathtt{q}$	$[5, 5]$
$\mathtt{r}$	$[-5, -5]$

Abstract Interpretation in the Presence of Pointers

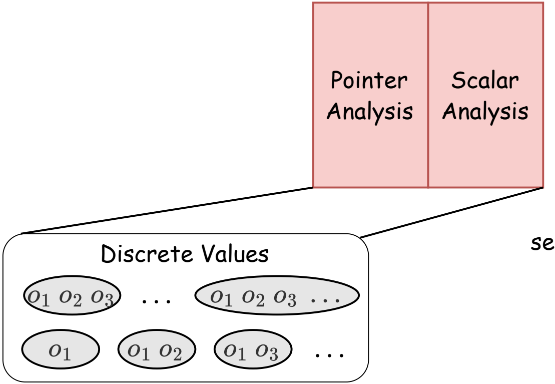
- SVFIR in the presence of pointers contain pointer-related statements including AddrStmt, GepStmt, LoadStmt and StoreStmt.
- Abstract interpretation needs to be performed on **a combined domain of intervals and addresses**.

SVFStmt	**C-Like form**
ConsStmt	$\ell : \mathtt{p} = \mathtt{c}$
CopyStmt	$\ell : \mathtt{p} = \mathtt{q}$
BinaryStmt	$\ell : \mathtt{r} = \mathtt{p} \otimes \mathtt{q}$
PhiStmt	$\ell : \mathtt{r} = \mathtt{phi}(\mathtt{p}_1, \mathtt{p}_2, \ldots, \mathtt{p}_n)$
Sequence	$\ell_1 ; \ell_2$
BranchStmt	ℓ_1 : if($x < c$) then ℓ_2 else ℓ_3
AddrStmt	$\ell : \mathtt{p} = \mathtt{alloc}$
GepStmt	$\ell : \mathtt{p} = \&(\mathtt{q} \rightarrow \mathtt{i})$ or $\mathtt{p} = \&\mathtt{q}[\mathtt{i}]$
LoadStmt	$\ell : \mathtt{p} = *\mathtt{q}$
StoreStmt	$\ell : *\mathtt{p} = \mathtt{q}$

Combined Analysis



Combined Analysis Using Discrete Values



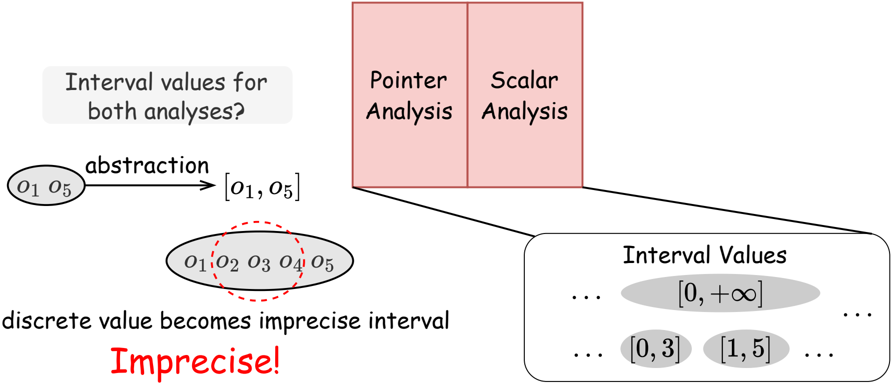
Discrete values for both analyses?

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

set of all possible values with inputs

Unscalable!

Combined Analysis Using Interval Values



Interval values for both analyses?

Pointer Analysis

Scalar Analysis

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

o_1 o_2 o_3 o_4 o_5

discrete value becomes imprecise interval

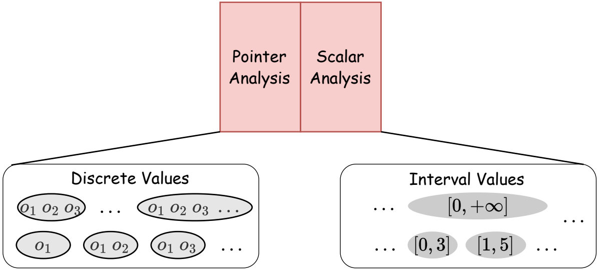
Imprecise!

Interval Values

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

$\ldots$ $[0, 3]$ $[1, 5]$ $\ldots$

Abstract Interpretation Over a Combined Domain

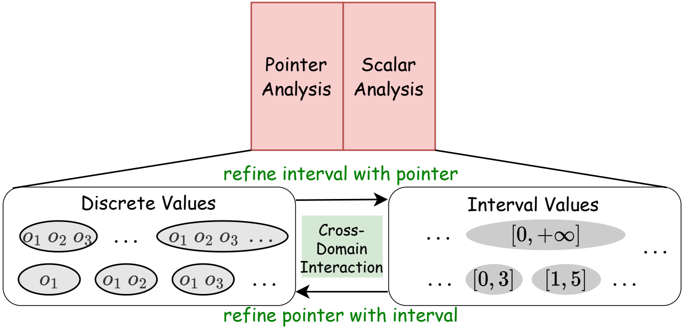


```
p = malloc(m*sizeof(int)); // p points to an array of size m
q = malloc(n*sizeof(int)); // q points to an array of size n
```

```
m = r[i];
```

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

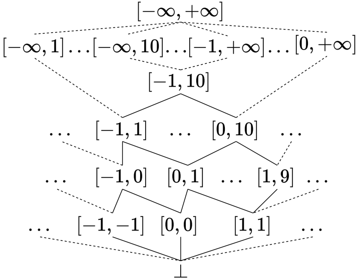
Abstract Interpretation Over a Combined Domain



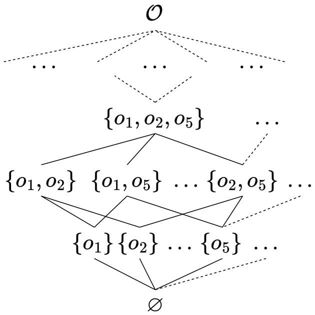
We require **a combination of interval and memory address domains** to precisely and efficiently perform abstract execution on SVFIR in the presence of pointers.

Abstract Interpretation over Interval and MemAddress Domains

A Combined Domain of Intervals and Discrete Memory Addresses


$[-\infty, +\infty]$
$[-\infty, 1] \ldots [-\infty, 10] \ldots [-1, +\infty] \ldots [0, +\infty]$
$[-1, 10]$
$\ldots$ $[-1, 1]$ $\ldots$ $[0, 10]$ $\ldots$
$\ldots$ $[-1, 0]$ $[0, 1]$ $\ldots$ $[1, 9]$ $\ldots$
$\ldots$ $[-1, -1]$ $[0, 0]$ $[1, 1]$ $\ldots$
$\perp$

Interval domain for scalar variables


$\mathcal{O}$
$\ldots$ $\ldots$ $\ldots$
$\{o_1, o_2, o_5\}$ $\ldots$
$\{o_1, o_2\}$ $\{o_1, o_5\}$ $\ldots$ $\{o_2, o_5\}$ $\ldots$
$\{o_1\}$ $\{o_2\}$ $\ldots$ $\{o_5\}$ $\ldots$
$\varnothing$

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

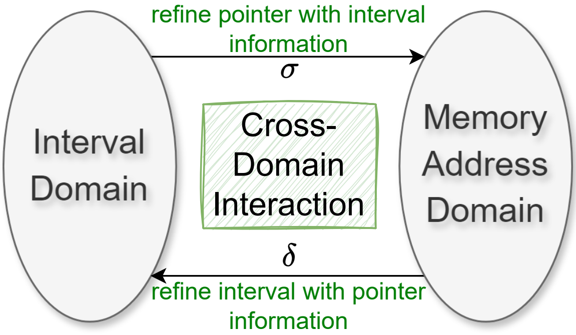
	Notation	**Domain**	**Data Structure Implementation**
Abstract trace	σ	$\mathbb{L} \times \mathbb{P} \rightarrow \mathit{Interval} \times \mathit{MemAddress}$	*preAbsTrace*, *postAbsTrace*
	δ	$\mathbb{L} \times \mathbb{O} \rightarrow \mathit{Interval} \times \mathit{MemAddress}$	
Abstract state	σ_L	$\mathbb{P} \rightarrow \mathit{Interval} \times \mathit{MemAddress}$	*AbstractState.varToAbsVal*
	δ_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 State and Abstract Trace

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

Abstract Trace for The Combined Domain


refine pointer with interval information
σ
Interval Domain
Cross-Domain Interaction
Memory Address Domain
δ
refine interval with pointer information

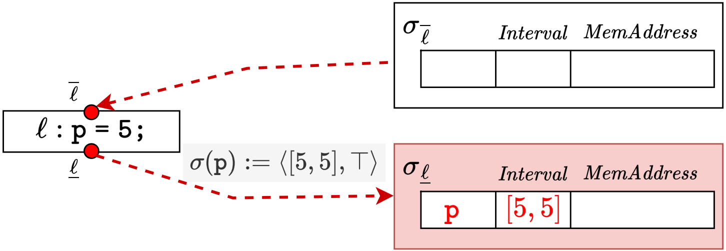
Abstract Execution Rules on SVFIR in the Presence of Pointers

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

SVFSTMT	C-Like form	Abstract Execution Rule
CONSSTMT	$\ell : \mathtt{p} = \mathtt{c}$	$\sigma_{\underline{\ell}}(\mathtt{p}) := \langle [\mathtt{c}, \mathtt{c}], \top \rangle$
COPYSTMT	$\ell : \mathtt{p} = \mathtt{q}$	$\sigma_{\underline{\ell}}(\mathtt{p}) := \sigma_{\overline{\ell}}(\mathtt{q})$
BINARYSTMT	$\ell : \mathtt{r} = \mathtt{p} \otimes \mathtt{q}$	$\sigma_{\underline{\ell}}(r) := \sigma_{\overline{\ell}}(p) \hat{\otimes} \sigma_{\overline{\ell}}(q)$
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	ℓ_1 : if$(x < c)$ then ℓ_2 else ℓ_3	$\sigma_{\overline{\ell_2}}(x) := \sigma_{\underline{\ell_1}}(x) \sqcap [-\infty, c-1],\ \text{if}\ \sigma_{\underline{\ell_1}}(x) \sqcap [-\infty, c-1] \neq \perp$ $\sigma_{\overline{\ell_3}}(x) := \sigma_{\underline{\ell_1}}(x) \sqcap [c, +\infty],\ \text{if}\ \sigma_{\underline{\ell_1}}(x) \sqcap [c, +\infty] \neq \perp$
SEQUENCE	$\ell_1 ; \ell_2$	$\delta_{\overline{\ell_2}} \sqsupseteq \delta_{\underline{\ell_1}}, \sigma_{\overline{\ell_2}} \sqsupseteq \sigma_{\underline{\ell_1}}$
ADDRSTMT	$\ell : \mathtt{p} = \mathtt{alloc}_{\mathtt{o_i}}$	$\sigma_{\underline{\ell}}(\mathtt{p}) := \langle \top, \{o_i\} \rangle$
GEPSTMT	$\ell : \mathtt{p} = \&(\mathtt{q} \rightarrow \mathtt{i})$ or $\mathtt{p} = \&\mathtt{q}[\mathtt{i}]$	$\sigma_{\underline{\ell}}(\mathtt{p}) := \bigsqcup_{\mathtt{o} \in \gamma(\sigma_{\overline{\ell}}(\mathtt{q}))} \bigsqcup_{j \in \gamma(\sigma_{\overline{\ell}}(\mathtt{i}))} \langle \top, \{\mathtt{o.fld}_j\} \rangle$
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 Execution on ConsStmt

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

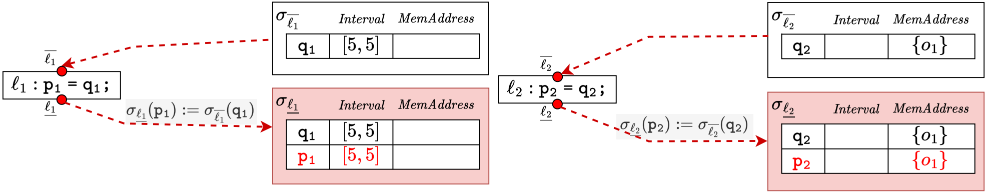

$\overline{\ell}$
ℓ : p = 5;
$\underline{\ell}$
$\sigma(\mathrm{p}) := \langle [5, 5], \top \rangle$
$\sigma_{\overline{\ell}}$
Interval
MemAddress
$\sigma_{\underline{\ell}}$
Interval
MemAddress
p
$[5, 5]$

Algorithm 10: 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 Execution on CopyStmt

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



$\overline{\ell_1}$

$\ell_1 : p_1 = q_1;$

$\underline{\ell_1}$

$\sigma_{\underline{\ell_1}}(p_1) := \sigma_{\overline{\ell_1}}(q_1)$

$\sigma_{\overline{\ell_1}}$	*Interval*	*MemAddress*
q_1	$[5, 5]$	

$\sigma_{\underline{\ell_1}}$	*Interval*	*MemAddress*
q_1	$[5, 5]$	
p_1	$[5, 5]$	

$\overline{\ell_2}$

$\ell_2 : p_2 = q_2;$

$\underline{\ell_2}$

$\sigma_{\underline{\ell_2}}(p_2) := \sigma_{\overline{\ell_2}}(q_2)$

$\sigma_{\overline{\ell_2}}$	*Interval*	*MemAddress*
q_2		$\{o_1\}$

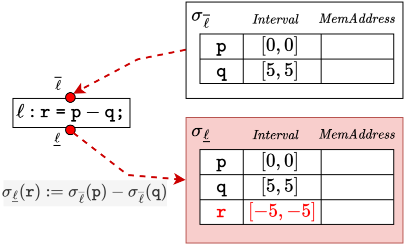
$\sigma_{\underline{\ell_2}}$	*Interval*	*MemAddress*
q_2		$\{o_1\}$
p_2		$\{o_1\}$

Algorithm 11: Abstract Execution Rule for CopyStmt

```
Function updateStateOnCopy(copy):
    node = copy → getICFGNode();
    as = getAbsStateFromTrace(node);
    lhs = copy → getLHSVarID();
    rhs = copy → getRHSVarID();
    as[lhs] = as[rhs];
```

Abstract Execution on BinaryStmt

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



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

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

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

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

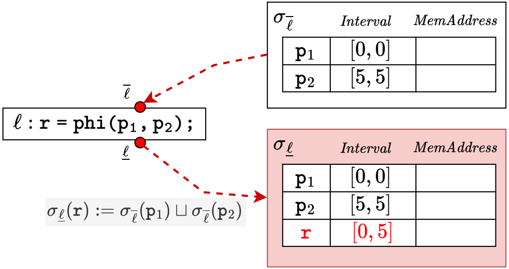
$\sigma_{\underline{\ell}}(\mathtt{r}) := \sigma_{\overline{\ell}}(\mathtt{p}) - \sigma_{\overline{\ell}}(\mathtt{q})$

Algorithm 12: Abstract Execution Rule for BinaryStmt

```
Function updateStateOnBinary(binary):
    node = binary→getICFGNode();
    as = getAbsStateFromTrace(node);
    op0 = binary→getOpVarID(0);
    op1 = binary→getOpVarID(1);
    res = binary→getResID();
    if !as.inVarToValTable(op0) then
        as[op0] = IntervalValue :: top()
    if !as.inVarToValTable(op1) then
        as[op1] = IntervalValue :: top()
    if
     as.inVarToValTable(op0)&&as.inVarToValTable(op1)
     then
        as[res] = as[op0]$\hat{\otimes}$as[op1]
```

Abstract Execution on PhiStmt

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



$\overline{\ell}$

ℓ : r = phi(p₁, p₂);

$\underline{\ell}$

$\sigma_{\underline{\ell}}(\mathtt{r}) := \sigma_{\overline{\ell}}(\mathtt{p_1}) \sqcup \sigma_{\overline{\ell}}(\mathtt{p_2})$

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

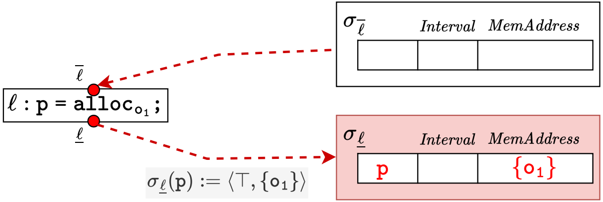
$\sigma_{\underline{\ell}}$	*Interval*	*MemAddress*
$\mathtt{p_1}$	$[0, 0]$	
$\mathtt{p_2}$	$[5, 5]$	
$\mathtt{r}$	$[0, 5]$	

Algorithm 13: Abstract Execution Rule for PhiStmt

```
Function updateStateOnPhi(phi):
    node = phi → getICFGNode();
    as = getAbsStateFromTrace(node);
    res = phi → getResID();
    rhs = AbstractValue();
    for i = 0; i < phi → getOpVarNum(); i + + do
        curId = phi → getOpVarID(i);
        rhs.join_with(as[curId])
    as[res] = rhs
```

Abstract Execution on AddrStmt

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

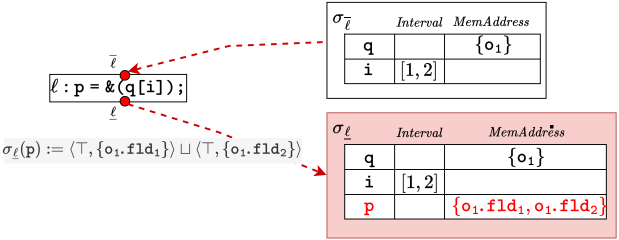

$\overline{\ell}$
ℓ : p = alloc$_{o_1}$;
$\underline{\ell}$
$\sigma_{\overline{\ell}}$
Interval
MemAddress
$\sigma_{\underline{\ell}}$
Interval
MemAddress
p
$\{o_1\}$
$\sigma_{\underline{\ell}}(\mathtt{p}) := \langle \top, \{o_1\} \rangle$

Algorithm 14: 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 Execution on GepStmt

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



$\overline{\ell}$
ℓ : p = &(q[i]);
$\underline{\ell}$
$\sigma_{\underline{\ell}}(\texttt{p}) := \langle \top, \{\texttt{o}_1\texttt{.fld}_1\}\rangle \sqcup \langle \top, \{\texttt{o}_1\texttt{.fld}_2\}\rangle$

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

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

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

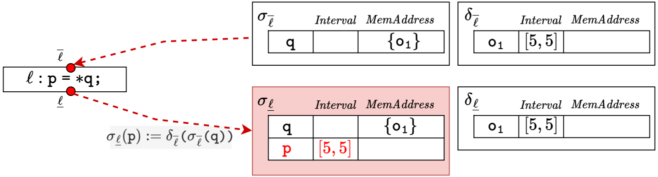
	Interval	*MemAddress*
q		$\{\texttt{o}_1\}$
i	$[1, 2]$	
p		$\{\texttt{o}_1\texttt{.fld}_1, \texttt{o}_1\texttt{.fld}_2\}$

Algorithm 15: Abstract Execution Rule for GepStmt

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

Abstract Execution on LoadStmt

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



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

$\sigma_{\overline{\ell}}$	*Interval*	*MemAddress*
q		$\{\mathtt{o}_1\}$

$\delta_{\overline{\ell}}$	*Interval*	*MemAddress*
$\mathtt{o}_1$	$[5, 5]$	

$\sigma_{\underline{\ell}}$	*Interval*	*MemAddress*
q		$\{\mathtt{o}_1\}$
p	$[5, 5]$	

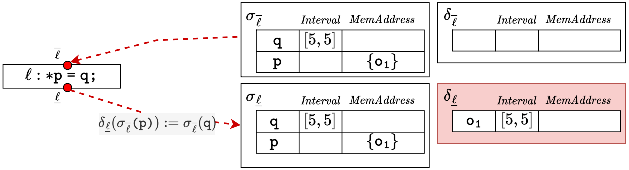
$\delta_{\underline{\ell}}$	*Interval*	*MemAddress*
$\mathtt{o}_1$	$[5, 5]$	

Algorithm 16: Abstract Execution Rule for LoadStmt

```
Function updateStateOnLoad(load):
    node = load→getICFGNode();
    as = getAbsStateFromTrace(node);
    rhs = load→getRHSVarID();
    lhs = load→getLHSVarID();
    as[lhs] = as.loadValue(rhs)
```

Abstract Execution on STORESTMT

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


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

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

$\delta_{\overline{\ell}}$	*Interval*	*MemAddress*

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

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

Algorithm 17: Abstract Execution Rule for STORESTMT

```
Function updateStateOnStore(store):
    node = store→getICFGNode();
    as = getAbsStateFromTrace(node);
    rhs = store→getRHSVarID();
    lhs = store→getLHSVarID();
    as.storeValue(lhs, as[rhs])
```

Handling Cycles (Pseudo Code)

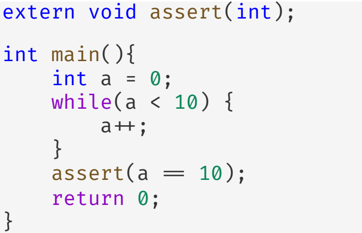
Algorithm 18: Abstract execution guided by WTO (part 2)

```
Function handleCycleWTO(cycle):
    cycle_head := cycle→head()→node() ;
    increasing := true ;
    cur_iter := 0 ;
    while true do
        cur_iter++;
        if $cur\_iter \geq Options :: WidenDelay()$ then
            $prev\_head\_state := postAbsTrace[cycle\_head]$;
            handleICFGNode(cycle→head());
            $cur\_head\_state := postAbsTrace[cycle\_head]$;
            if increasing then
                $postAbsTrace[cycle\_head] := prev\_head\_state.widen(cur\_head\_state)$
                 ;
                if $postAbsTrace[cycle\_head] == prev\_head\_state$ then
                    increasing := false;
                    continue;
            else
                $postAbsTrace[cycle\_head] := prev\_head\_state.narrow(cur\_head\_state)$
                 ;
                if $postAbsTrace[cycle\_head] == prev\_head\_state$ then
                    break ;
        else
            handleICFGNode(cycle→head());
        handleWTOComponents(cycle→getWTOComponents());
```

Algorithm 19: Handle WTO components

```
Function handleCycleBody(cycle):
    for $it = cycle \rightarrow begin(); it! = cycle \rightarrow end(); ++it$ do
        $cur = *it$;
        if $vertex = SVFUtil :: dyn\_cast\langle ICFGWTONode\rangle(cur)$ then
            $handleWTONode(vertex \rightarrow node())$
        else if $cycle2 = SVFUtil :: dyn\_cast\langle ICFGWTOCycle\rangle(cur)$ then
            $handleCycle(cycle2)$
        else
            assert(false&&"unknownWTOtype!")
```

An Example: Abstract Execution

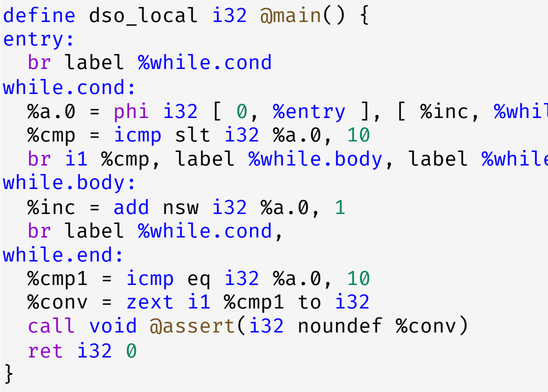


```c
extern void assert(int);

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

Source Code

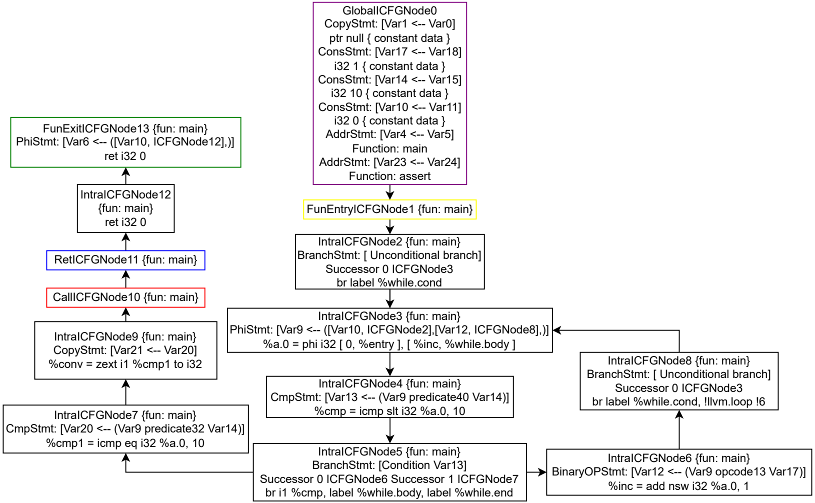
Compile to LLVM IR



```llvm
define dso_local i32 @main() {
entry:
  br label %while.cond
while.cond:
  %a.0 = phi i32 [ 0, %entry ], [ %inc, %whil
  %cmp = icmp slt i32 %a.0, 10
  br i1 %cmp, label %while.body, label %while
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 Execution



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

FunEntryICFGNode1 {fun: main}

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

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

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

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

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

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

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

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

CallICFGNode10 {fun: main}

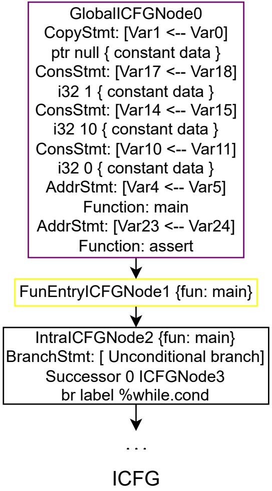
RetICFGNode11 {fun: main}

IntraICFGNode12
{fun: main}
ret i32 0

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

ICFG

An Example: Abstract Execution



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

FunEntryICFGNode1 {fun: main}

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

. . .

ICFG

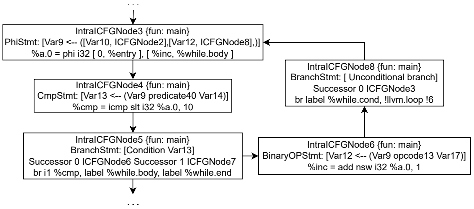
Algorithm 20: Abstract execution guided by WTO

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

$postAbsTrace[\ell_0].varToAbsVal$:

Variable	*Interval*	*MemAddress*
Var0	$\top$	{0x7f00}
Var1	$\top$	{0x7f00}
Var18	[1,1]	$\top$
Var17	[1,1]	$\top$
Var14	[10,10]	$\top$
Var15	[10,10]	$\top$
Var10	[0,0]	$\top$
Var11	[0,0]	$\top$
	. . .	

An Example: Abstract Execution



. . .

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

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

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

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

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

. . .

ICFG

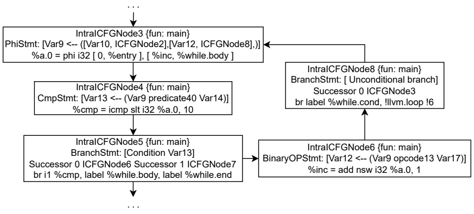
$postAbsTrace[\ell_3].varToAbsVal$:

Variable	*Interval*	*MemAddress*
	. . .	
Var10	[0,0]	⊤
Var9	[0,0]	⊤
	. . .	

Algorithm 12: Abstract execution guided by WTO (part 2)

```
Function handleCycleWTO(cycle):
    cycle_head := cycle→head()→node() ;
    increasing := true ;
    cur_iter := 0 ;
    while true do
        cur_iter++ // cur_iter ≡ 1, Options :: WidenDelay() ≡ 3;
        if cur_iter ≥ Options :: WidenDelay() then
            prev_head_state := postAbsTrace[cycle_head];
            handleICFGNode(cycle→head());
            cur_head_state := postAbsTrace[cycle_head];
            if increasing then
                postAbsTrace[cycle_head] := prev_head_state.widen(cur_head_state) ;
                if postAbsTrace[cycle_head] == prev_head_state then
                    increasing := false;
                    continue;
            else
                postAbsTrace[cycle_head] := prev_head_state.narrow(cur_head_state) ;
                if postAbsTrace[cycle_head] == prev_head_state then
                    break ;
        else
            handleICFGNode(cycle→head());
        handleWTOComponents(cycle→getWTOComponents());
```

An Example: Abstract Execution



. . .

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

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

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

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

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

. . .

ICFG

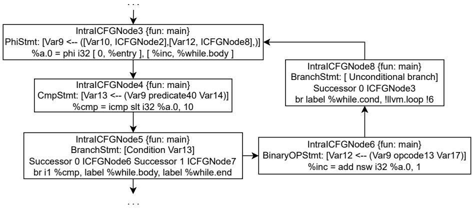
$postAbsTrace[\ell_8].varToAbsVal$:

Variable	*Interval*	*MemAddress*
	. . .	
Var10	[0,0]	$\top$
Var9	[0,0]	$\top$
Var12	[1,1]	$\top$
	. . .	

Algorithm 12: Abstract execution guided by WTO (part 2)

```
Function handleCycleWTO(cycle):
    cycle_head := cycle→head()→node() ;
    increasing := true ;
    cur_iter := 0 // cur_iter ≡ 1, Options :: WidenDelay() ≡ 3;
    while true do
        cur_iter++;
        if cur_iter ≥ Options :: WidenDelay() then
            prev_head_state := postAbsTrace[cycle_head];
            handleICFGNode(cycle→head());
            cur_head_state := postAbsTrace[cycle_head];
            if increasing then
                postAbsTrace[cycle_head] := prev_head_state.widen(cur_head_state) ;
                if postAbsTrace[cycle_head] == prev_head_state then
                    increasing := false;
                    continue;
            else
                postAbsTrace[cycle_head] := prev_head_state.narrow(cur_head_state) ;
                if postAbsTrace[cycle_head] == prev_head_state then
                    break ;
        else
            handleICFGNode(cycle→head());
        handleWTOComponents(cycle→getWTOComponents());
```

An Example: Abstract Execution


. . .
IntraICFGNode3 {fun: main}
PhiStmt: [Var9 <-- ([Var10, ICFGNode2],[Var12, ICFGNode8],)]
%a.0 = phi i32 [0, %entry], [%inc, %while.body]
IntraICFGNode4 {fun: main}
CmpStmt: [Var13 <-- (Var9 predicate40 Var14)]
%cmp = icmp slt i32 %a.0, 10
IntraICFGNode5 {fun: main}
BranchStmt: [Condition Var13]
Successor 0 ICFGNode6 Successor 1 ICFGNode7
br i1 %cmp, label %while.body, label %while.end
IntraICFGNode8 {fun: main}
BranchStmt: [Unconditional branch]
Successor 0 ICFGNode3
br label %while.cond, !llvm.loop !6
IntraICFGNode6 {fun: main}
BinaryOPStmt: [Var12 <-- (Var9 opcode13 Var17)]
%inc = add nsw i32 %a.0, 1
. . .

ICFG

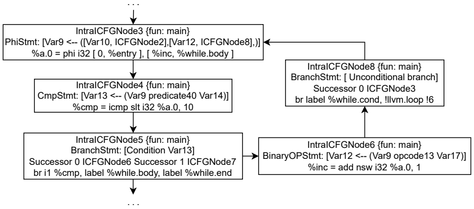
postAbsTrace[ℓ_3].*varToAbsVal* :

Variable	*Interval*	*MemAddress*
	. . .	
Var9	[0,1]	⊤
Var12	[1,1]	⊤
	. . .	

Algorithm 12: Abstract execution guided by WTO (part 2)

```
Function handleCycleWTO(cycle):
    cycle_head := cycle→head()→node() ;
    increasing := true ;
    cur_iter := 0 ;
    while true do
        cur_iter++ // cur_iter ≡ 2, Options :: WidenDelay() ≡ 3;
        if cur_iter ≥ Options :: WidenDelay() then
            prev_head_state := postAbsTrace[cycle_head];
            handleICFGNode(cycle→head());
            cur_head_state := postAbsTrace[cycle_head];
            if increasing then
                postAbsTrace[cycle_head] := prev_head_state.widen(cur_head_state) ;
                if postAbsTrace[cycle_head] == prev_head_state then
                    increasing := false;
                    continue;
            else
                postAbsTrace[cycle_head] := prev_head_state.narrow(cur_head_state) ;
                if postAbsTrace[cycle_head] == prev_head_state then
                    break ;
        else
            handleICFGNode(cycle→head());
        handleWTOComponents(cycle→getWTOComponents());
```

An Example: Abstract Execution



...

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

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

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

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

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

...

ICFG

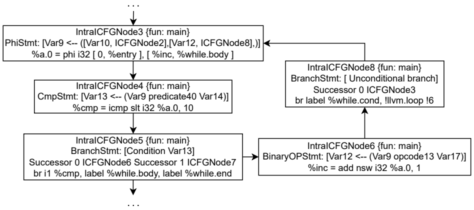
$postAbsTrace[\ell_8].varToAbsVal$:

Variable	*Interval*	*MemAddress*
	...	
Var9	[0,1]	$\top$
Var12	[1,2]	$\top$
	...	

Algorithm 12: Abstract execution guided by WTO (part 2)

```
Function handleCycleWTO(cycle):
    cycle_head := cycle→head()→node() ;
    increasing := true ;
    cur_iter := 0 ;
    while true do
        cur_iter++ // cur_iter ≡ 2, Options :: WidenDelay() ≡ 3;
        if cur_iter ≥ Options :: WidenDelay() then
            prev_head_state := postAbsTrace[cycle_head];
            handleICFGNode(cycle→head());
            cur_head_state := postAbsTrace[cycle_head];
            if increasing then
                postAbsTrace[cycle_head] := prev_head_state.widen(cur_head_state) ;
                if postAbsTrace[cycle_head] == prev_head_state then
                    increasing := false;
                    continue;
            else
                postAbsTrace[cycle_head] := prev_head_state.narrow(cur_head_state) ;
                if postAbsTrace[cycle_head] == prev_head_state then
                    break ;
        else
            handleICFGNode(cycle→head());
        handleWTOComponents(cycle→getWTOComponents());
```

An Example: Abstract Execution



...

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

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

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

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

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

...

ICFG

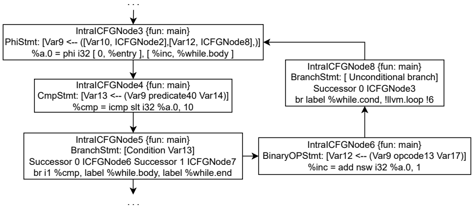
$postAbsTrace[\ell_3].varToAbsVal$:

Variable	*Interval*	*MemAddress*
	...	
Var9	$[0, +\infty]$	$\top$
Var12	[1,2]	$\top$
	...	

Algorithm 12: Abstract execution guided by WTO (part 2)

```
Function handleCycleWTO(cycle):
    cycle_head := cycle→head()→node() ;
    increasing := true ;
    cur_iter := 0 ;
    while true do
        cur_iter++ // cur_iter ≡ 3,  Options :: WidenDelay ≡ 3;
        if cur_iter ≥ Options :: WidenDelay() then
            prev_head_state := postAbsTrace[cycle_head];
            handleICFGNode(cycle→head());
            cur_head_state := postAbsTrace[cycle_head];
            if increasing then
                postAbsTrace[cycle_head] := prev_head_state.widen(cur_head_state) ;
                if postAbsTrace[cycle_head] == prev_head_state then
                    increasing := false;
                    continue;
            else
                postAbsTrace[cycle_head] := prev_head_state.narrow(cur_head_state) ;
                if postAbsTrace[cycle_head] == prev_head_state then
                    break ;
        else
            handleICFGNode(cycle→head());
        handleWTOComponents(cycle→getWTOComponents());
```

An Example: Abstract Execution



. . .

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

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

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

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

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

. . .

ICFG

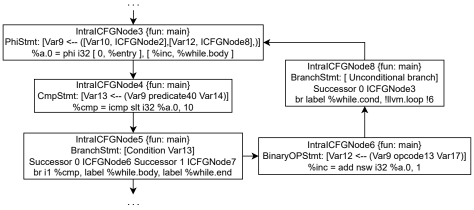
$postAbsTrace[\ell_8].varToAbsVal$:

Variable	*Interval*	*MemAddress*
	. . .	
Var9	[0,9]	⊤
Var12	[1,10]	⊤
	. . .	

Algorithm 12: Abstract execution guided by WTO (part 2)

```
Function handleCycleWTO(cycle):
    cycle_head := cycle→head()→node() ;
    increasing := true ;
    cur_iter := 0 ;
    while true do
        cur_iter++;
        if cur_iter ≥ Options :: WidenDelay() then
            prev_head_state := postAbsTrace[cycle_head];
            handleICFGNode(cycle→head());
            cur_head_state := postAbsTrace[cycle_head];
            if increasing then
                postAbsTrace[cycle_head] := prev_head_state.widen(cur_head_state) ;
                if postAbsTrace[cycle_head] == prev_head_state then
                    increasing := false;
                    continue;
            else
                postAbsTrace[cycle_head] := prev_head_state.narrow(cur_head_state) ;
                if postAbsTrace[cycle_head] == prev_head_state then
                    break ;
        else
            handleICFGNode(cycle→head());
        handleWTOComponents(cycle→getWTOComponents()) // cur_iter ≡ 3;
```

An Example: Abstract Execution



. . .
IntraICFGNode3 {fun: main}
PhiStmt: [Var9 <-- ([Var10, ICFGNode2],[Var12, ICFGNode8],)]
%a.0 = phi i32 [0, %entry], [%inc, %while.body]
IntraICFGNode4 {fun: main}
CmpStmt: [Var13 <-- (Var9 predicate40 Var14)]
%cmp = icmp slt i32 %a.0, 10
IntraICFGNode5 {fun: main}
BranchStmt: [Condition Var13]
Successor 0 ICFGNode6 Successor 1 ICFGNode7
br i1 %cmp, label %while.body, label %while.end
IntraICFGNode6 {fun: main}
BinaryOPStmt: [Var12 <-- (Var9 opcode13 Var17)]
%inc = add nsw i32 %a.0, 1
IntraICFGNode8 {fun: main}
BranchStmt: [Unconditional branch]
Successor 0 ICFGNode3
br label %while.cond, !llvm.loop !6
. . .

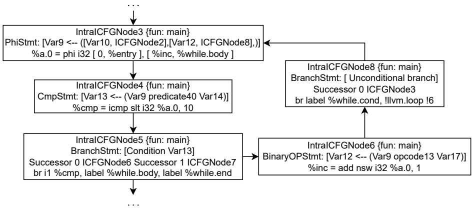
ICFG

$postAbsTrace[\ell_3].varToAbsVal$:

Variable	*Interval*	*MemAddress*
	. . .	
Var9	$[0, +\infty]$	$\top$
Var12	[1,10]	$\top$
	. . .	

```
Algorithm 12: Abstract execution guided by WTO (part 2)
Function handleCycleWTO(cycle):
    cycle_head := cycle→head()→node() ;
    increasing := true ;
    cur_iter := 0;
    while true do
        cur_iter++ // cur_iter ≡ 4;
        if cur_iter ≥ Options :: WidenDelay() then
            prev_head_state := postAbsTrace[cycle_head];
            handleICFGNode(cycle→head());
            cur_head_state := postAbsTrace[cycle_head];
            if increasing then
                postAbsTrace[cycle_head] := prev_head_state.widen(cur_head_state) ;
                if postAbsTrace[cycle_head] == prev_head_state then
                    increasing := false;
                    continue;
            else
                postAbsTrace[cycle_head] := prev_head_state.narrow(cur_head_state) ;
                if postAbsTrace[cycle_head] == prev_head_state then
                    break ;
        else
            handleICFGNode(cycle→head());
        handleWTOComponents(cycle→getWTOComponents());
```

An Example: Abstract Execution


...

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

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

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

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

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

...

ICFG

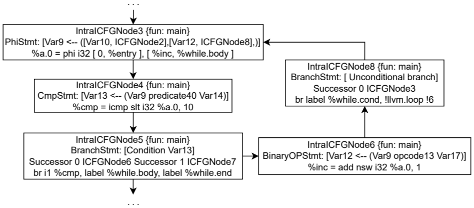
$postAbsTrace[\ell_3].varToAbsVal$:

Variable	*Interval*	*MemAddress*
	...	
Var9	$[0, 10]$	$\top$
Var12	[1,10]	$\top$
	...	

Algorithm 12: Abstract execution guided by WTO (part 2)

```
Function handleCycleWTO(cycle):
    cycle_head := cycle→head()→node() ;
    increasing := true ;
    cur_iter := 0 ;
    while true do
        cur_iter++ // cur_iter ≡ 5;
        if cur_iter ≥ Options :: WidenDelay() then
            prev_head_state := postAbsTrace[cycle_head];
            handleICFGNode(cycle→head()) // increasing ≡ false;
            cur_head_state := postAbsTrace[cycle_head];
            if increasing then
                postAbsTrace[cycle_head] := prev_head_state.widen(cur_head_state) ;
                if postAbsTrace[cycle_head] == prev_head_state then
                    increasing := false;
                    continue;
            else
                postAbsTrace[cycle_head] := prev_head_state.narrow(cur_head_state) ;
                if postAbsTrace[cycle_head] == prev_head_state then
                    break ;
        else
            handleICFGNode(cycle→head());
        handleWTOComponents(cycle→getWTOComponents());
```

An Example: Abstract Execution



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

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

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

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

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

ICFG

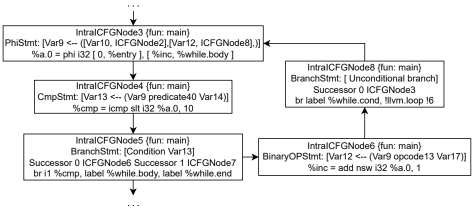
$postAbsTrace[\ell_8].varToAbsVal$:

Variable	*Interval*	*MemAddress*
	...	
Var9	[0,9]	$\top$
Var12	[1,10]	$\top$
	...	

Algorithm 12: Abstract execution guided by WTO (part 2)

```
Function handleCycleWTO(cycle):
    cycle_head := cycle→head()→node() ;
    increasing := true ;
    cur_iter := 0 ;
    while true do
        cur_iter++;
        if $cur\_iter \geq Options :: WidenDelay()$ then
            $prev\_head\_state := postAbsTrace[cycle\_head]$;
            handleICFGNode(cycle→head());
            $cur\_head\_state := postAbsTrace[cycle\_head]$;
            if increasing then
                $postAbsTrace[cycle\_head] := prev\_head\_state.widen(cur\_head\_state)$ ;
                if $postAbsTrace[cycle\_head] == prev\_head\_state$ then
                    increasing := false;
                    continue;
            else
                $postAbsTrace[cycle\_head] := prev\_head\_state.narrow(cur\_head\_state)$ ;
                if $postAbsTrace[cycle\_head] == prev\_head\_state$ then
                    break ;
        else
            handleICFGNode(cycle→head());
        handleWTOComponents(cycle→getWTOComponents()) // $cur\_iter \equiv 5$;
```

An Example: Abstract Execution



...

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

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

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

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

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

...

ICFG

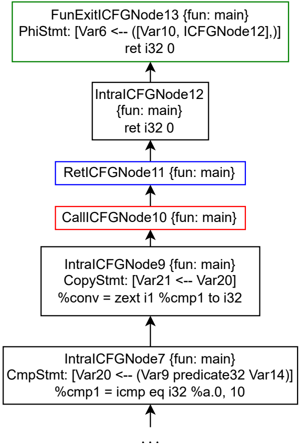
$postAbsTrace[\ell_3].varToAbsVal$:

Variable	Interval	MemAddress
	…	
Var9	[0, 10]	⊤
Var12	[1,10]	⊤
	…	

Algorithm 12: Abstract execution guided by WTO (part 2)

```
Function handleCycleWTO(cycle):
    cycle_head := cycle→head()→node() ;
    increasing := true ;
    cur_iter := 0 ;
    while true do
        cur_iter++ // cur_iter ≡ 6;
        if cur_iter ≥ Options :: WidenDelay() then
            prev_head_state := postAbsTrace[cycle_head];
            handleICFGNode(cycle→head()) // increasing ≡ false;
            cur_head_state := postAbsTrace[cycle_head];
            if increasing then
                postAbsTrace[cycle_head] := prev_head_state.widen(cur_head_state) ;
                if postAbsTrace[cycle_head] == prev_head_state then
                    increasing := false;
                    continue;
            else
                postAbsTrace[cycle_head] := prev_head_state.narrow(cur_head_state) ;
                if postAbsTrace[cycle_head] == prev_head_state then
                    break ;
        else
            handleICFGNode(cycle→head());
        handleWTOComponents(cycle→getWTOComponents());
```

An Example: Abstract Execution


FunExitICFGNode13 {fun: main}
PhiStmt: [Var6 <-- ([Var10, ICFGNode12],)]
ret i32 0
IntraICFGNode12
{fun: main}
ret i32 0
RetICFGNode11 {fun: main}
CallICFGNode10 {fun: main}
IntraICFGNode9 {fun: main}
CopyStmt: [Var21 <-- Var20]
%conv = zext i1 %cmp1 to i32
IntraICFGNode7 {fun: main}
CmpStmt: [Var20 <-- (Var9 predicate32 Var14)]
%cmp1 = icmp eq i32 %a.0, 10
. . .

ICFG

Algorithm 13: Abstract execution guided by WTO

```
Function handleStatement($\ell$):
    $tmpAS := preAbsTrace[\ell]$;
    if $\ell$ is BinaryStmt then
        updateStateOnBinary($\ell$);
    ...;
```

$postAbsTrace[\ell_7].varToAbsVal$:

Variable	*Interval*	*MemAddress*
	. . .	
Var9	[10,10]	$\top$
Var20	[1,1]	$\top$
	. . .	