EDAP15: Program Analysis

DATAFLOW ANALYSIS 3

Christoph Reichenbach

Welcome back!

Some Administrativa:

- Quiz deadlines: some slack if you missed a deadline
  - 8 days buffer (cumulative across all quizzes), not counting weekends
- Automatic feedback with “hidden tests” for lab 1 online now
- Let us know if htis helps; may offer for lab 3, too

Questions?

Lecture Overview

Foundations

Static Analysis

Dynamic Analysis

Properties

Control Flow

01 Foundations

03 04 Types

12 Instrumentation

02 Constructing Program Analyses in JastAdd

05 06 07 **Data Flow**

05 Intraprocedural

13 Analysis

08 09 Memory

10 Interprocedural

11 Indirect

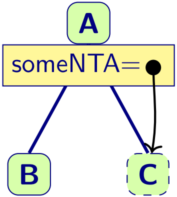
14 Review

Non-Terminal Attributes

JastAdd

```
syn nta C AnyNode.someNTA() = new C(this);
```

- AST node as attribute
- Reifying implicit constructs (making them explicit in AST)
  - Built-in types
  - Built-in functions, constants
  - "Null Objects", handle missing declarations (⇒ EDAN65)



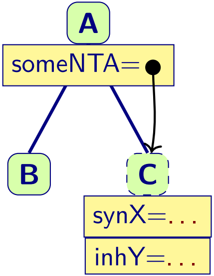
Beware

- NTAs must be **fresh** objects
- AST will be inconsistent if you re-use nodes
  JastAdd does not check for this!

Non-Terminal Attributes

JastAdd

```
syn nta C AnyNode.someNTA() = new C(this);
```



- NTA may have attributes
- Owner node must provide inherited attributes
  `A.someNTA().inhY() = ...`

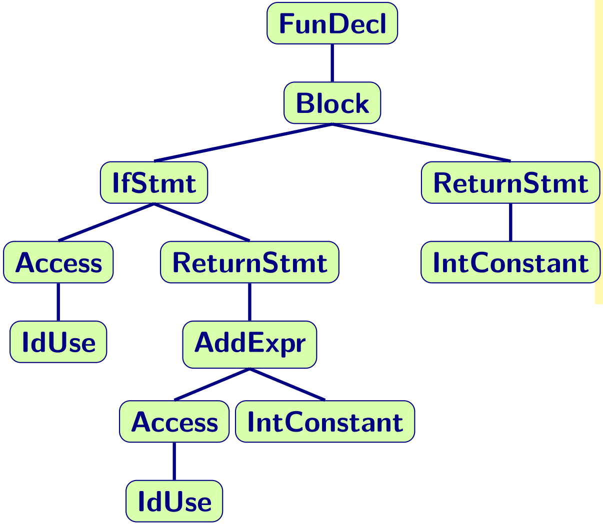
Summary

- Nonterminal Attributes (NTAs):
  - "Synthetic" AST node
  - Useful e.g. for CFG nodes that have no AST equivalent
  - Need to be *fresh*
  - Need to be owned by exactly one parent
- Function like normal AST nodes
  - Can define / inherit attributes
  - Can participate in collection attributes

Building CFGs

- **1.**CFG separate from AST
  - AST $\Rightarrow$ CFG translation
  - Often simplified
  - *Advantages*:
    - Fewer CFG node types
    - Analyses can communicate results by transforming CFG (Remove unreachable CFG nodes etc.)
    - *Common in compiler mid-ends/back-ends*
- **2. CFG is part of AST**
  - *Some* AST nodes are also CFG nodes
  - *Advantages*:
    - No translation needed
    - *Common in compiler front-ends and IDEs*
    - Teal: Uses JastAdd's IntraCFG framework

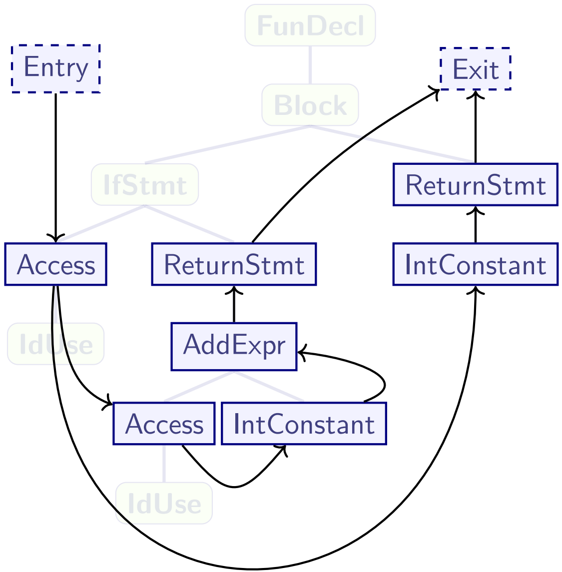
CFGs on the AST



Teal

```
fun f(x) = {
  if x {
    return x + 1;
  }
  return 0;
}
```

CFGs on the AST



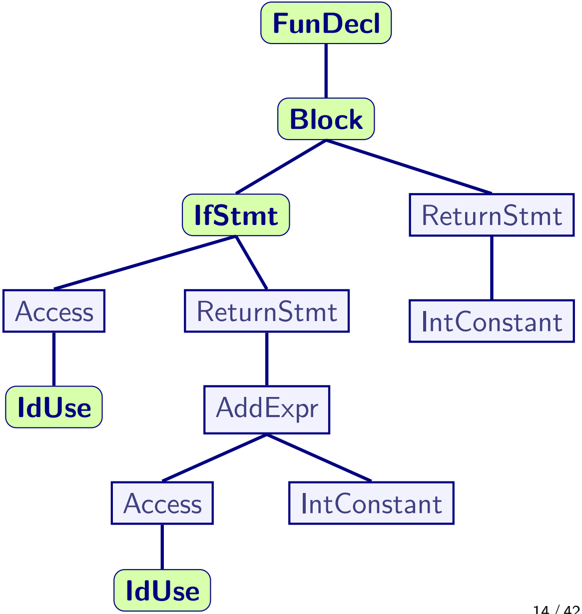
- Some ASTNodes are CFGNodes
  - For example, Teal:
    - All **Expr**, some **Stmt**
    - Special NTAs: **Entry**, **Exit**,...
- `CFGNode.succ()` attribute for CFG successors: $\boxed{\text{N1}} \longrightarrow \boxed{\text{N2}}$

Teal

```
fun f(x) = {
  if x {
    return x + 1;
  }
  return 0;
}
```

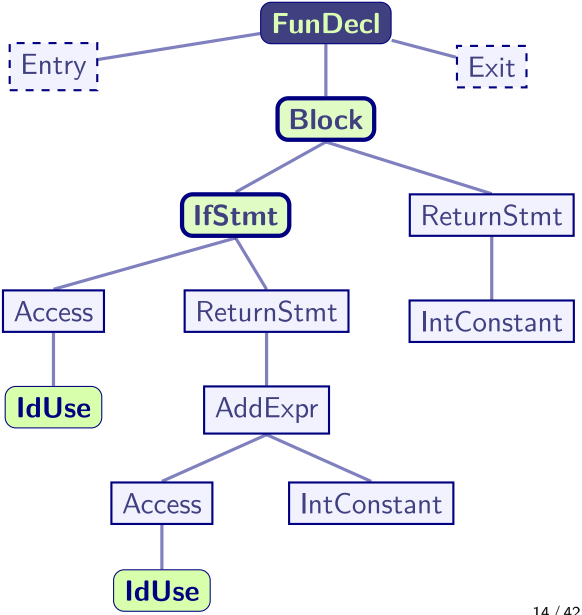
Constructing CFGs on the AST (1/2)

- Categorise AST nodes by role in CFG
  - $\boxed{\text{CFGNode}}$: part of CFG



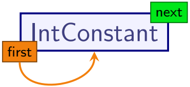
Constructing CFGs on the AST (1/2)

- Categorise AST nodes by role in CFG
  - CFGNode: part of CFG
  - **CFGRoot**: start/end of CFG with Entry / Exit NTAs (e.g., FunDecl)
  - **CFGSupport**: not part of CFG but influence CFG edges

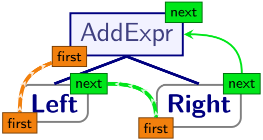


Constructing CFGs on the AST (2/2)

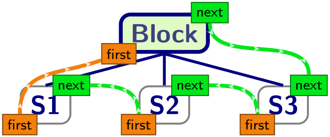
```
interface CFGNode extends CFGSupport;
→ syn Set CFGSupport.firstNodes()
→ inh Set CFGSupport.nextNodes()
```



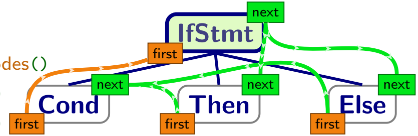

```
AddExpr.firstNodes() = getLeft().firstNodes()
AddExpr.getLeft().nextNodes() = getRight().firstNodes()
AddExpr.getRight().nextNodes() = new Set({this})
```




```
BlockStmt.firstNodes() = getStmt(0).firstNodes()
BlockStmt.getStmt(i).nextNodes() =
         if (i<size):  getStmt(i+1).firstNodes()
         else:  this.nextNodes()
```




```
IfStmt.firstNodes() = getCond().firstNodes()
IfStmt.getCond().nextNodes() =
       getThen().firstNodes() ∪ getElse().firstNodes()
IfStmt.getThen().nextNodes() = this.nextNodes()
IfStmt.getElse().nextNodes() = this.nextNodes()
```



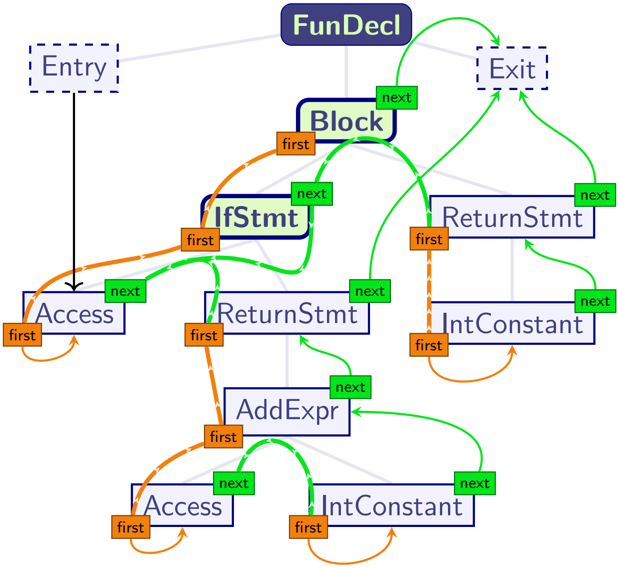
Constructing CFGs on the AST (1/2)

- Categorise AST nodes by role in CFG
  - CFGNode: part of CFG
  - **CFGRoot**: start/end of CFG with Entry / Exit NTAs (e.g., FunDecl)
  - **CFGSupport**: not part of CFG but influence CFG edges
  - **ASTNode**s: can ignore
- Construct edges
  - For each subtree: first CFGNodes in subtree?
  - For each CFGNode: next CFGNodes after self?

⟶ `succ()`

⟶ `firstNodes()`

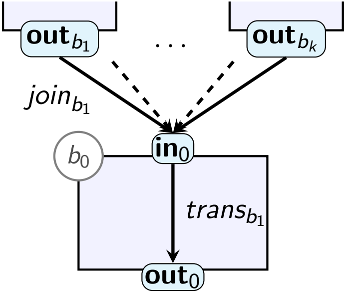
⟶ `nextNodes()`



Summary

- CFG can be separate or overlaid on AST
- Teal uses an overlay CFG
- CFGNodes are:
  - ASTNodes that participate in CFG
  - Some NTAs:
    - **Entry**: Subprogram start
    - **Exit**: Subprogram end
    - Others can be useful e.g. for exception handling
- Constructing CFG with IntraCFG:
  - firstNodes: For this *subtree*, which CFGNodes execute first? *synthesised attribute*
  - nextNodes: For this *CFGNode*, which CFGNodes execute next? *inherited attribute*

Implementing Data Flow Analysis



JastAdd

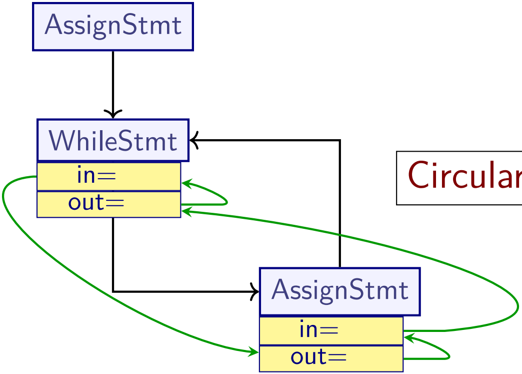
```
syn Lattice CFGNode.in() {
  Lattice r = ⊥;
  for (CFGNode b: pred()) {
    r = r ⊔ b.out();
  }
  return r;
}
syn Lattice CFGNode.out() {
  return trans(in());
}
```

JastAdd

```
// Default: trans() is no-op
syn Lattice CFGNode.trans(Lattice v) = v;

// Specialised transfer function
syn Lattice AssignStmt.trans(Lattice v) = ...
```

Fixpoints and Reference Attributes



Circular Dependency

This solution is *not well-defined*

Fixpoints from Circular Attributes

JastAdd

```
syn Lattice CFGNode.out() circular [Lattice.bottom()];
```

Lattice $\qquad\qquad\qquad\qquad \perp$

- **Circular Attributes**
  - JastAdd allows circular dependency *if explicitly declared*
  - `CFGNode.out()` can now recursively call itself

$$\begin{array}{lll} v_1 = \perp & = \texttt{CFGNode.out()} & \text{at iteration 1} \\ v_2 & = \texttt{CFGNode.out()} & \text{at iteration 2} \\ & \ldots & \\ v_k & = \texttt{CFGNode.out()} & \text{at iteration k} \end{array}$$

  - Iterates until v_{k-1}`.equals(`v_k`)`

Beware

- Your lattice must have a correct `equals()` method
- You must be in a monotone framework

JastAdd can ***not*** *check for this!*

Implementation Strategy

- Definitions for analysis a on lattice $\mathcal{L}$:

Attribute	Forward	Backward
$\mathcal{L}$ a`In()`	$\bigsqcup\{p.a\texttt{Out()} \mid p \in \textsf{pred}()\}$	a`Transfer(`a`Out())`
$\mathcal{L}$ a`Transfer(`$\mathcal{L}$`)`	transfer fn	transfer fn
$\mathcal{L}$ a`Out()`:	a`Transfer(`a`In())`	$\bigsqcup\{p.a\texttt{In()} \mid p \in \textsf{succ}()\}$

- Not necessary to declare all attributes as circular
  - Only one attribute in each cycle need be circular

. . . but can declare circular if unsure

Combining with Expressions

- Data flow analysis resolves *flow-sensitive* information
- May still combine with simple synthesised attributes
  Example: Constant propagation / folding
  - computing `Expr.constantValue()`
  - `Access.constantValue()` $\Leftarrow$ `constantOut()`
  - but `AddExpr.constantValue()` will not typically use `constantOut()` / `constantIn()` directly
  - uses `getLeft().constantValue()` / `getRight().constantValue()`

Summary

- Attributes that depend on themselves:
  Usually $\implies$ *AG not well-defined*
- **Circular** attributes are exception
  - JastAdd suppresses recursion check
  - Repeated evaluation
  - Evaluation stops once current result `.equals()` last result
- It is up to attribute definition to guarantee termination!
  - Monotone framework
  - *Finite lattice height*
    - (Or *widening*, later today)

Dimensions of Data Flow

- Data Flow analysis is versatile:
- Can adjusting:
  - Lattice and transfer functions
  - Data representation
  - Specialised lattices
  - CFG history awareness
  - Analysis integration (later!)
  - Treatment of subroutine calls (later!)

Numerical Domains

Teal

```
// valid index range:  [0, 2]
var a := [1, 2, 3];
var i := 0;
var result = 0;
while i <= 3 {
  result += a[i];
  i := i + 1;
}
```

- Bug: `i` may be 3, and out of bounds for `a`
- Analysis: Compute bounding intervals $[min, max]$
  - **Interval Abstract Domain**
- $[\![\texttt{i}]\!] = [0, 3]$

Numerical Domains

Teal

```
var a := [1, 2, 3];
var i := 0;
var ⟦i⟧ = [0,2] ew array[int](3);
while i < 3 {
  var j := 0;
  var c ⟦j⟧ = [0,2]
  while j < 3 - i {
    c := c + a[i + j];
           ⟦i + j⟧ = [0,4]
    j := j + 1;
  }
  result[i] := c;
  i := i + 1;
}
```

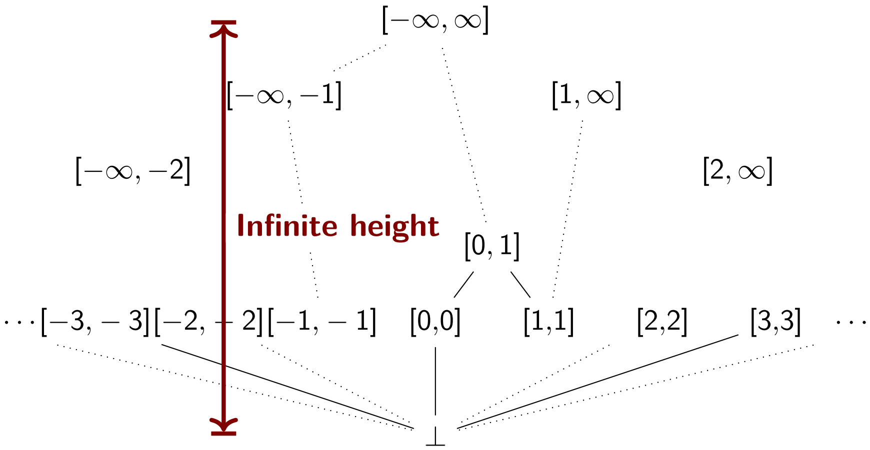
Out of bounds?

- Guarantee: $j < 3 - i$
  $\implies j + i < 3$
- Array access is safe!
- Analysis must capture relations between variables
  - **Octagon Abstract Domain**

Numerical Domains

- **Interval Abstract Domain**
  - Constraints: $x \in [min_x, max_x]$
- **Octagon Abstract Domain**
  - Constraints: $\pm x \pm y \leq c$
  - (x, y variables, c constant number)
- **Polyhedra Abstract Domain**
  - $c_1x_1 + c_2x_2 + \ldots + c_nx_n \leq c_0$
  - $c_1x_1 + c_2x_2 + \ldots + c_nx_n = c_0$
- Increasingly powerful, increasingly expensive to analyse

Interval Domain

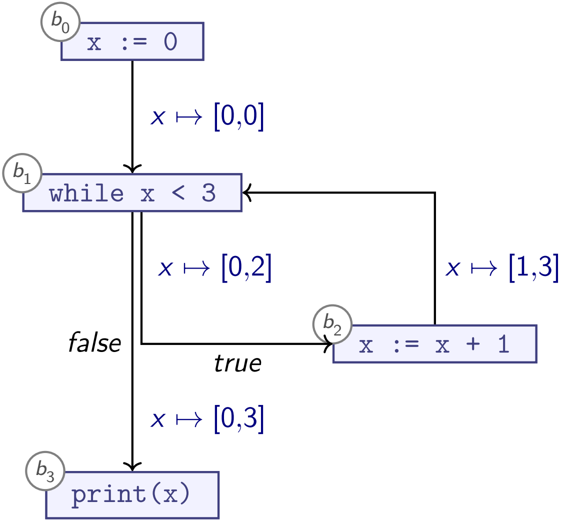


- $\top = [-\infty, \infty]$
- $[l_1, r_1] \sqcup [l_2, r_2] = [min(l_1, l_2), max(r_1, r_2)]$

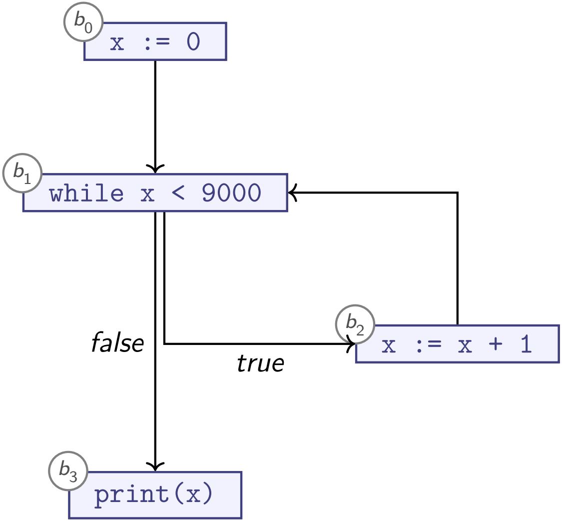
Summary

- Numerical Abstract Domains capture linear relations between variables and constants
  - **Interval Abstract Domain**: $x \in [min_x, max_x]$
  - Octagon Abstract Domain: $\pm x \pm y \leq c$
  - Polyhedra Abstract Domain: Arbitrary linear relationships
- Infinite Domain height: No termination guarantee with our current tools

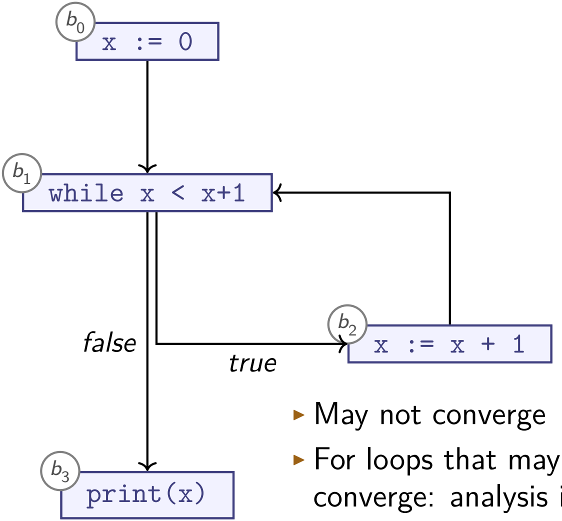
Applying the Interval Domain



Applying the Interval Domain

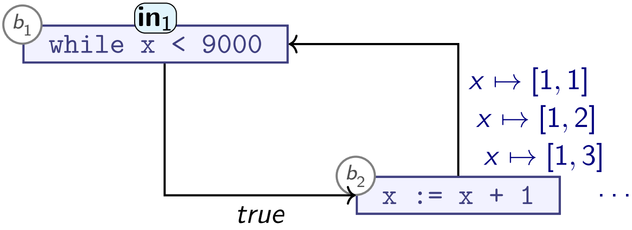


Applying the Interval Domain



- May not converge
- For loops that may take long to converge: analysis is slow

Widening



- Inefficient: no reason to assume 2, 3, ... will help us converge
- Detection: when updating **in**$_1$:
  - Check if we have converged
  - Otherwise, **widen**

$$v_1 \nabla v_2 = \begin{cases} v_1 & \iff \quad v_1 = v_2 \\ \textbf{widen}(v_1 \sqcup v_2) & \iff \quad v_1 \neq v_2 \end{cases}$$

  - For a suitable **widen** function

Widening Functions

- For convergence: satisfy Ascending Chain Condition on:

$$v_{i+1} = \mathbf{widen}(v_i)$$

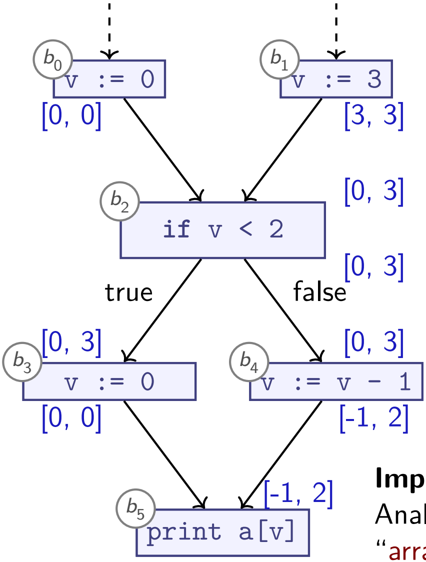
- Suitable functions for Interval Domain?
  - $\mathbf{widen}_{\top}(v) = \top$
    - Very conservative
    - Ensures convergence
  - $\mathbf{widen}_{10000}([l,r]) = [l - 10000, r + 10000]$
    - *No convergence*: still allows infinite ascending chain
  - $\mathbf{widen}_{\mathcal{K}}([l,r]) = [max(\{v \in \mathcal{K} | v < l\}), min(\{v \in \mathcal{K} | v > r\})]$
    - Ensures convergence *iff* $\mathcal{K}$ *is finite*
    - Must pick "good" $\mathcal{K}$
    - Common strategy:
      $\mathcal{K} = \{-\infty, \infty\} \cup$ all numeric literals in program
      Our example: $\mathcal{K} = \{-\infty, 0, 1, 9000, \infty\}$

```
var x := 0;
while x < 9000 {
    x := x + 1;
}
```

Summary

- **Widening** allows us to use infinite domains $\mathcal{L}$
- Use **widen** function
  - **widen** must satisfy Ascending Chain Condition on $\mathcal{L}$
  - **widen**($\mathcal{L}$) generates finite lattice
- Widening operator ∇ applies **widen** function iff needed
- Approach:
  1. Before analysis runs: we design analysis on infinite-height lattice
  2. When analysis runs on concrete program:
     - **widen** constructs finite-height lattice specific to program
     - ∇ applies **widen** on demand
       MFP: When updating: $\mathbf{in}_i := \mathbf{in}_i \nabla \mathbf{out}_j$

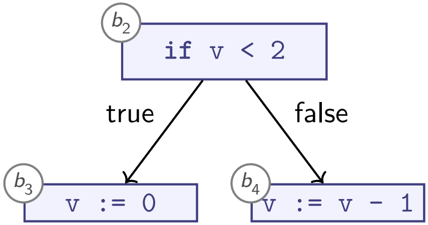
Conditionals



Imprecision can yield false positive
Analysis concludes:
"array index may be negative"

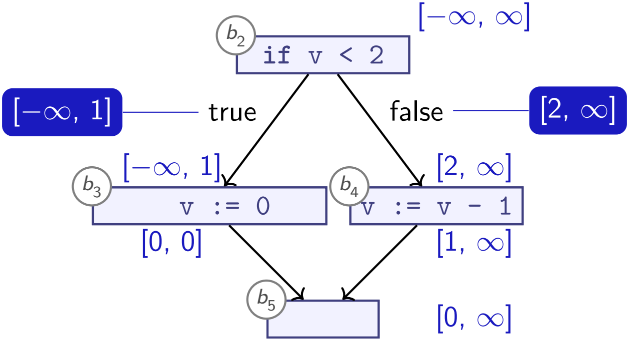
Handling Conditionals (1/2)

- So far: Did not make use of conditional predicate
  - `true` branch: only if `v < 2`
    `v` $\in$ `[-∞, 1]`
  - `false` branch: only if $\neg$`(v < 2)`
    `v` $\in$ `[2, ∞]`



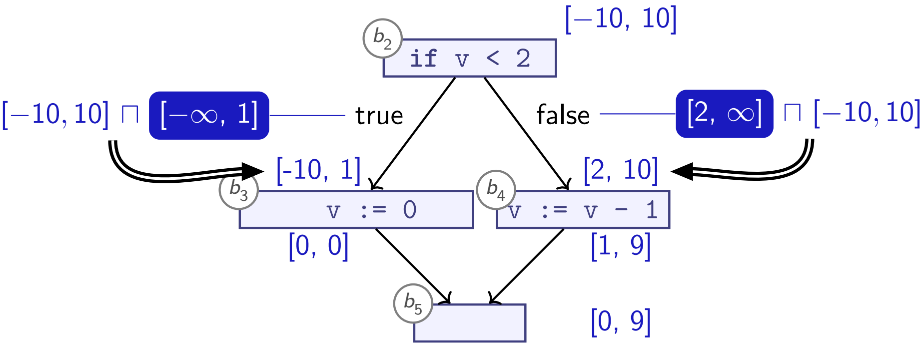
- **Control Sensitive** analysis utilises this information
  - Filter possible values

Handling Conditionals (2/2)



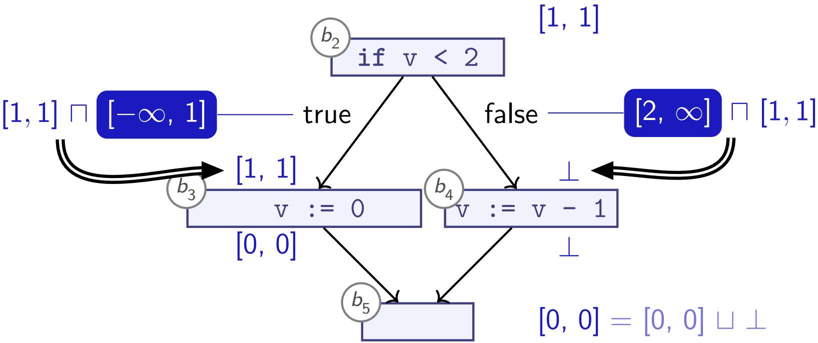
- *Idea*: Split interval for v for true/false branches
  - Analyse each branch with part of original interval
  - "Re-assemble" interval on join afterwards

Handling Conditionals (2/2)



- *Idea*: Split interval for v for true/false branches
  - Analyse each branch with part of original interval
  - "Re-assemble" interval on join afterwards
- If not $v \mapsto \top$:
  - **Filter** with lattice *meet*: $\sqcap$

Contradictions



Summary

- **Control sensitive** analysis considers conditionals:
  - May propagate different information along different edges:
    - **if** P:
    - Special transfer function for '**assert** P' on 'true' edge
    - Special transfer function for '**assert** `not` P' on 'false' edge
- More precise than **control insensitive** analysis
- Utilises *Lattice Meet* operation $\sqcap$

  Intuition:
  $a \sqcap b$ "satisfy a **and** b"
  $a \sqcup b$ "satisfy a **or** b"
- $a \sqcap b = \perp$ can happen: *branch will never execute*

Lecture Overview

Foundations

Static Analysis

Dynamic Analysis

Properties

Control Flow

01 Foundations

02 Constructing Program Analyses in JastAdd

03 04 Types

05 06 07 **Data Flow**

08 09 Memory

05 Intraprocedural

10 Interprocedural

11 Indirect

12 Instrumentation

13 Analysis

14 Review

Summary and Outlook

- Summary:
  - Non-Terminal Attributes
  - Building CFGs
  - Circular Attributes
  - Control Sensitivity & Path Sensitivity
  - Numerical Domains (esp. the Interval Domain)
  - Widening
- Next up: Analysing the Heap

`http://cs.lth.se/EDAP15`