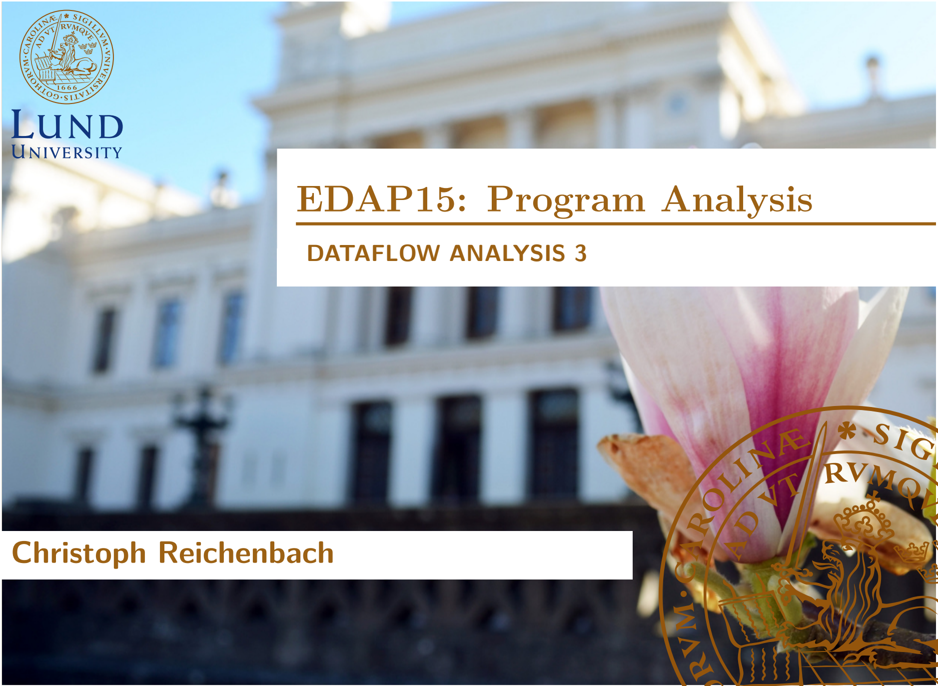


EDAP15: Program Analysis

DATAFLOW ANALYSIS 3

Christoph Reichenbach

Welcome back!

Some Administrativa:

- Extension for Quiz 06 until Tuesday (slides were late!)
- Quiz deadlies: some slack if you missed a deadline
  - 5 days buffer (cumulative across all quizzes), not counting weekends

Questions?

Lecture Overview

Foundations

- 01 Foundations
- 02 Constructing Program Analyses in JastAdd

Static Analysis

Properties

- 03 04 Types
- 05 06 07 **Data Flow**
- 08 09 Memory

Control Flow

- 05 Intraprocedural
- 10 Interprocedural
- 11 Indirect

Dynamic Analysis

- 12 Instrumentation
- 13 Analysis

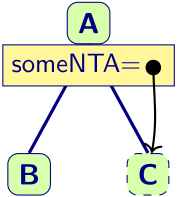
- 14 Review

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)



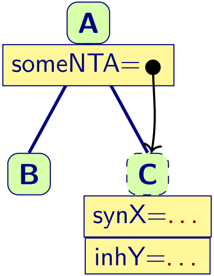
Beware

- 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() = ...`

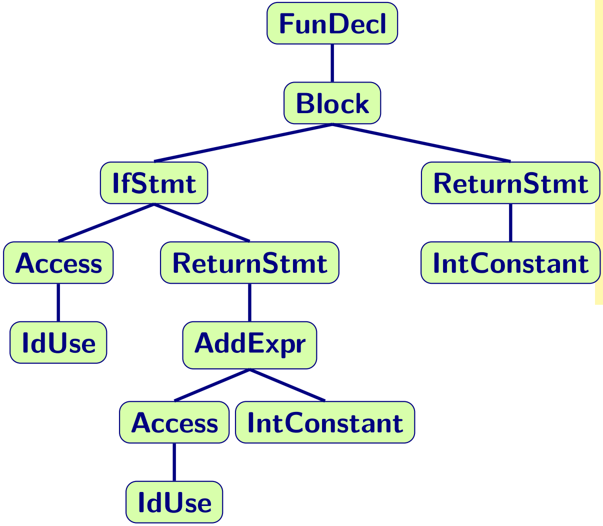
Summary

- 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

- CFG is separate from AST
  - Translate AST into CFG
  - Optionally: simplify representation
  - *Advantages*:
    - Reduce number of node types
    - Analyses can communicate results by transforming CFG (Remove unreachable CFG nodes etc.)
    - *Common in compiler mid-ends/back-ends*
- **CFG is part of AST**
  - Some AST nodes are also CFG nodes
  - *Advantages*:
    - Straightforward error reporting
    - Avoid complexity of translation
    - *Common in compiler front-ends and IDEs*
    - Teal: Uses JastAdd's IntraCFG framework

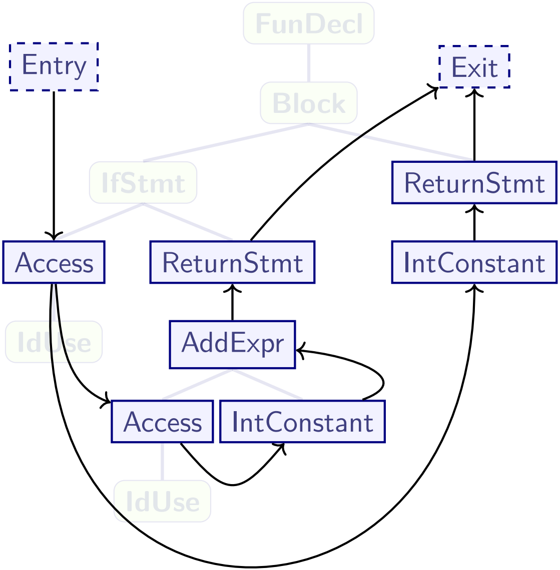
CFGs on the AST



Teal

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

CFGs on the AST



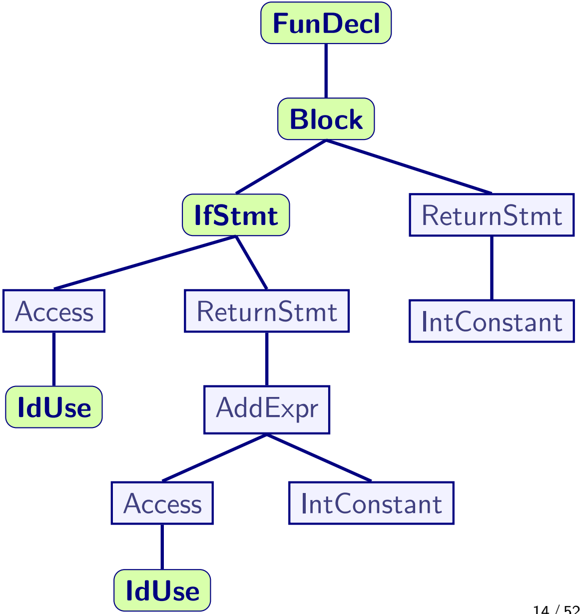
- Some **ASTNode**s are CFGNodes
  - For example, Teal:
    - All **Expr**, some **Stmt**
    - Special NTAs: **Entry**, **Exit**, ...
- `CFGNode.succ()` attribute for CFG successors: N1 ⟶ N2

Teal

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

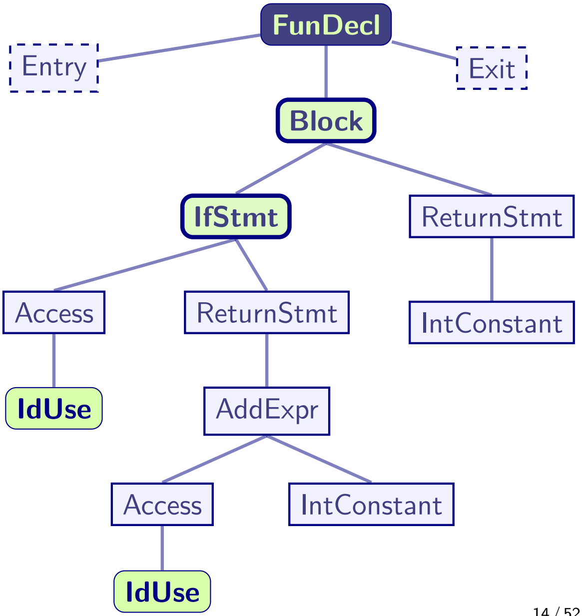
Constructing CFGs on the AST (1/2)

- Categorise AST nodes by role in CFG
  - CFGNode: part of CFG



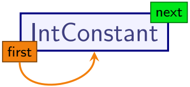
Constructing CFGs on the AST (1/2)

- 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

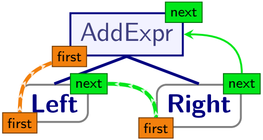


Constructing CFGs on the AST (2/2)

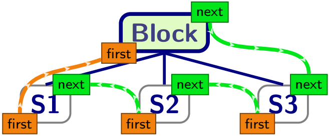
```
interface CFGNode extends CFGSupport;
  syn Set CFGSupport.firstNodes()
  inh Set CFGSupport.nextNodes()
```



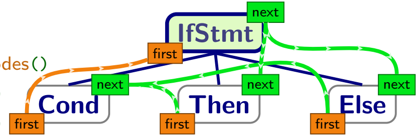

```
AddExpr.firstNodes() = getLeft().firstNodes()
AddExpr.getLeft().nextNodes() = getRight().firstNodes()
AddExpr.getRight().nextNodes() = new Set({this})
```




```
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()
```



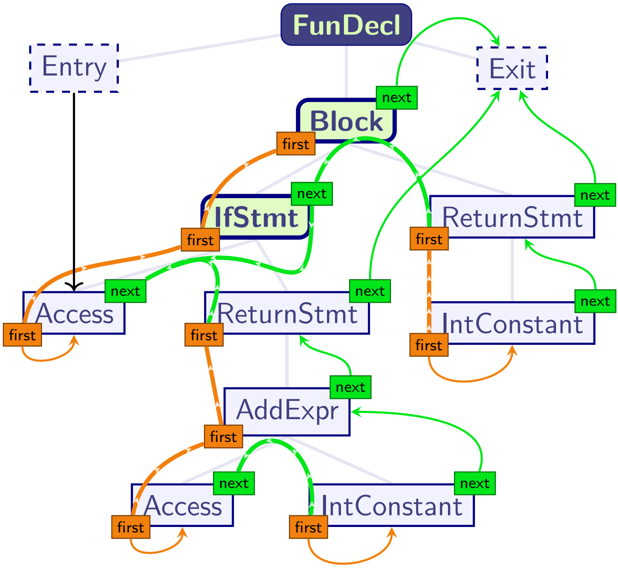
Constructing CFGs on the AST (1/2)

- 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
  - ASTNodes: can ignore
- Construct edges
  - For each subtree: first CFGNodes in subtree?
  - For each CFGNode: next CFGNodes after self?

⟶ `succ()`

⟶ `firstNodes()`

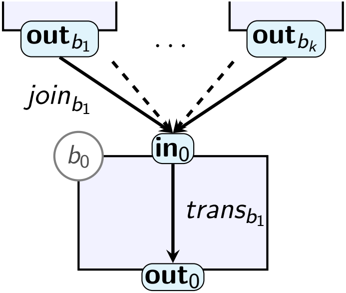
⟶ `nextNodes()`



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

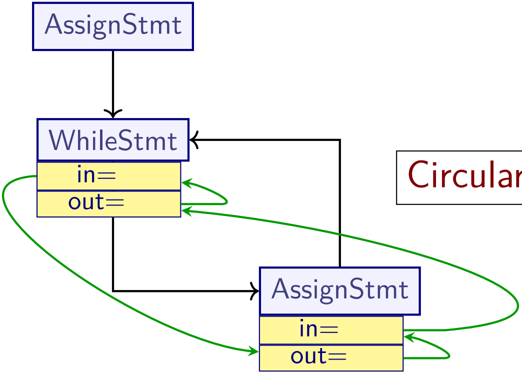
```
syn Lattice CFGNode.in() {
  Lattice r = ⊥;
  for (CFGNode b:  pred()) {
    r = r ⊔ b.out();
  }
  return r;
}
syn Lattice CFGNode.out() {
  return trans(in());
}
```

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 [new Lattice()];
```

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`)`

<u>Beware</u>

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

JastAdd cannot not check for this!

Implementation Strategy

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

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

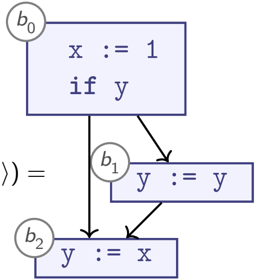
- Not necessary to declare all attributes as circular
  - Circularity allowed as long as one attribute on circle is declared circular

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)

Naïve Iteration Revisited

Analysis on
$\mathbb{Z}_\perp^\top \times \mathbb{Z}_\perp^\top$



$$trans_0(\{x \mapsto v_x,\ y \mapsto v_y\}) = \{x \mapsto \mathbf{1},\ y \mapsto v_y\}$$

$$trans_{all}(\langle \mathbf{in}_0, \mathbf{out}_0, \mathbf{out}_1, \mathbf{out}_2 \rangle) = \left\langle \begin{array}{l} \mathbf{in}_0, \\ trans_0(\mathbf{out}_0), \\ trans_1(\mathbf{out}_1), \\ trans_2(\mathbf{out}_0 \sqcup \mathbf{out}_1) \end{array} \right\rangle$$

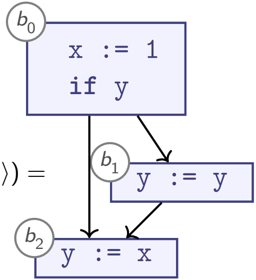
$$trans_1(S) = S$$

$$trans_2(\{x \mapsto \mathbf{v_x},\ y \mapsto v_y\}) = \{x \mapsto v_x,\ y \mapsto \mathbf{v_x}\}$$

	$\mathbf{I}$	$trans_{all}^1(\mathbf{I})$	$trans_{all}^2(\mathbf{I})$	$trans_{all}^3(\mathbf{I})$
$\mathbf{in}_0$	$\perp$	$\perp$	$\perp$	$\perp$
$\mathbf{out}_0$	$\perp$	$x \mapsto 1$	$x \mapsto 1$	$x \mapsto 1$
$\mathbf{out}_1$	$\perp$	$\perp$	$x \mapsto 1$	$x \mapsto 1$
$\mathbf{out}_2$	$\perp$	$\perp$	$x \mapsto 1, y \mapsto 1$	$x \mapsto 1, y \mapsto 1$

Naïve Iteration Revisited

Analysis on
$\mathbb{Z}_\perp^\top \times \mathbb{Z}_\perp^\top$



$$trans_0(\{x \mapsto v_x,\ y \mapsto v_y\}) = \{x \mapsto \mathbf{1},\ y \mapsto v_y\}$$

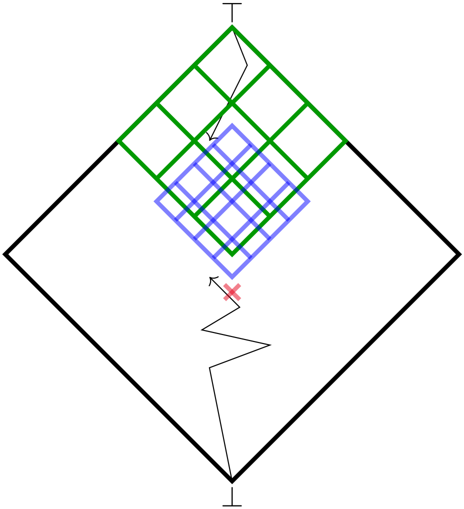
$$trans_{all}(\langle \mathbf{in}_0, \mathbf{out}_0, \mathbf{out}_1, \mathbf{out}_2 \rangle) = \left\langle \begin{array}{l} \mathbf{in}_0, \\ trans_0(\mathbf{out}_0), \\ trans_1(\mathbf{out}_1), \\ trans_2(\mathbf{out}_0 \sqcup \mathbf{out}_1) \end{array} \right\rangle$$

$$trans_1(S) = S$$

$$trans_2(\{x \mapsto \mathbf{v_x},\ y \mapsto v_y\}) = \{x \mapsto v_x,\ y \mapsto \mathbf{v_x}\}$$

	$\mathbf{I}$	$trans_{all}^1(\mathbf{I})$	$trans_{all}^2(\mathbf{I})$	$trans_{all}^3(\mathbf{I})$
$\mathbf{in}_0$	$\top$	$\top$	$\top$	$\top$
$\mathbf{out}_0$	$\top$	$x \mapsto 1, y \mapsto \top$	$x \mapsto 1, y \mapsto \top$	$x \mapsto 1, y \mapsto \top$
$\mathbf{out}_1$	$\top$	$\top$	$x \mapsto 1, y \mapsto \top$	$x \mapsto 1, y \mapsto \top$
$\mathbf{out}_2$	$\top$	$\top$	$\top$	$x \mapsto 1, y \mapsto 1$

Least Fixed Point vs MFP



MFP

Naïve Iteration

MOP

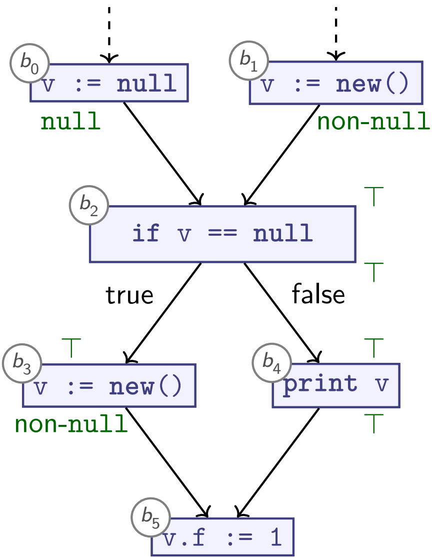
Summary

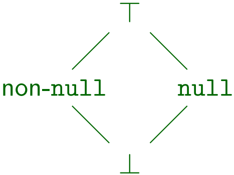
- **MFP**
  - Efficient
  - Fixpoint $\sqsupseteq$ starting point
- **Naïve fixpoint iteration**
  - Fixpoint may be *above* or *below* starting point
- **MOP**
  - One fixpoint, no "starting point"
  - Maximal Precision
  - Undecidable in general
- This list of fixpoint algorithms is not exhaustive
- Different fixpoint lattices per algorithm
- All fixpoints are sound overapproximations

Dimensions of Data Flow

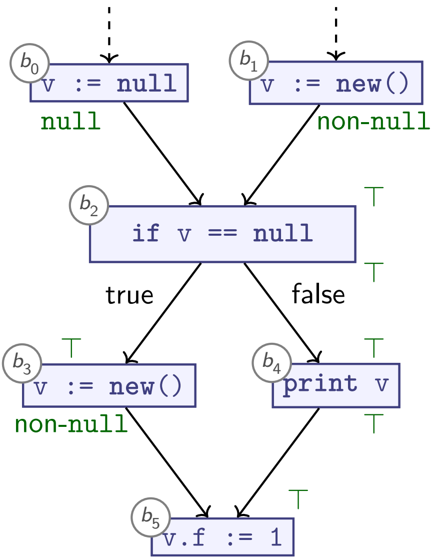
- Data Flow analysis is highly versatile
- Scalable by adjusting:
  - Lattice and transfer functions
  - Treatment of subroutine calls
  - Data representation
- Today we explore four dimensions of scalability:
  - **More precision**: Control- and Path sensitivity
  - **More speed**: Gen/Kill sets
  - **Infinite lattices**: Widening

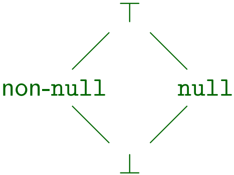
Control Sensitivity





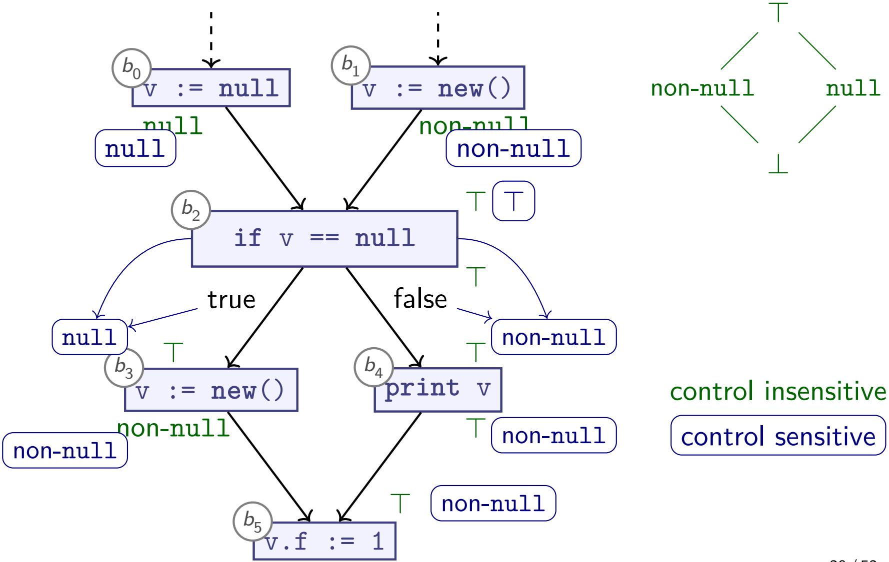
Control Sensitivity



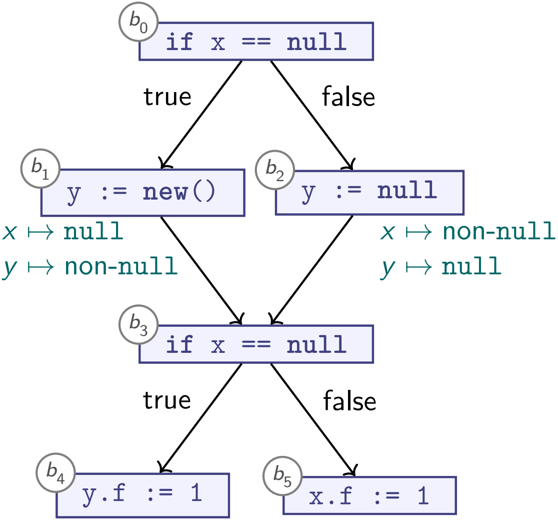


control insensitive

Control Sensitivity

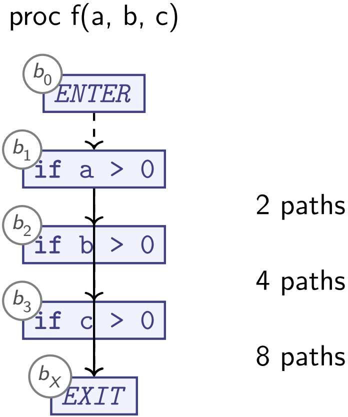


Multiple Conditionals



Should we carry path information across merge points?

Path Sensitivity



Number of paths grows exponentially

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
- **Path sensitive** analysis considers one sequence of CFG edges (execution path) at a time:
  - May propagate different information along different paths
  - High precision possible, but must cover *all* paths
  - Number of paths $O(\#$ of conditionals)
  - Avoid exponential blow-up by merging (as before)
  - Path-sensitive procedure summaries might require exponential number of cases
  - *Usually* not practical

Product Lattices over Binary Lattices

$$\begin{array}{c} \textit{true} \\ | \\ \textit{false} \end{array} \times \begin{array}{c} \textit{true} \\ | \\ \textit{false} \end{array} \times \cdots \times \begin{array}{c} \textit{true} \\ | \\ \textit{false} \end{array}$$

- Recall binary lattices:
  - $\top = \textit{true}$
  - $\bot = \textit{false}$
  - $\sqcup = \textit{logical ``or''}$
  - $\sqcap = \textit{logical ``and''}$
- Computer hardware can compute $\sqcup$, $\sqcap$ of multiple lattices in parallel:
  - Bitwise or/and

  $\Longrightarrow$ Highly efficient
- Can represent other lattices efficiently, too

$\boxed{\textbf{Give rise to highly efficient } \textbf{\textit{Gen-/Kill-Set}} \textbf{ based program analysis}}$

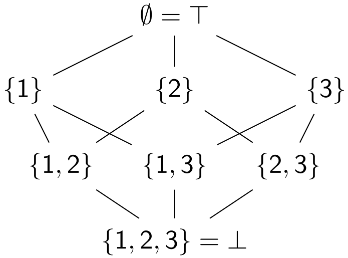
Dataflow Analysis

Analyse properties of variables or basic blocks

Examples in practice:

- *Live Variables*
  Is this variable ever read?
- *Reaching Definitions*
  What are the possible values for this variable?
- *Available Expressions*
  What variable definitely has which expression?

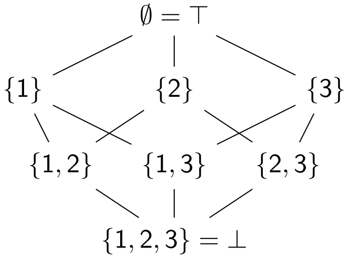
Analyses on Powersets (1/2)



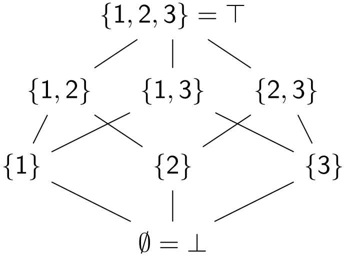
$join_b = \cup$

- Common: 'Which elements of S are possible / necessary?'
  - $S \subseteq \mathbb{Z}$ (*Reaching Definitions*)
    - $S =$ Numeric Constants in code $\cup \{0, 1\}$
  - $S =$ Variables (*Live Variables*)
  - $S =$ Program Locations (*alt. Reaching Definitions*)
  - $S =$ Types
- Abstract Domain: Powerset $\mathcal{P}(S)$
  - Finite iff S is finite

Analyses on Powersets (2/2)



$join_b = \cup$



$join_b = \cap$

- $join_b$ can be $\cup$ or $\cap$
- $\cup$:
  - Property that is true over *any* path
  - **May**-analysis (e.g., Reaching Definitions)
- $\cap$:
  - Property that is true over *all* paths
  - **Must**-analysis

Gen-Sets and Kill-Sets

- Many transfer functions $trans_b$ have the following form:
  - Remove set of options $kill_{x,b}$ from each variable x
  - Add set of options $gen_{x,b}$ to each variable x
  - Don't depend on other variables

  $$trans_b(\{x \mapsto A, \ldots\}) = \{x \mapsto (A \setminus kill_{x,b}) \cup gen_{x,b}, \ldots\}$$

- Bit-vector implementation:
  - $A \setminus B$: bitwise-AND with bitwise-NOT of B
  - $A \cup B$: bitwise-OR
- Examples:
  - *Reaching Definitions* on finite domain
    - *gen*: assignment to var in current basic block
    - *kill*: other existing assignments to same var
  - *Live Variables*
    - *gen*: used variables
    - *kill*: overwritten variables

Gen/Kill: Available Expressions

"Which expressions do we currently have evaluated and stored?"

C

```c
int a = 3 + x;
int y = 2 + z;
if (z > 0) {
  x = 4;
}
f(2 + z); // Can re-use y here!
f(3 + x); // Cannot use a, since x changed
```

- Forward analysis
- *gen*: any (sub)expressions computed
- *kill*: old expressions whose variables changed
- $join_b = \cap$

Gen/Kill: Very Busy Expressions

"Which expression do we definitely need to evaluate at least once?"

C

```c
// (x / 42) is very busy:  (A),(B)
if (z > 0) {
  x = 4 + x / 42; // (A)
  y = 1;
} else {
  x = x / 42; // (B)
}
g(x);
```

- Backward analysis
- *gen*: any (sub)expression computed
- *kill*: old expressions whose variables changed
- $join_b = \cap$

Summary

- Common: Abstract Domain is powerset of some set S
- Transfer function trans_b:

$$\mathit{trans}_b(\{x \mapsto A, \ldots\}) = \{x \mapsto (A \setminus \mathit{kill}_{x,b}) \cup \mathit{gen}_{x,b}, \ldots\}$$

- *kill*: 'Kill set': Entries of S to remove
- *gen*: 'Gen set': Entries of S to add
- join_b is $\cup$ or $\cap$
- Often admits very efficient implementation

	May	**Must**
Forward	Reaching Definitions	Available Expressions
Backward	Live Variables	Very Busy Expressions

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**
- `i` : $[0, 3]$

Numerical Domains

Teal

```
var a := [1, 2, 3];
var i := 0;
var r i: [0,2] new 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;
}
```

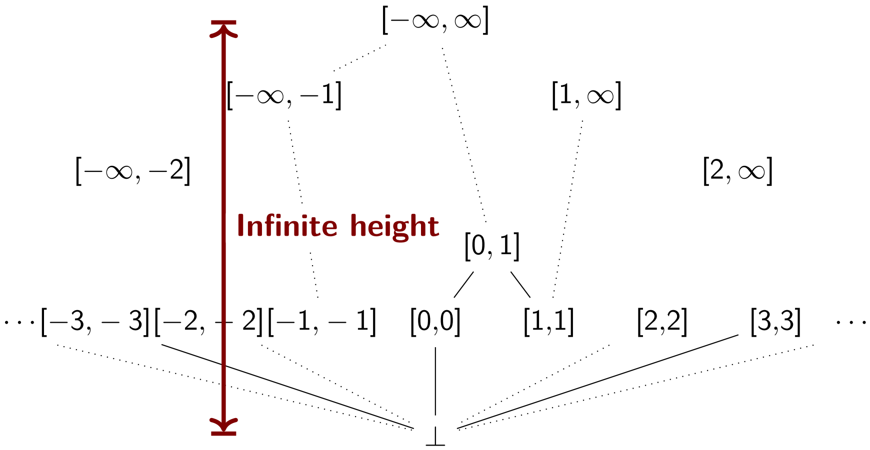
Out of bounds?

- Guarantee: $j < 3 - i$
  $\Longrightarrow j + i < 3$
- 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

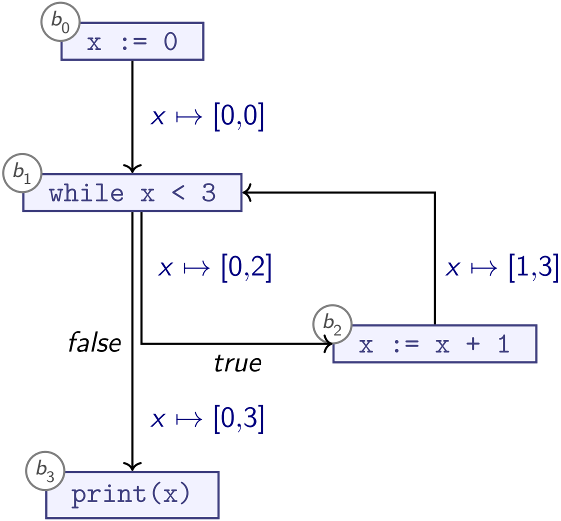


- $\top = [-\infty, \infty]$
- $[l_1, r_1] \sqcup [l_2, r_2] = [min(l_1, l_2), max(r_1, r_2)]$

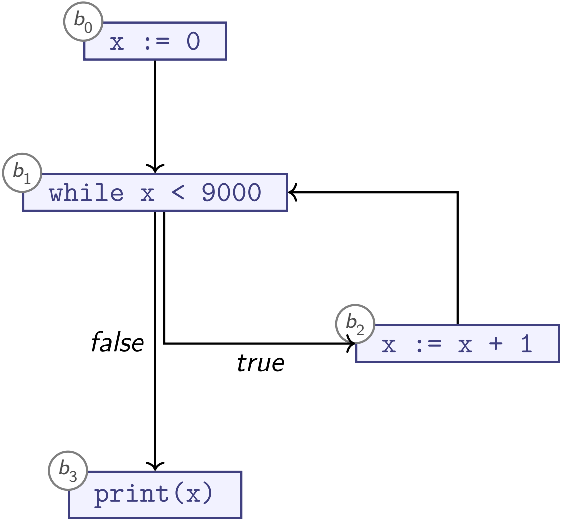
Summary

- 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

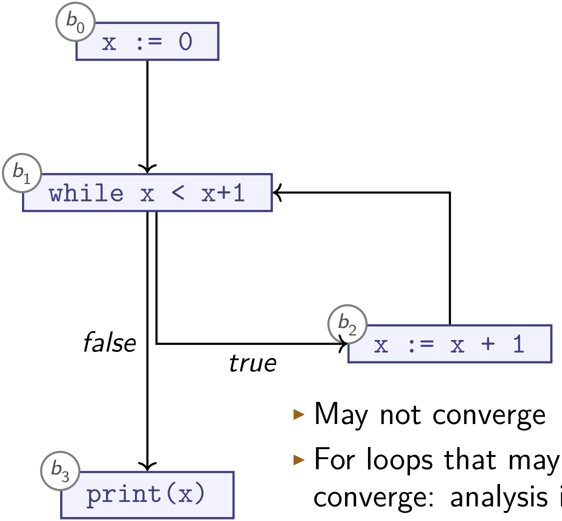
Applying the Interval Domain



Applying the Interval Domain

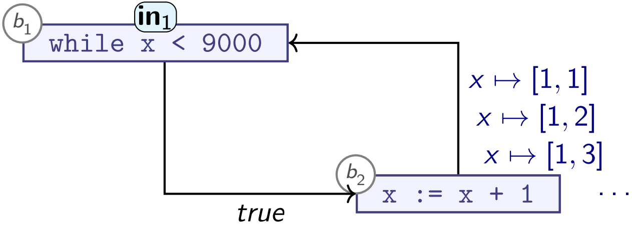


Applying the Interval Domain



- 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**

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

  - For a suitable **widen** function

Widening Functions

- For convergence: satisfy Ascending Chain Condition on:

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

- Suitable functions for Interval Domain?
  - $\textbf{widen}_\top(v) = \top$
    - Very conservative
    - Ensures convergence
  - $\textbf{widen}_{10000}([l,r]) = [l - 10000, r + 10000]$
    - *No convergence*: still allows infinite ascending chain
  - $\textbf{widen}_\mathcal{K}([l,r]) = [max(\{v \in \mathcal{K} | v < l\}), min(\{v \in \mathcal{K} | v > r\})]$
    - 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: $\mathbf{in}_i := \mathbf{in}_i \nabla \mathbf{out}_j$

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
  - Gen-Kill style analyses
  - Numerical Domains (Interval Domain etc.)
  - Widening
- Next up: Analysing the Heap

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