Abstract Interpretation
Non-Standard Semantics

Lecture 8-9

ECS 240

The Problem

- It is useful to predict program behavior *statically* (without running the program)
  - For optimizing compilers
  - For software engineering tools
- The semantics we studied so far give us the precise semantics
- However, precise static predictions are impossible
  - The exact semantics is not computable
- We must settle for approximate, but correct static analysis (e.g. VC vs. WP)

The Plan

- We will introduce abstract interpretation by example
- Starting with a miniscule language we will build up to a fairly realistic application
- Along the way we will see most of the ideas and difficulties that arise in a big class of applications

A Tiny Language

- Consider the following language of arithmetic

$$e ::= n \mid e_1 * e_2$$

- The denotational semantics of this language

$$[\![n]\!] = n$$

$$[\![e_1 * e_2]\!] = [\![e_1]\!] \times [\![e_2]\!]$$

- For this language the precise semantics is computable

An Abstraction

- Assume that we are interested not in the value of the expression, but only in its sign:
  - positive (+), negative (-), or zero (0)

- We can define an abstract semantics that computes **only** the sign of the result

$$\sigma\text{: Exp} \to \{-, 0, +\}$$

$\sigma(n) = \text{sign}(n)$

$\sigma(e_1 * e_2) = \sigma(e_1) \otimes \sigma(e_2)$

$\otimes$	-	0	+
-	+	0	-
0	0	0	0
+	-	0	+

Correctness of Sign Abstraction

- We can show that the abstraction is correct in the sense that it correctly predicts the sign

$$[\![e]\!] > 0 \Leftrightarrow \sigma(e) = +$$
$$[\![e]\!] = 0 \Leftrightarrow \sigma(e) = 0$$
$$[\![e]\!] < 0 \Leftrightarrow \sigma(e) = -$$

- Our semantics is abstract but precise

- Proof is by structural induction on expression e
  - Each case repeats similar reasoning

Another View of Soundness

- We associate with each concrete value an abstract value:

$$\beta : \mathbb{Z} \rightarrow \{ -, 0, + \}$$

- This is called the abstraction function
- Conversely we can also define the concretization function:

$$\gamma : \{-, 0, +\} \rightarrow \mathcal{P}(\mathbb{Z})$$

$$\gamma(+) = \{ n \in \mathbb{Z} \mid n > 0 \}$$
$$\gamma(0) = \{ 0 \}$$
$$\gamma(-) = \{ n \in \mathbb{Z} \mid n < 0 \}$$

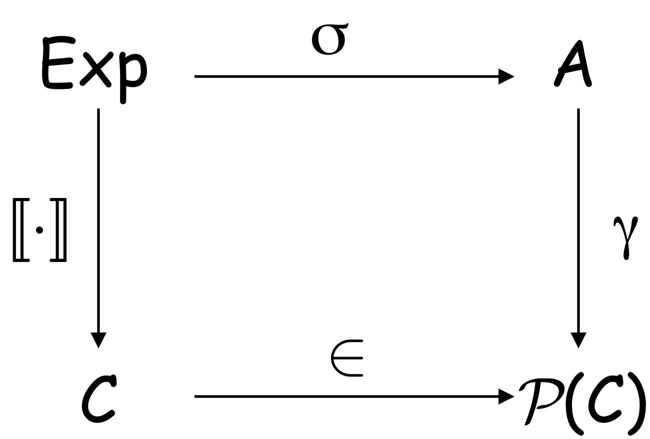
Another View of Soundness (Cont.)

- Soundness can be stated succinctly

$$\forall e \in \text{Exp}.\ [\![e]\!] \in \gamma(\sigma(e))$$

(the true value of the expression is among the concrete values represented by the abstract value of the expression)

- Let *C* be the concrete domain (e.g. $\mathbb{Z}$) and *A* be the abstract domain (e.g. {-, 0, +})



Another View of Soundness (Cont.)

- Consider the generic abstraction of an operator

$$\sigma(e_1 \text{ op } e_2) = \sigma(e_1) \ \underline{\text{op}} \ \sigma(e_2)$$

- This is sound iff

$$\forall a_1 \forall a_2.\ \gamma(a_1 \ \underline{\text{op}} \ a_2) \supseteq \{n_1 \text{ op } n_2 \mid n_1 \in \gamma(a_1), n_2 \in \gamma(a_2)\}$$

- E.g. $\gamma(a_1 \otimes a_2) \supseteq \{ n_1 * n_2 \mid n_1 \in \gamma(a_1), n_2 \in \gamma(a_2) \}$

- This reduces the proof of correctness to one proof for each operator

Abstract Interpretation

- This is our first example of an abstract interpretation.
- We carry out computation in an abstract domain
- The abstract semantics is a sound approximation of the standard semantics
- The concretization and abstraction functions establish the connection between the two domains

Adding Unary Minus and Addition

- We extend the language to $e ::= n \mid e_1 * e_2 \mid - e$
- We define $\sigma(- e) = \ominus\, \sigma(e)$

	-	0	+
$\ominus$	+	0	-

- Now we add addition: $e ::= n \mid e_1 * e_2 \mid - e \mid e_1 + e_2$
- We define $\sigma(e_1 + e_2) = \sigma(e_1) \oplus \sigma(e_2)$

$\oplus$	-	0	+
-	-	-	?
0	-	0	+
+	?	+	+

Adding Addition

- The sign values are not closed under addition
- What should be the value of "$+ \oplus -$"?
- Start from the soundness condition:

$$\gamma(+ \oplus -) \supseteq \{ n_1 + n_2 \mid n_1 > 0, n_2 < 0\} = \mathbb{Z}$$

- We don't have an abstract value whose concretization includes $\mathbb{Z}$, so we add one: $\top$

$\oplus$	-	0	+	$\top$
-	-	-	$\top$	$\top$
0	-	0	+	$\top$
+	$\top$	+	+	$\top$
$\top$	$\top$	$\top$	$\top$	$\top$

Examples

- Abstract computation might loose information

$$[\![(1 + 2) + -3]\!] = 0$$

$$\sigma((1+2) + -3) = (\sigma(1) \oplus \sigma(2)) \oplus \sigma(-3) = (+ \oplus +) \oplus - = \top$$

- We loose some precision
- But this will simplify the computation of the abstract answer in cases when the precise answer is not computable

Adding Division

- Fairly straightforward except for division by 0
  - We say that there is no answer in that case
  - $\gamma(+ \oslash 0) = \{ n \mid n = n_1 / 0 , n_1 > 0 \} = \emptyset$
- We introduce $\perp$ to be the abstraction of the $\emptyset$
  - We also use the same abstraction for non-termination !

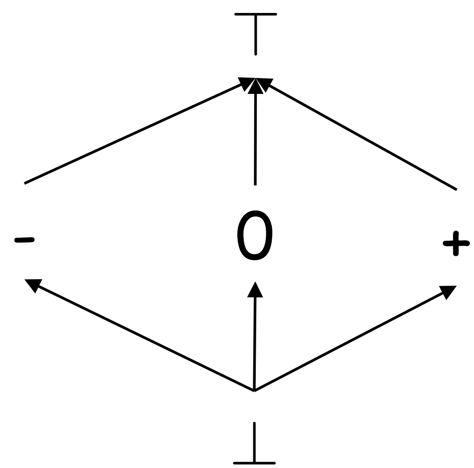
$\oslash$	-	0	+	$\top$	$\perp$
-	+	0	-	$\top$	$\perp$
0	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$
+	-	0	+	$\top$	$\perp$
$\top$	$\top$	$\top$	$\top$	$\top$	$\perp$
$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$

The Abstract Domain

- Our abstract domain forms a <u>lattice</u>
  - A partial order is induced by γ

    $$a_1 \leq a_2 \quad \text{iff } \gamma(a_1) \subseteq \gamma(a_2)$$

    - We say that a_1 is more precise that a_2 !
  - Every <u>finite subset</u> has a least-upper bound (lub) and a greatest-lower bound (glb)



Lattice Facts

- A lattice is complete when all subsets have lub and glb
  - Even infinite ones
- Every finite lattice is complete
- Every complete lattice is a CPO
  - Since a chain is a subset
- Not every CPO is a complete lattice
  - Might not even be a lattice

More Lattice Facts

- Early work in denotational semantics used lattices
  - But it was latter seen that only chains need to have lub
  - And there was no need for $\top$ and glb

- In abstract interpretation we'll use $\top$ to denote "I don't know"
  - Corresponds to all values in the concrete domain

More Definitions

- We can start with the abstraction function

  $\beta : C \to A$ (maps a concrete value to the best abstract value)

  – A must be a lattice
- From here we can derive the concretization function

  $\gamma : A \to \mathcal{P}(C)$

  $\gamma(a) = \{ x \in C \mid \beta(x) \leq a \}$
- And the abstraction for sets

  $\alpha : \mathcal{P}(C) \to A$

  $\alpha(S) = \text{lub} \{ \beta(x) \mid x \in S \}$

Example

- Consider our sign lattice

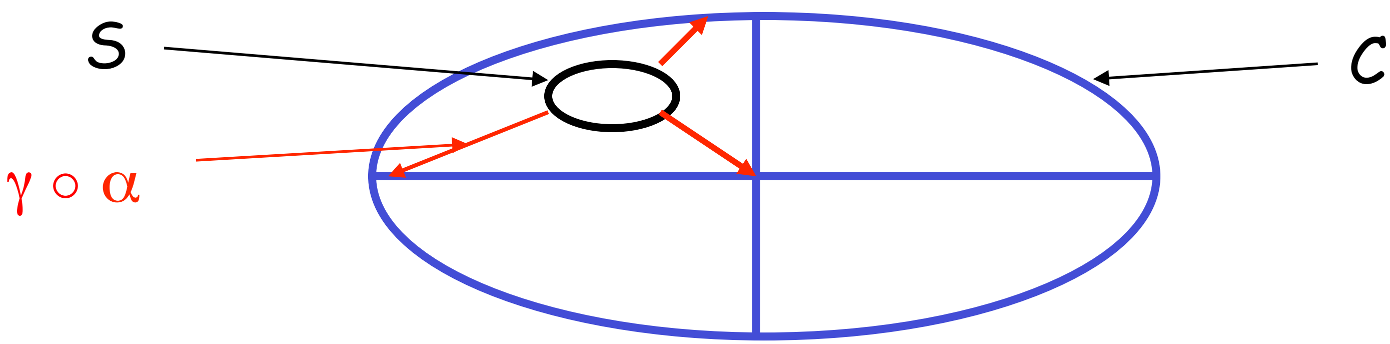
$$\beta(n) = \begin{cases} + & \text{if } n > 0 \\ 0 & \text{if } n = 0 \\ - & \text{if } n < 0 \end{cases}$$

- $\alpha(S) = \text{lub} \{ \beta(x) \mid x \in S \}$
  - Example: $\alpha(\{1, 2\}) = \text{lub} \{ + \} = +$
    $\alpha(\{1, 0\}) = \text{lub} \{ +, 0 \} = \top$
    $\alpha(\{\}) = \text{lub} \{\} = \bot$
- $\gamma(a) = \{ n \mid \beta(n) \leq a \}$
  - Example: $\gamma(+) = \{ n \mid \beta(n) \leq + \} = \{ n \mid \beta(n) = + \} = \{ n \mid n > 0 \}$
    $\gamma(\top) = \{ n \mid \beta(n) \leq \top \} = \mathbb{Z}$
    $\gamma(\bot) = \{ n \mid \beta(n) \leq \bot \} = \emptyset$

Galois Connections

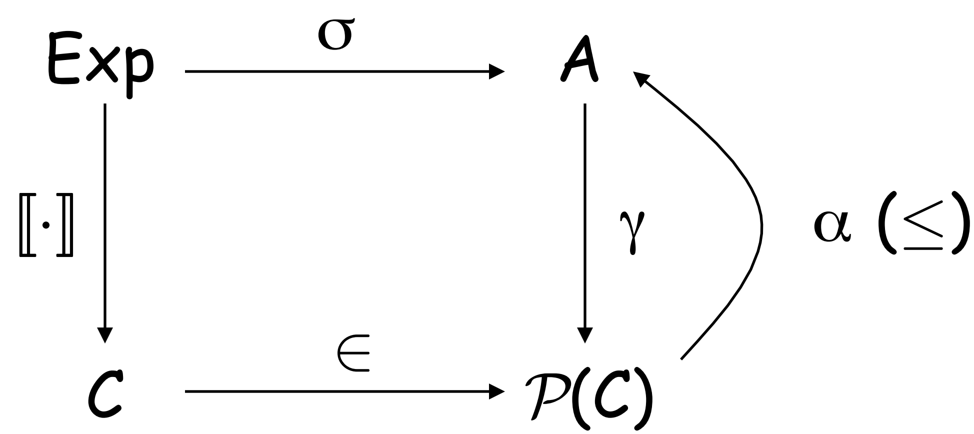
- We can show that
  - γ and α are monotonic (with the $\subseteq$ ordering on $\mathcal{P}(C)$)
  - $\alpha(\gamma(a)) = a$ for all $a \in A$
  - $\gamma(\alpha(S)) \supseteq S$ for all $S \in \mathcal{P}(C)$

- Such a pair of functions is called a <u>Galois connection</u>
  - Between lattices A and $\mathcal{P}(C)$



Correctness Condition

- In general, abstract interpretation satisfies the following diagram



Correctness Conditions

Conditions for correct abstract interpretations

1. α and γ are monotonic

2. α and γ form a Galois connection

3. Abstraction of operations is correct

$$a_1 \underline{op} a_2 = \alpha(\gamma(a_1) \text{ op } \gamma(a_2))$$

So far

- Introduced abstract interpretation
- Two mappings form a Galois connection
  - An abstraction mapping from concrete to abstract values
  - A concretization mapping from abstract to concrete values
- Next look a bit more at Galois connections
- Then extend these ideas from expressions to programs

Why Galois Connections ?

- We have an abstract domain A
  - An abstraction function $\beta : \mathbb{Z} \to A$
  - Induces $\alpha : \mathcal{P}(\mathbb{Z}) \to A$ and $\gamma : A \to \mathcal{P}(\mathbb{Z})$
- We argued that for correctness

$$\gamma(a_1 \underline{op} a_2) \supseteq \gamma(a_1) \text{ op } \gamma(a_2)$$

  - We wish for the set on the left to be as small as possible
  - To reduce the loss of information through abstraction
- For each set $S \subseteq C$, define $\alpha(S)$ as follows:
  - Pick S' the smallest that includes S and is in the image of γ
  - Define $\alpha(S) = \gamma^{-1}(S')$
  - Then we define: $a_1 \underline{op} a_2 = \alpha(\gamma(a_1) \text{ op } \gamma(a_2))$
- Then α and γ form a Galois connection

Abstract Interpretation for Imperative Programs

- So far we abstracted the value of expressions
- We want now to abstract the state at each point in the program
- First we define the concrete semantics that we are abstracting
  - We use a collecting semantics

The Collecting Semantics

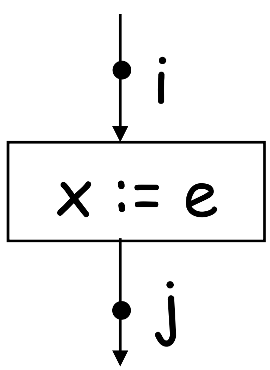
- Recall
  - A state $\sigma \in \Sigma$ = Var $\rightarrow \mathbb{Z}$
  - States vary from program point to program point
- We introduce a set of program points: Labels
- We want to answer questions like:
  - Is x always positive at label i ?
  - Is x always greater or equal to y at label j ?
- To answer these questions it helps to construct

$$C \in \text{Contexts} = \text{Labels} \rightarrow \mathcal{P}(\Sigma)$$

  - For each label, all the states at that label
  - This is called the collecting semantics of the program
- How can we define the collecting semantics ?

Defining the Collecting Semantics

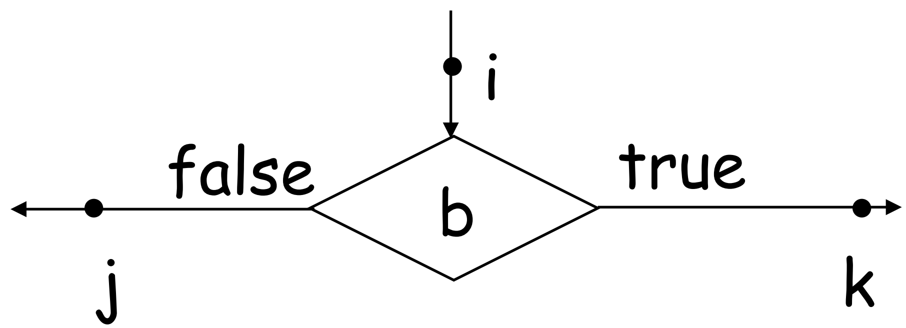
- We first define relations between the collecting semantics at different labels
  - We do it for a flowchart program
  - It can be done for IMP with careful definition of program points
- Define a label on each edge in the flowchart
- For assignment



$$C_j = \{\sigma[x := n] \mid \sigma \in C_i \wedge [\![e]\!]\sigma = n\}$$

Defining the Collecting Semantics

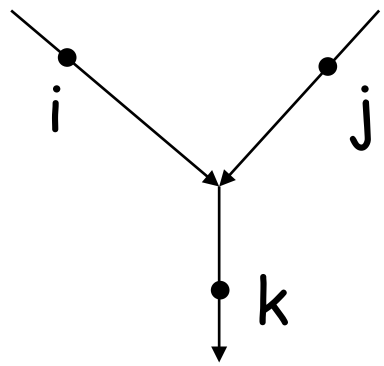
- For conditionals



$$C_j = \{\ \sigma \mid \sigma \in C_i \wedge [\![b]\!]\sigma = \text{false}\}$$
$$C_k = \{\ \sigma \mid \sigma \in C_i \wedge [\![b]\!]\sigma = \text{true}\}$$

Defining the Collecting Semantics

- For a join

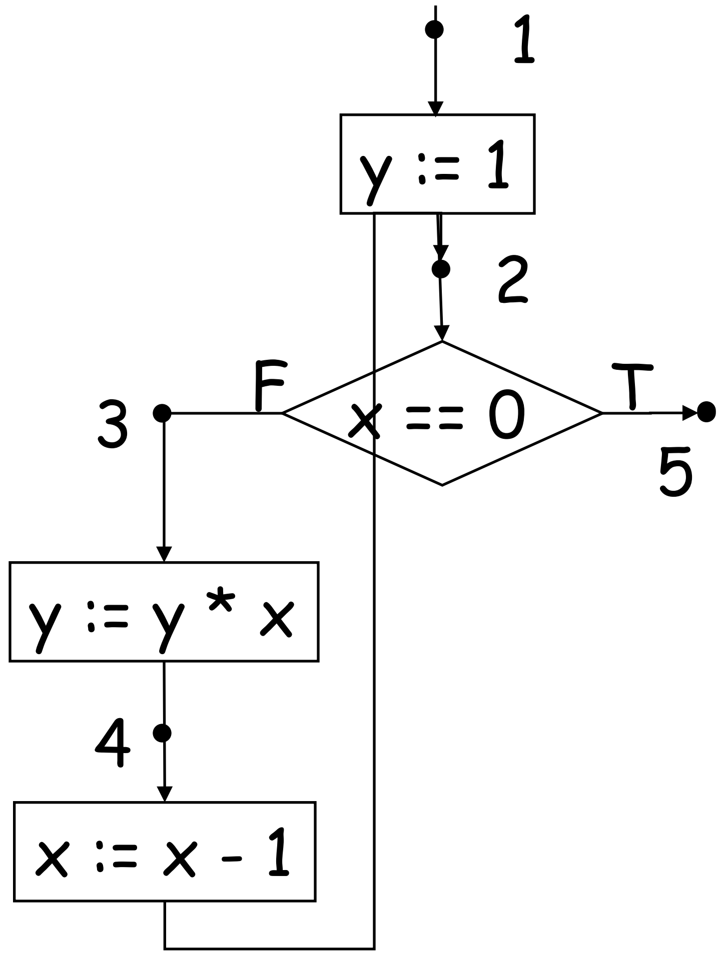


$$C_k = C_i \cup C_j$$

- Verify that these relations are monotonic
  - If we increase a C_i all other C_j can only increase

Collecting Semantics: Example

- Consider the following program (assume $x \geq 0$ initially)



$C_1 = \{\sigma \mid \sigma(x) \geq 0\}$

$C_2 = \{\sigma[y:=1] \mid \sigma \in C_1\} \cup \{\sigma[x:=\sigma(x)-1] \mid \sigma \in C_4\}$

$C_3 = C_2 \cap \{\sigma \mid \sigma(x) \neq 0\}$

$C_5 = C_2 \cap \{\sigma \mid \sigma(x) = 0\}$

$C_4 = \{\sigma[y:=\sigma(y)*\sigma(x) \mid \sigma \in C_3\}$

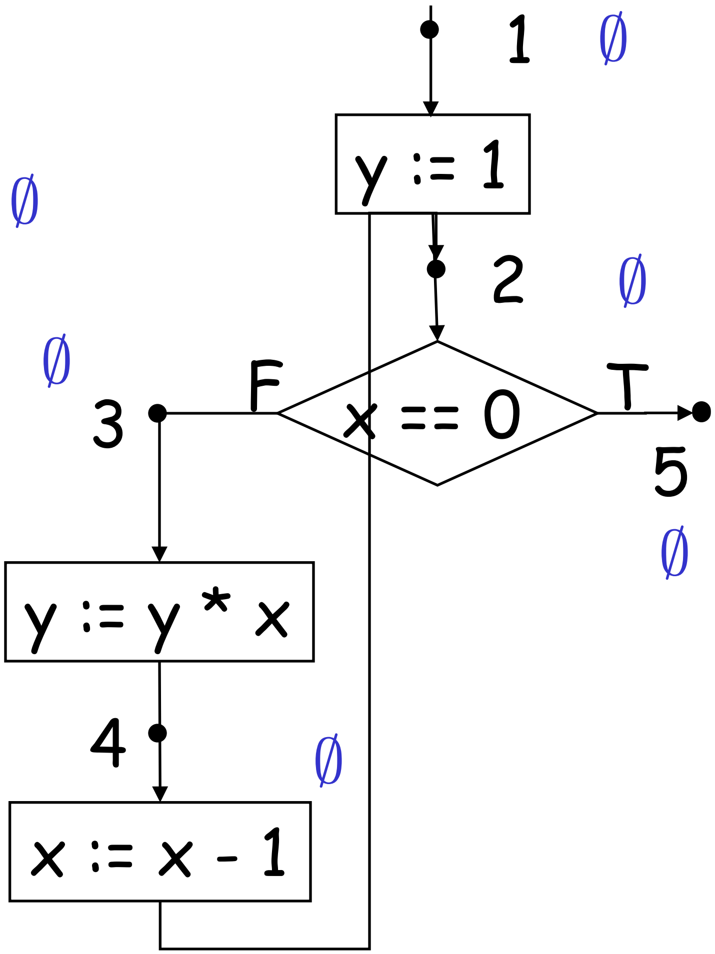
The Collecting Semantics

- We have an equation with the unknown C
  - The equation is defined by a monotonic and continuous function on the domain Labels $\rightarrow \mathcal{P}(\Sigma)$

- We can use the least fixed-point theorem
  - We start with $C^0 = \lambda L.\emptyset$
  - We apply the relations between C_i and C_j to construct $C^1{}_i$ from $C^0{}_j$
  - We stop when $C^k = C^{k-1}$
  - The problem is that we'll go on forever for most programs
  - But we know the fixed point exists

Collecting Semantics: Example

- Consider the following program (assume $x \geq 0$ initially)



$$C_1 = \{\sigma \mid \sigma(x) \geq 0\}$$

$$C_2 = \{\sigma[y{:=}1] \mid \sigma \in C_1\} \cup \{\sigma[x{:=}\sigma(x)-1] \mid \sigma \in C_4\}$$

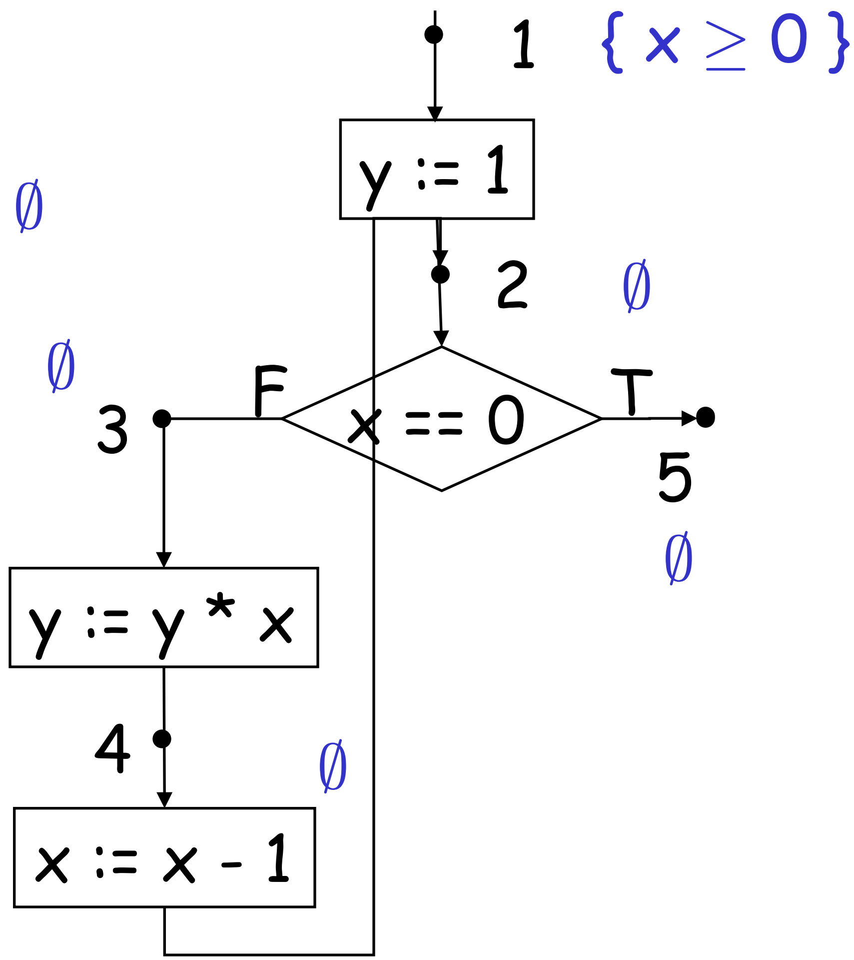
$$C_3 = C_2 \cap \{\sigma \mid \sigma(x) \neq 0\}$$

$$C_5 = C_2 \cap \{\sigma \mid \sigma(x) = 0\}$$

$$C_4 = \{\sigma[y{:=}\sigma(y)*\sigma(x) \mid \sigma \in C_3\}$$

Collecting Semantics: Example

- Consider the following program (assume $x \geq 0$ initially)



$C_1 = \{\sigma \mid \sigma(x) \geq 0\}$

$C_2 = \{ \sigma[y:=1] \mid \sigma \in C_1\} \cup \{\sigma[x:=\sigma(x)-1] \mid \sigma \in C_4\}$

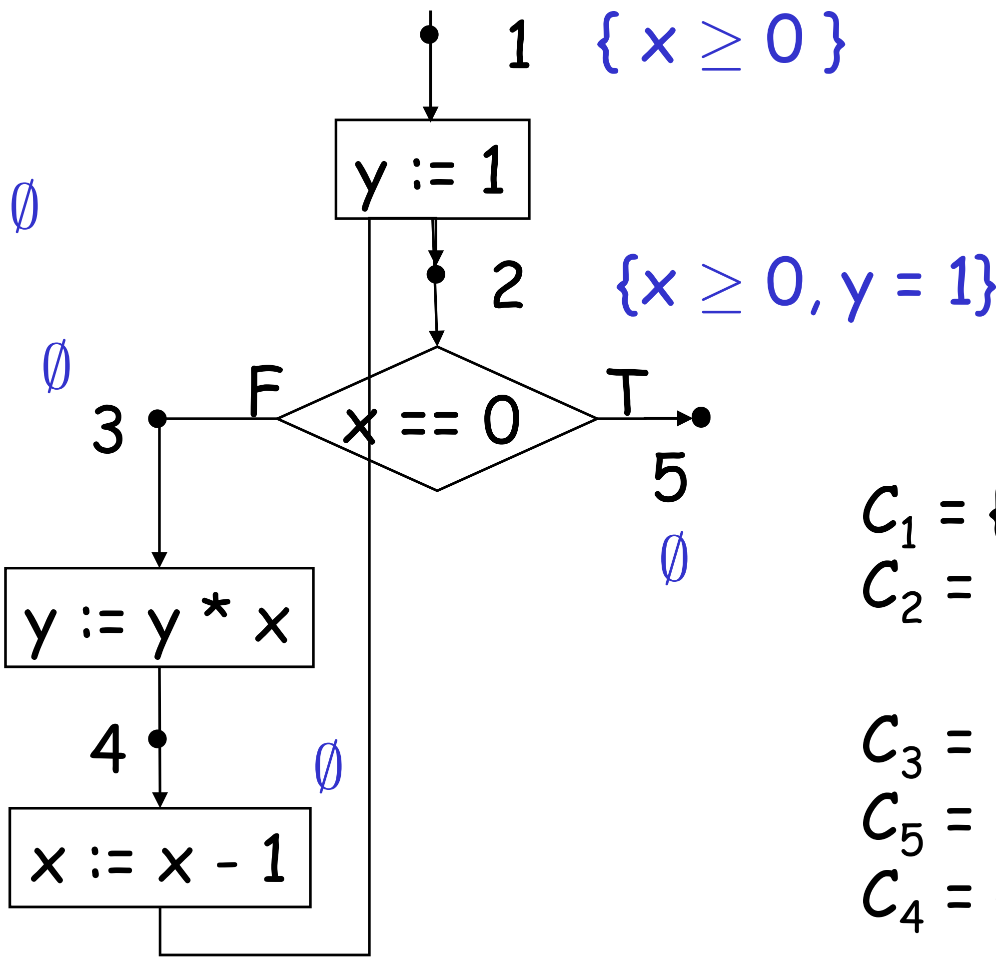
$C_3 = C_2 \cap \{\sigma \mid \sigma(x) \neq 0\}$

$C_5 = C_2 \cap \{\sigma \mid \sigma(x) = 0\}$

$C_4 = \{\sigma[y:=\sigma(y)*\sigma(x) \mid \sigma \in C_3\}$

Collecting Semantics: Example

- Consider the following program (assume $x \geq 0$ initially)



$C_1 = \{\sigma \mid \sigma(x) \geq 0\}$

$C_2 = \{\sigma[y:=1] \mid \sigma \in C_1\}$
$\quad \cup \{\sigma[x:=\sigma(x)-1] \mid \sigma \in C_4\}$

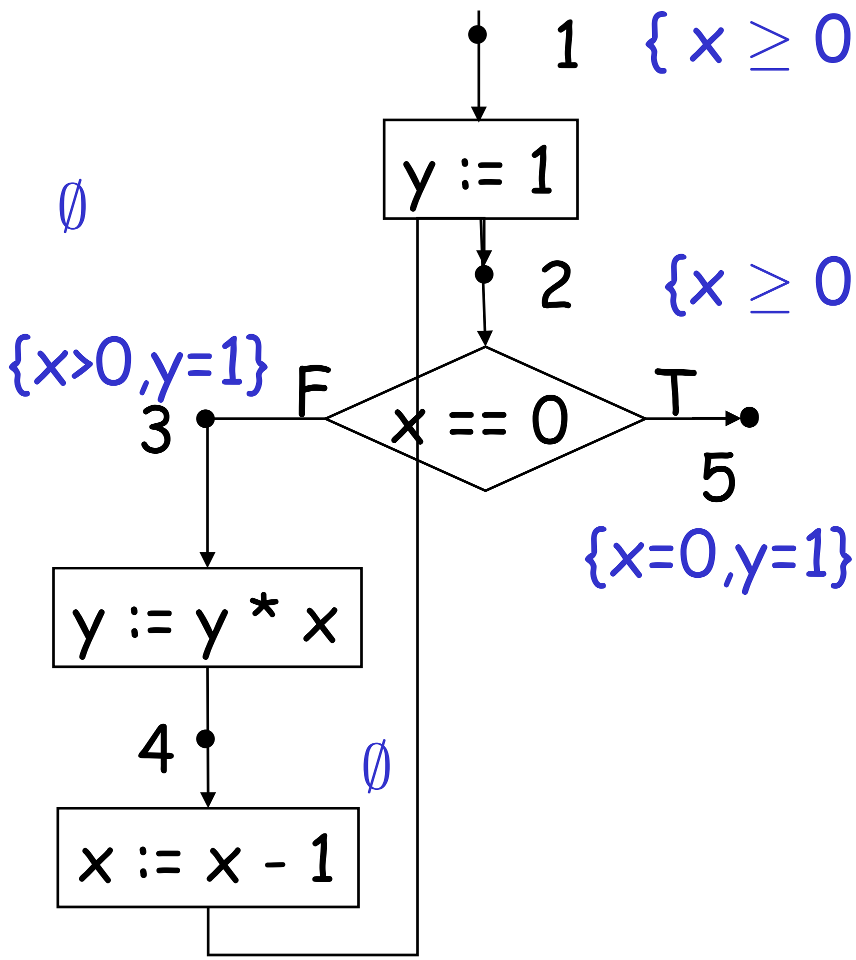
$C_3 = C_2 \cap \{\sigma \mid \sigma(x) \neq 0\}$

$C_5 = C_2 \cap \{\sigma \mid \sigma(x) = 0\}$

$C_4 = \{\sigma[y:=\sigma(y)*\sigma(x) \mid \sigma \in C_3\}$

Collecting Semantics: Example

- Consider the following program (assume $x \geq 0$ initially)



$$C_1 = \{\sigma \mid \sigma(x) \geq 0\}$$

$$C_2 = \{\sigma[y{:=}1] \mid \sigma \in C_1\} \cup \{\sigma[x{:=}\sigma(x)-1] \mid \sigma \in C_4\}$$

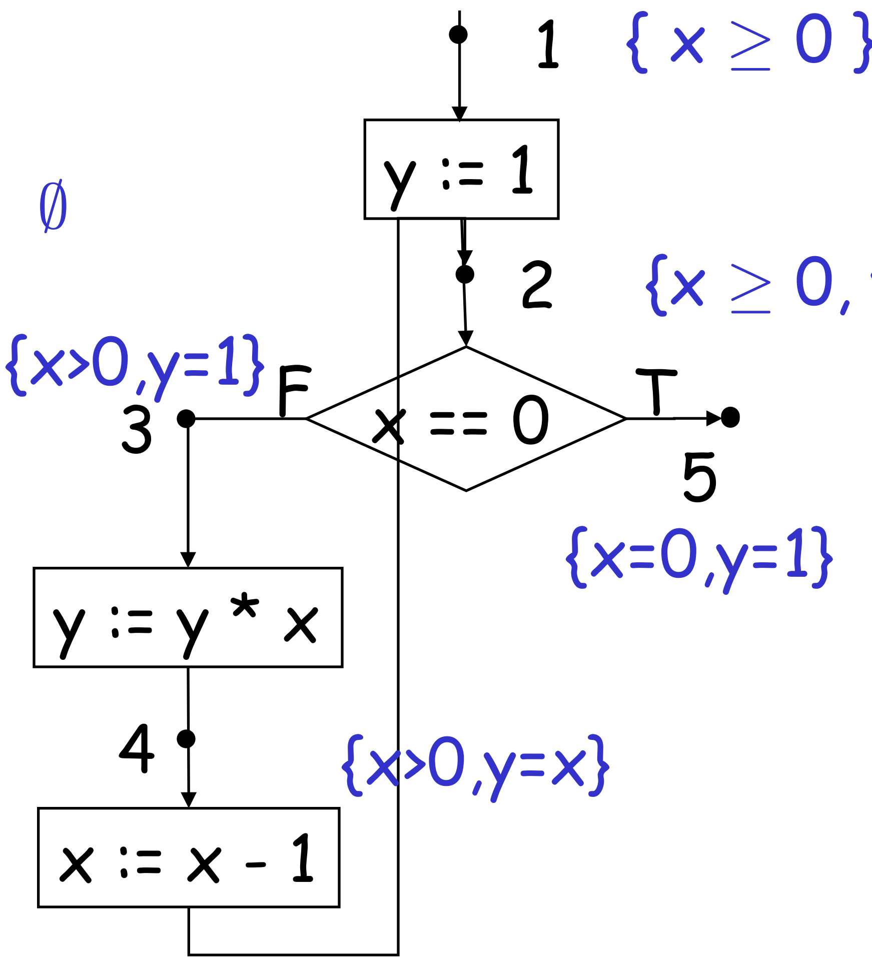
$$C_3 = C_2 \cap \{\sigma \mid \sigma(x) \neq 0\}$$

$$C_5 = C_2 \cap \{\sigma \mid \sigma(x) = 0\}$$

$$C_4 = \{\sigma[y{:=}\sigma(y)*\sigma(x) \mid \sigma \in C_3\}$$

Collecting Semantics: Example

- Consider the following program (assume $x \geq 0$ initially)



$C_1 = \{\sigma \mid \sigma(x) \geq 0\}$

$C_2 = \{ \sigma[y{:=}1] \mid \sigma \in C_1\} \cup \{\sigma[x{:=}\sigma(x)-1] \mid \sigma \in C_4\}$

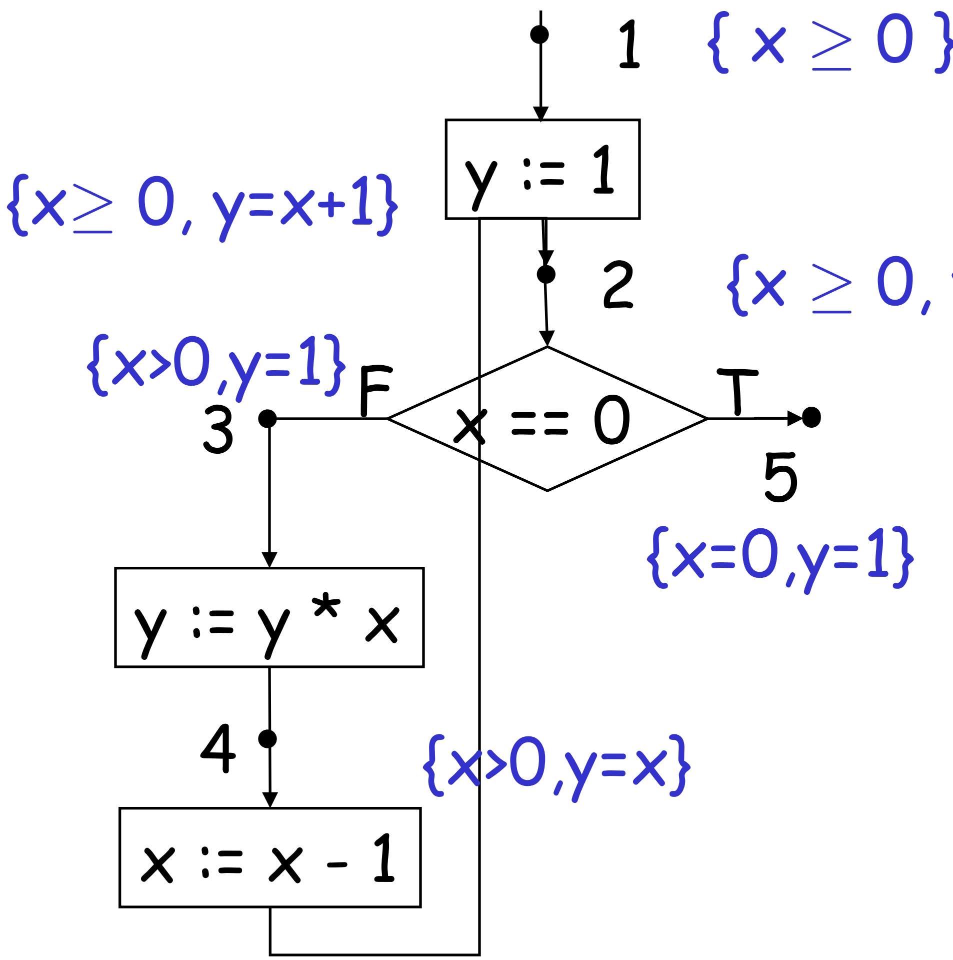
$C_3 = C_2 \cap \{\sigma \mid \sigma(x) \neq 0\}$

$C_5 = C_2 \cap \{\sigma \mid \sigma(x) = 0\}$

$C_4 = \{\sigma[y{:=}\sigma(y)*\sigma(x) \mid \sigma \in C_3\}$

Collecting Semantics: Example

- Consider the following program (assume $x \geq 0$ initially)



$$C_1 = \{\sigma \mid \sigma(x) \geq 0\}$$

$$C_2 = \{\, \sigma[y{:=}1] \mid \sigma \in C_1\} \cup \{\sigma[x{:=}\sigma(x)-1] \mid \sigma \in C_4\}$$

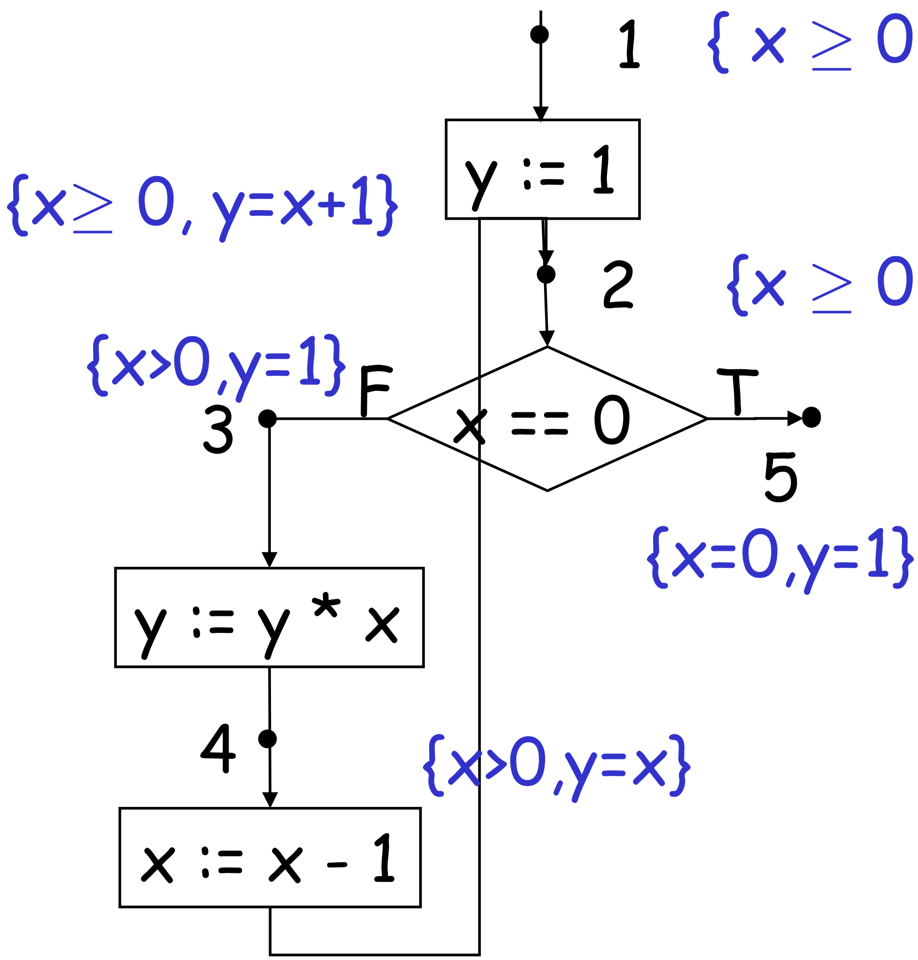
$$C_3 = C_2 \cap \{\sigma \mid \sigma(x) \neq 0\}$$

$$C_5 = C_2 \cap \{\sigma \mid \sigma(x) = 0\}$$

$$C_4 = \{\sigma[y{:=}\sigma(y)*\sigma(x) \mid \sigma \in C_3\}$$

Collecting Semantics: Example

- Consider the following program (assume $x \geq 0$ initially)



$C_1 = \{\sigma \mid \sigma(x) \geq 0\}$

$C_2 = \{\sigma[y:=1] \mid \sigma \in C_1\} \cup \{\sigma[x:=\sigma(x)-1] \mid \sigma \in C_4\}$

$C_3 = C_2 \cap \{\sigma \mid \sigma(x) \neq 0\}$

$C_5 = C_2 \cap \{\sigma \mid \sigma(x) = 0\}$

$C_4 = \{\sigma[y:=\sigma(y)*\sigma(x) \mid \sigma \in C_3\}$

Abstract Interpretation

- We pick a complete lattice A (abstractions for $\mathcal{P}(\Sigma)$)
  - Along with a monotonic abstraction $\alpha : \mathcal{P}(\Sigma) \rightarrow A$
  - Alternatively, pick $\beta : \Sigma \text{ -> } A$
  - This uniquely defines its Galois connection γ
- We take the relations between C_i and move them to the abstract domain:

$$a \in \text{Labels} \rightarrow A$$

- Assignment

Concrete: $C_j = \{\sigma[x := n] \mid \sigma \in C_i \wedge [\![e]\!]\sigma = n\}$

Abstract: $a_j = \alpha\ \{\sigma[x := n] \mid \sigma \in \gamma(a_i) \wedge [\![e]\!]\sigma = n\}$

Abstract Interpretation

- Conditional

  Concrete: $C_j = \{ \sigma \mid \sigma \in C_i \wedge [\![b]\!]\sigma = \text{false}\}$ and

  $C_k = \{ \sigma \mid \sigma \in C_i \wedge [\![b]\!]\sigma = \text{true}\}$

  Abstract: $a_j = \alpha \{ \sigma \mid \sigma \in \gamma(a_i) \wedge [\![b]\!]\sigma = \text{false}\}$ and

  $a_k = \alpha \{ \sigma \mid \sigma \in \gamma(a_i) \wedge [\![b]\!]\sigma = \text{true}\}$

- Join

  Concrete: $C_k = C_i \cup C_j$

  Abstract: $a_k = \alpha (\gamma(a_i) \cup \gamma(a_j)) = \text{lub} \{a_i, a_j\}$

Least Fixed-Points in the Abstract Domain

- Now we have a recursive equation with unknown "a"
  - Defined by a monotonic and continuous function on the domain Labels $\rightarrow$ A

- We can use the least fixed-point theorem:
  - Start with $a^0 = \lambda L.\bot$
  - Apply the monotonic function to compute a^{k+1} from a^k
  - Stop when $a^{k+1} = a^k$

- Exactly the same computation as for the collecting semantics
  - What is new ?

Least Fixed Point in Abstract Domain

- We have a hope of termination

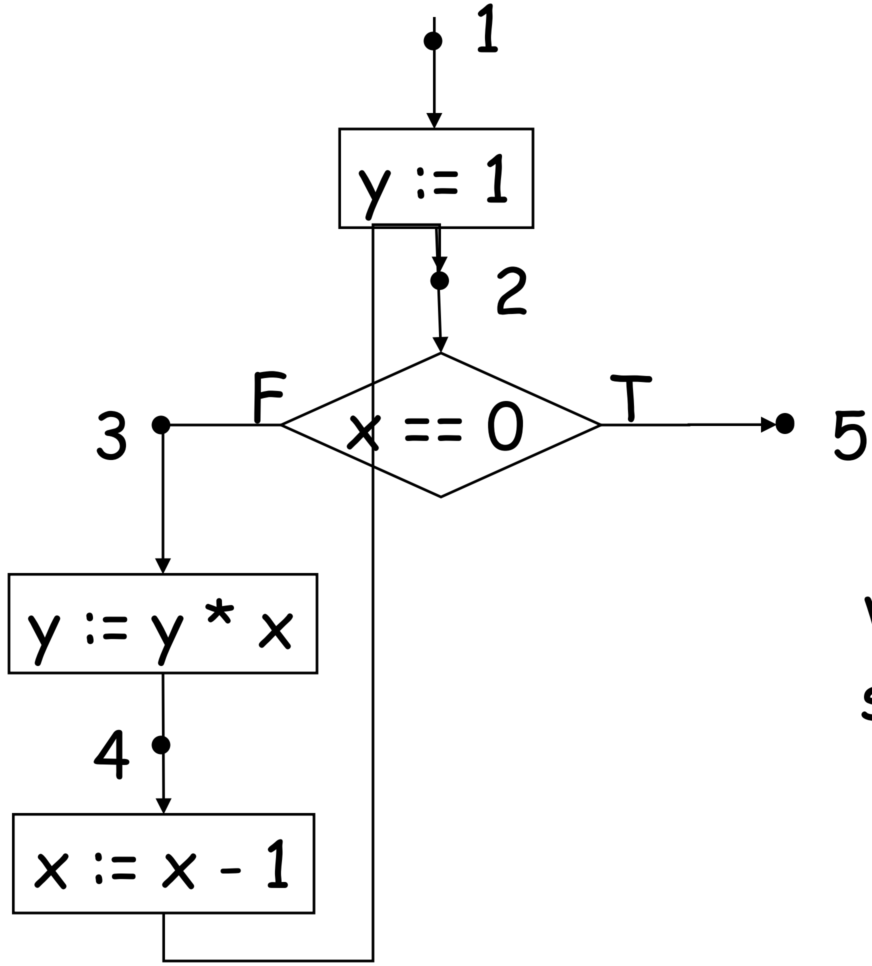
- The classic setup is when A has only uninteresting chains (finite number of elements in each chain)
  - We say that A has finite height (say h)

- In this case the computation takes at most O(h * |Labels|^2) steps
  - At each step "a" makes progress on at least one label
  - We can only make progress h times
  - And each time we must compute |Labels| elements

- This is a quadratic analysis: good news

Abstract Interpretation: Example

- Consider the following program



We want to do sign analysis on it

The Abstract Domain for Sign Analysis

- Consider the complete lattice $S = \{ \bot, -, 0, +, \top \}$

- From it construct the complete lattice $A = \{x, y\} \to S$
  - With point-wise ordering as usual
  - The abstract state consists of the sign for x and y

- We start with $a^0 = \lambda L.\lambda v \in \{x,y\}.\bot$

Example

Label		Iterations →										
1	x	+									+	
	y	$\top$									$\top$	
2	x	$\bot$	+			$\top$					$\top$	
	y	$\bot$	+						$\top$		$\top$	
3	x	$\bot$		+			$\top$				$\top$	
	y	$\bot$		+						$\top$	$\top$	
4	x	$\bot$			+			$\top$			$\top$	
	y	$\bot$			+			$\top$			$\top$	
5	x	$\bot$					0				0	
	y	$\bot$					+			$\top$	$\top$	

Notes

- We abstracted the state of each variable independently

$$A = \{x, y\} \to \{\bot, -, 0, +, \top\}$$

- We lost relationships between variables
  - E.g., that at a point x and y are always of the same sign
  - In the previous abstraction we get $\{x := \top, y := \top\}$ at 2

- We can also abstract the state as a whole

$$A = \mathcal{P}(\{\bot, -, 0, +, \top\} \times \{\bot, -, 0, +, \top\})$$

  - For the previous example we now get the abstraction $\{(0, +), (+, +)\}$ at 2

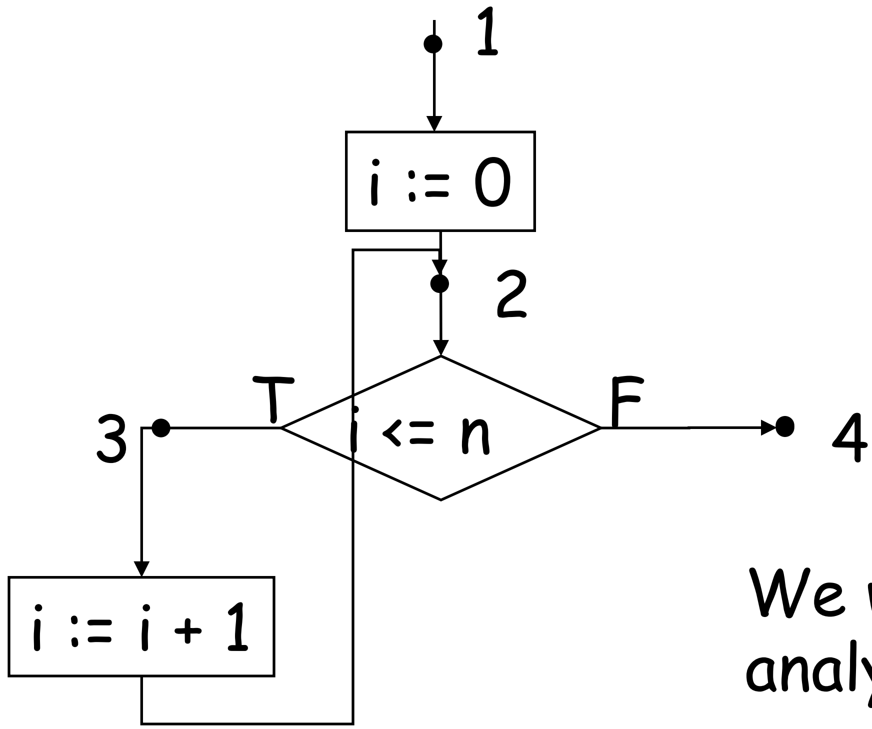
Other Abstract Domains

- Range analysis
  - Lattice of ranges: R ={ $\bot$, [n..m], ($-\infty$, m], [n, $+\infty$), $\top$ }
  - It is a complete lattice
    - [n..m] $\sqcup$ [n'..m'] = [min(n, n')..max(m,m')]
    - [n..m] $\sqcap$ [n'..m'] = [max(n, n')..min(m, m')]
    - With appropriate care in dealing with ∞
  - $\beta : \mathbb{Z} \to$ R such that β(n) = [n..n]
  - $\alpha : \mathcal{P}(\mathbb{Z}) \to$ R such that α(S) = lub {β(n) | n $\in$ S} = [min(S)..max(S)]
  - γ : R $\to \mathcal{P}$(Z) such that γ(r) = { n | n $\in$ r }

- This lattice has infinite-height chains
  - So the abstract interpretation might not terminate !

Example of Non-Termination

- Consider this (common) program fragment



We want to do range analysis for it

Example of Non-Termination

- Consider the sequence of abstract states at point 2
  - [0..0], [0..1], [0..2], …
  - The analysis never terminates
  - Or terminates very late if the loop bound is known statically

- It is time to approximate even more: <u>widening</u>
- We redefine the join (lub) operator of the lattice to ensure that from [0..0] upon union with [1..1] the result is [0..+∞) and not [0..1]
- Now the sequence of states is
  - [0..0], [0, +∞), [0, +∞) Done (no more infinite chains)

Other Abstract Domains

- Linear relationships between variables
  - A convex polyhedron is a subset of $\mathbb{Z}^k$ whose elements satisfy a number of inequalities: $a_1 x_1 + a_2 x_2 + \ldots + a_k x_k \geq c$
  - This is a complete lattice. Use linear programming methods for computing lub
- Linear relationships with at most two variables
  - Like convex polyhedra but with at most two variables per constraint
  - Octagons: $x \pm y >= c$ have efficient algorithms
- Modulo constraints
  - E.g. even and odd

Summary of Abstract Interpretation

- AI is a very powerful technique that underlies a large number of program analyses
- AI can also be applied to functional and logic programming languages

- There are a few success stories
  - Strictness analysis for lazy functional languages
  - PolySpace for linear constraints

- In most other cases however AI is still slow

- When the lattices have infinite height and widening heuristics are used the result becomes unpredictable