Abstract Interpretation

Announcements

- Will conclude with dataflow and abstract interpretation this week and move on

- HW3 and HW4?

$x[i] = y$

...

$y' = x'[j]$

$x = r1.m(r2)$

$m' \leftarrow resolve(A, m)$

ret $m'(this, parameters)$

(@return)

r1 → A, B

A m()

B m()

RTA r1, r2, r3 : I

0-CFA r1 : S_{r1}, r2 : S_{r2}, r3 : S_{r2}

Abstract Interpretation

- Patrick Cousot and Radhia Cousot, POPL'77
- A general framework
  - Combines ideas from dataflow analysis (monotone frameworks and fixpoint iteration) and formal verification (axiomatic semantics)
  - Building static analyses
  - Reasoning about correctness of static analysis
  - Comparing static analyses

Lecture Notes Based On

- “Principles of Program Analysis” by Nielsen, Nielsen and Hankin, Chapter 3
  - Alex Salcianu’s friendlier account of Chapter 3: https://web.eecs.umich.edu/~bchandra/courses/papers/Salcianu_AbstractInterpretation.pdf

- Lecture notes by Xavier Rival, ENS
  - https://www.di.ens.fr/~rival/semverif-2017/sem-11-ai.pdf

Outline

- Overview
- Semantics
- Notion of abstraction
- Concretization and abstraction functions
- Galois Connections
- Applications of abstract interpretation

Overview

Points-to analysis is an abstraction. Abstracts infinitely many heap objects $\mathbf{h_j}$ created at site **i** into a single $\mathbf{o_i}$

Program Execution:

$x \rightarrow \mathbf{h_j{:}A}$

x = y.n(z)
passes value of **z** to parameter **p**

Points-to Analysis (PTA):

$x \rightarrow \mathbf{o_i{:}A}$

x = y.n(z)
$\mathbf{pts(z) \sqsubseteq pts(p)}$

Operational Semantics

- Also called trace semantics, or concrete semantics, models a trace of execution
- Memory state maps variables (**V**) to values (**Z**):

  $\boldsymbol{\sigma} : \mathbf{V} \rightarrow \mathbf{Z}$ $\sigma = [x \rightarrow 1, y \rightarrow 2, z \rightarrow 3]$

- Control state describes where we are

  label ℓ (note: we used the term program point)

- Describes transition $(\ell_1, \boldsymbol{\sigma}_1) \rightarrow (\ell_2, \boldsymbol{\sigma}_2)$

(read: program executes statement at label ℓ_1 in state $\boldsymbol{\sigma}_1$ transitioning to label ℓ_2 in state $\boldsymbol{\sigma}_2$)

A Simple Imperative Language: Syntax (We've Seen This Before!)

E ::= **x** | **n**	simple expression
S ::= ~~**x =** *E*~~ | **x =** *E Op E*	assignment
| **while (b)** *Seq*	loop
| **if (b)** *Seq* **else** *Seq*	conditional
Seq ::= **{** *S*; … *S*; **}**	sequence

- **V** is the set of program variables, $\mathbf{x} \in \mathbf{V}$
- **Z** is the set of values variables take, $\mathbf{n} \in \mathbf{Z}$

A Simple Imperative Language: Operational Semantics $[\![E]\!](\sigma)$

- Operational semantics of expressions:
  - $[\![\mathbf{n}]\!](\boldsymbol{\sigma}) = \mathbf{n}$ // constant **n** evaluates to **n**
  - $[\![\mathbf{x}]\!](\boldsymbol{\sigma}) = \boldsymbol{\sigma}(\mathbf{x})$ // variable **x** evaluates to the value **n** that **x** maps to in **σ**

$\sigma = [x \to 1, y \to 1]$

$\ell_1 \; x = 1$ $\quad \sigma' = [x \to 1, y \to 1]$

$\ell_2 \; y = 2$

$\sigma'' = [x \to 1, y \to 2]$

- Assignment: ℓ_j : $\underline{\mathbf{x} = E}$; ℓ_i : …

$(\ell_j, \boldsymbol{\sigma}) \to (\ell_i, \underbrace{\boldsymbol{\sigma}[\mathbf{x} \leftarrow [\![E]\!](\boldsymbol{\sigma})]}_{\sigma'})$

- Assignment: ℓ_j : $\mathbf{x} = E_1 \; Op \; E_2$; ℓ_i : …

$(\ell_j, \sigma) \to (\ell_i, \sigma[x \leftarrow [\![E_1]\!](\sigma) \; Op \; [\![E_2]\!](\sigma)])$

A Simple Imperative Language: Operational Semantics

- Loop: ℓ_j : **while (b)** { ℓ_i : S; … } ℓ_k : ...

  If $[\![b]\!](\sigma) == True$ then $(\ell_j, \sigma) \rightarrow (\ell_i, \sigma)$

  If $[\![b]\!](\sigma) == False$ then $(\ell_j, \sigma) \rightarrow (\ell_k, \sigma)$

- Conditional: ℓ_j : **if (b)** { ℓ_T : … } **else** { ℓ_F : ... }

  If $[\![b]\!](\sigma) == True$ then $(\ell_j, \sigma) \rightarrow (\ell_T, \sigma)$

  If $[\![b]\!](\sigma) == False$ then $(\ell_j, \sigma) \rightarrow (\ell_F, \sigma)$

- Sequence: $\underline{\underline{\{}}$ ℓ_0 : $\underset{x=5}{S}$; ℓ_1 … **}** $(\ell_0, \sigma) \rightarrow^{*} (\ell_j, \underline{\underline{\sigma'}})$ for while stmt seq.

  more than one steps.

  - Transition defined by composition of individual transition relations

Example

σ: empty map

$(1,\sigma) \rightarrow (2,\sigma')$

σ: b→0

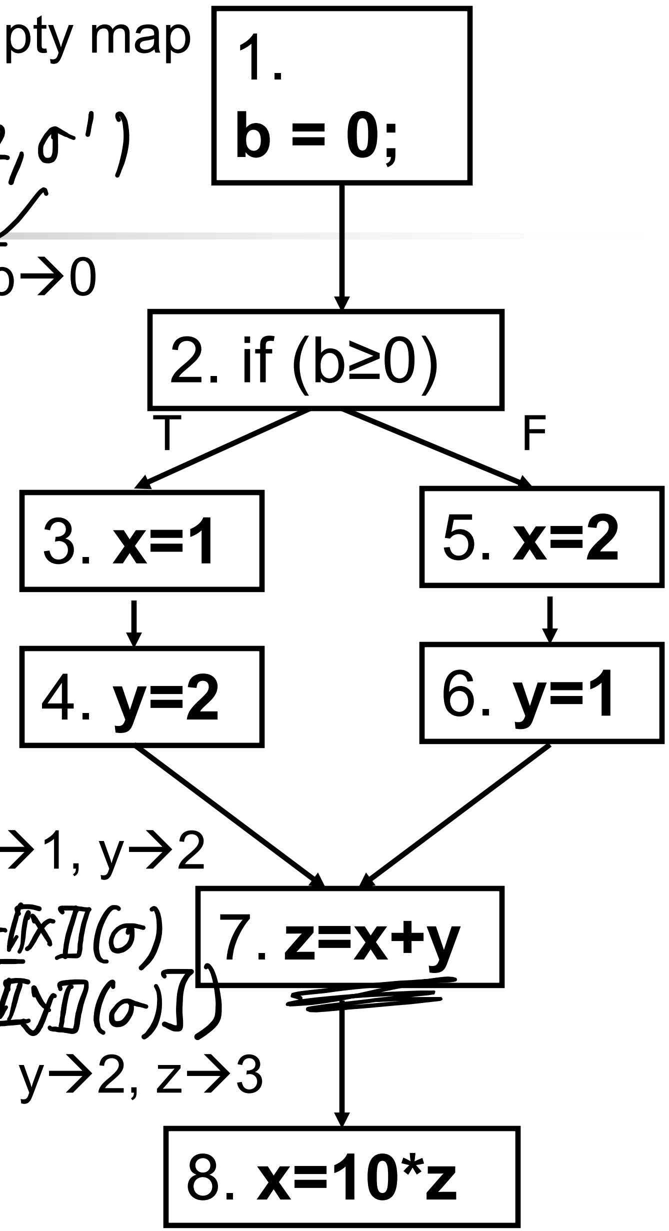
σ: b→0

σ: b→0, x→1

σ: b→0, x→1, y→2

$(7,\sigma) \rightarrow (8,\sigma[z \leftarrow [\![x]\!](\sigma) + [\![y]\!](\sigma)])$

σ: b→0, x→1, y→2, z→3

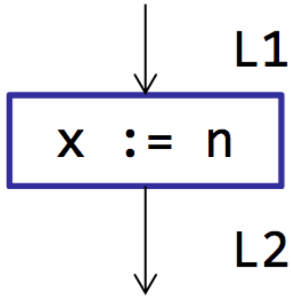


σ: b→0, x→30, y→2, z→3

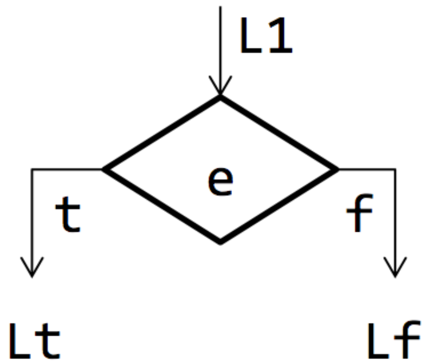
Collecting Semantics

- Collects states (i.e., **σ**'s) along all possible traces of execution at a given label (i.e., program point)

- Given a label, we are interested in a function
  - $\mathscr{C}$: ***Labels → 2^Σ*** Σ is set of all σ
  - The set of all states a program can have at ℓ_i

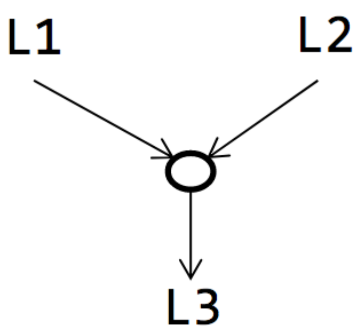


$$\mathcal{C}[L2] = \{\sigma[x \to n] \mid \sigma \in \mathcal{C}[L1]\}$$



$$\mathcal{C}[Lt] = \{\sigma \mid \sigma \in \mathcal{C}[L1], [\![e]\!]\sigma = true\}$$

$$\mathcal{C}[Lf] = \{\sigma \mid \sigma \in \mathcal{C}[L1], [\![e]\!]\sigma = false\}$$



$$\mathcal{C}[L3] = \mathcal{C}[L1] \cup \mathcal{C}[L2]$$

Collecting Semantics

- "Ground truth"
  - We base reasoning about correctness (soundness) of static analysis off of it
- Undecidable
- Relation to MOP solution?
- Define abstraction of state and semantics
- Goal: show that abstraction "accounts for" all values computed by the collecting semantics

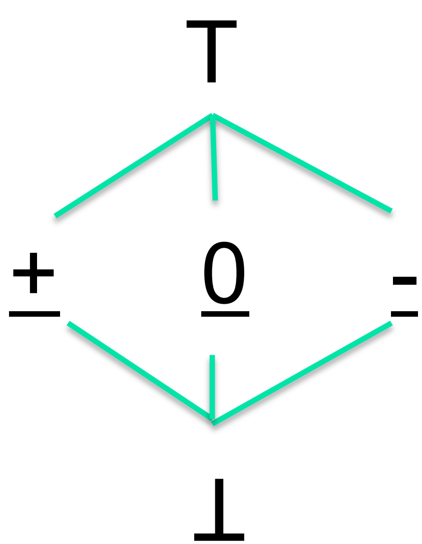
Outline

- Overview
- Semantics
- Notion of abstraction
- Concretization and abstraction functions
- Galois Connections
- Applications of abstract interpretation

Abstraction Example 1: signs

- Concrete values: sets of integers
- Abstract values: signs

Lattice of signs:



- ⊥ represents the empty set
- \+ represents any set of positive integers
- 0 represents set { 0 }
- \- represents any set of negative integers
- T represents any set of integers

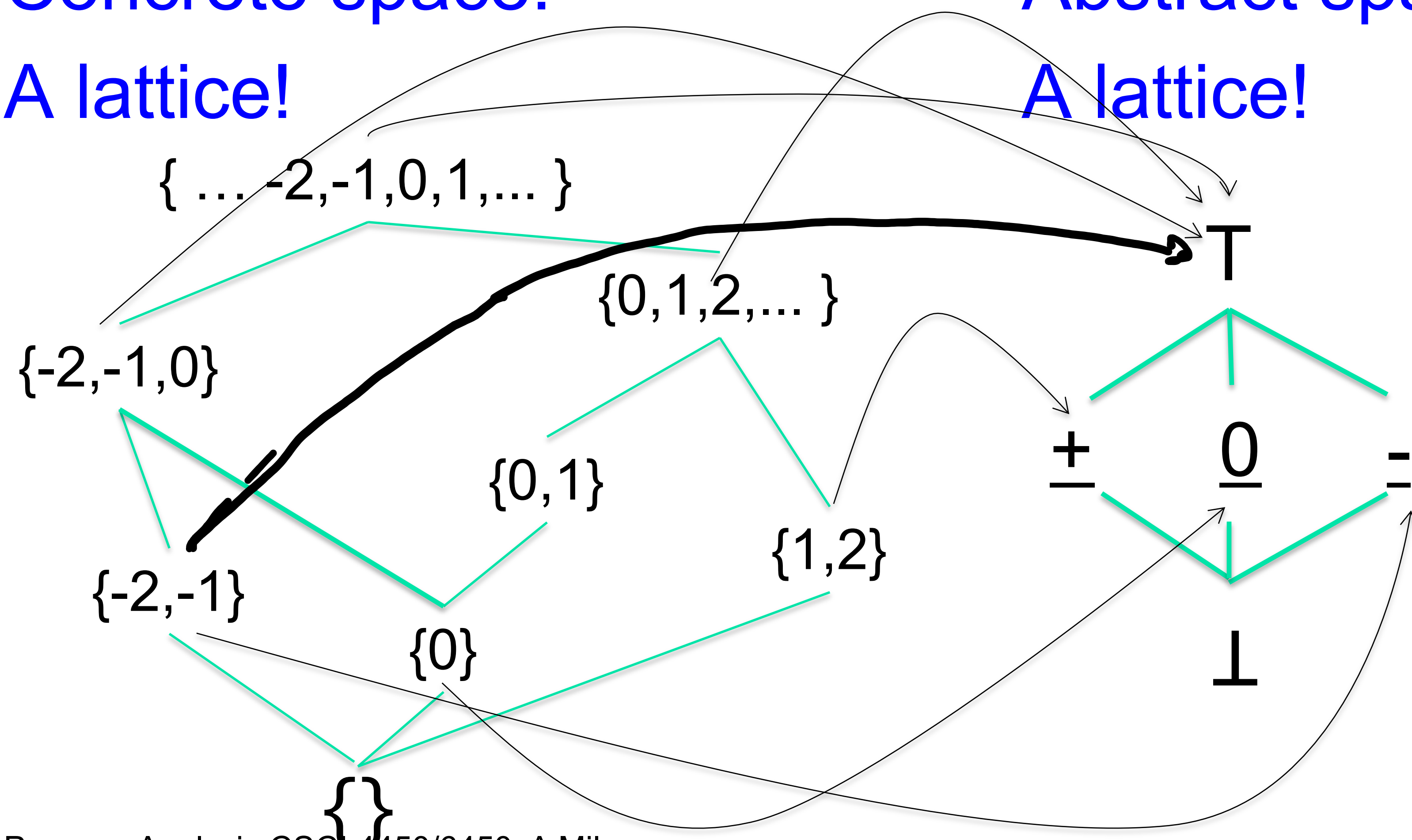
Abstraction Example 1: signs

Concrete space:

A lattice!

Abstract space:

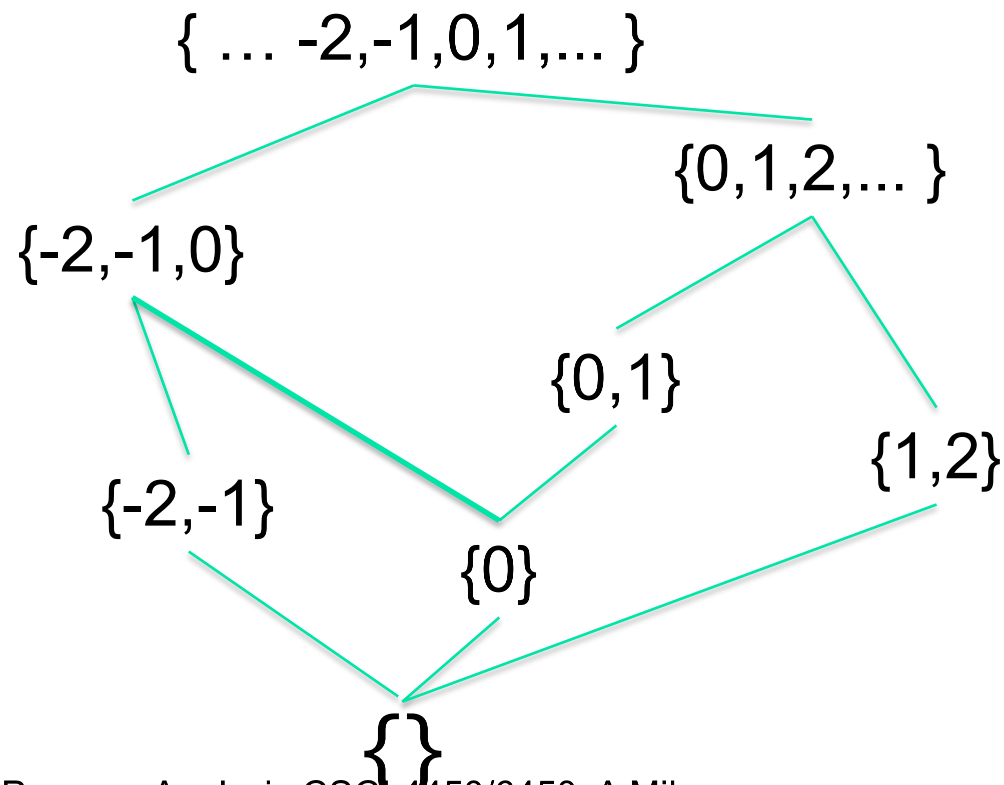
A lattice!



Abstraction Example 1: signs

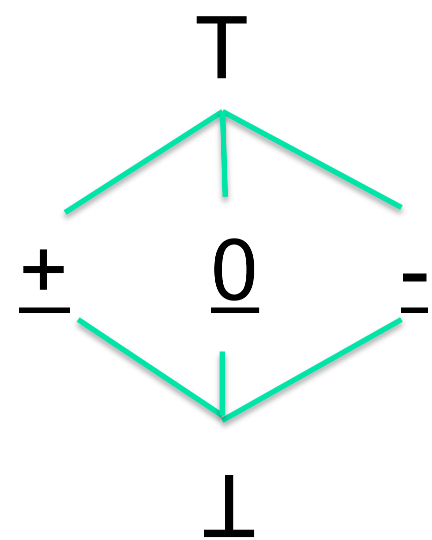
Concrete space:

A lattice!



Abstract space:

A lattice!



Abstraction Example 1: signs

- Concrete elements: elements of the concrete lattice **c** $\in$ $\mathbf{2^Z}$
- Abstract elements: elements of abstract lattice of signs
- Abstraction relation relates concrete elements to abstract ones: **c** $\vdash_S$ **a** (i.e., **a** represents **c,** or conversely **c** is represented by **a**)

$$\{1,2,3\} \vdash_S \pm$$

$$\{1,2,3\} \vdash_S \mathrm{T}$$

Abstraction Example 1: signs

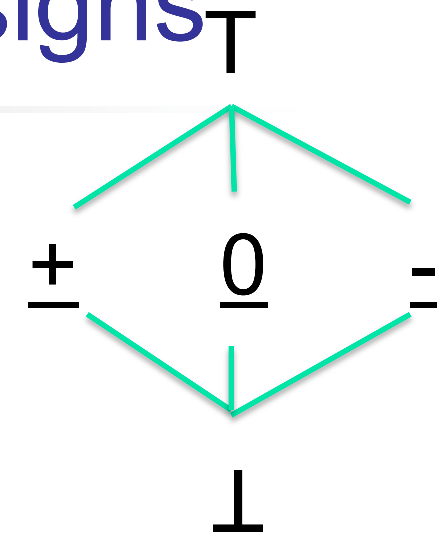
- We use the abstraction relation to define abstract semantics, i.e., the execution of program statements over abstract elements

- If **x** is $\underline{+}$ and **y** is $\underline{+}$ then **x + y** is $\underline{+}$
  - **x's** value is a positive integer
  - **y's** value is a positive integer
  - Therefore, the concrete value of **x + y** is a positive integer too, thus represented by $\underline{+}$

Abstraction Example 1: signs

- If **x** is $\underline{+}$ and **y** is $\underline{+}$ then **x + y** is $\underline{+}$
- Analysis computes over abstract elements
- Correctness conclusion, informally: if analysis (works on abstract elements **a**) determines that **x** at label ℓ is **a**, then **a** represents the set of concrete values **c** collected by the collecting semantics at ℓ

Abstraction Example 1: signs



- We can also use U and ∩

if **x** is $\underline{+}$ and **y** is $\underline{+}$ then **x** U **y** is $\underline{+}$

How about if **x** is $\underline{+}$ and **y** is $\underline{\mathbf{0}}$?

then **x** U **y** is T

because only {0,1,2,3,…} $\vdash_s$ T holds

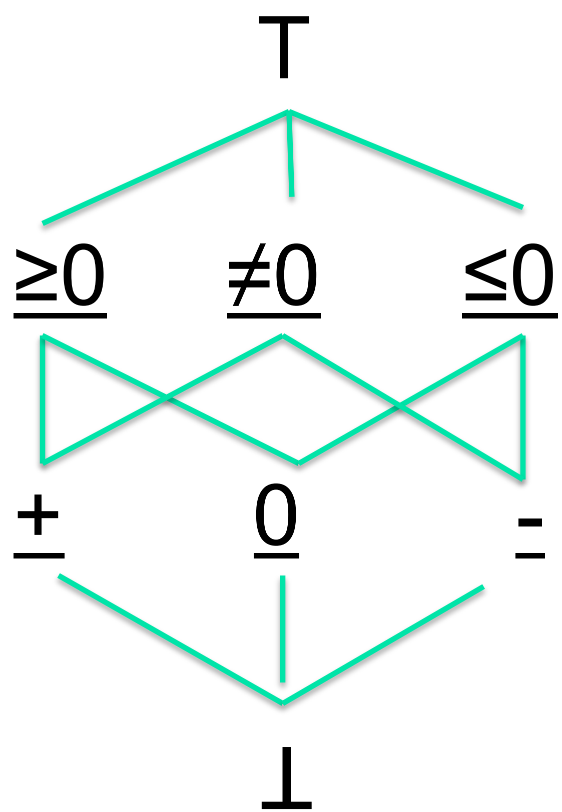
No other relation holds

In the abstract, we include negative integers

in **x** U **y** (we lose precision!)

Abstraction Example 1: signs

- Refine the abstract space



- ⊥ represents the empty set
- + represents any set of positive integers
- 0 represents set { 0 }
- - represents any set of negative integers
- T represents any set of integers

- ≥0 represents any set of non-negative integers
- ≤0 represents any set of non-positive integers
- ≠0 represents any set of non-zero integers

Abstraction Examples

- Will continue next time