

# Abstract interpretation with bounded numeric intervals

Second assignment of the Software Verification  
course, A.Y. 2022/2023

Christian Micheletti  
January 3, 2024



UNIVERSITÀ  
DEGLI STUDI  
DI PADOVA

- 1 The Language
  - Arithmetic Expressions
  - Boolean Expressions
- 2 Abstract Interpreter
  - Abstract States
  - Abstract Semantics
- 3 Implementation
  - The Bounded Intervals Domain
- 4 Introduction
- 5 First section

The language is a variation of the While language seen in class. It differs on:

- it admits some syntactic sugar (it's not minimal);
- its semantic functions model divergence and state changes in both arithmetic and boolean expressions.

$$\begin{aligned} AExp ::= & n \mid x \mid -e \mid (e) \\ & \mid e_1 + e_2 \mid e_1 - e_2 \mid e_1 * e_2 \mid e_1 / e_2 \\ & \mid x++ \mid ++x \mid x-- \mid --x \end{aligned}$$

The syntax allows arithmetic expression that change the state, such as  $x++$  and  $x--$ .

The operator  $(\cdot/\cdot) : \mathbb{N} \rightarrow \mathbb{N} \rightarrow \mathbb{N}$  returns the quotient of the two arguments. It's undefined when the second argument is 0.

$\mathcal{A} : AExp \rightarrow State \hookrightarrow \mathbb{Z} \times State$

$$\mathcal{A}[[n]]\varphi = (n\mathbb{Z}, \varphi)$$

$$\mathcal{A}[[x]]\varphi = (\varphi(x), \varphi)$$

$$\mathcal{A}[[e]]\varphi = \mathcal{A}[[e]]\varphi$$

$$\mathcal{A}[[ - e ]] \varphi = \begin{cases} (-a, \varphi') & \mathcal{A}[[e]]\varphi = (a, \varphi') \\ \uparrow & (\mathcal{A}[[e]]\varphi) \uparrow \end{cases}$$

$\mathcal{A} : AExp \rightarrow State \hookrightarrow \mathbb{Z} \times State$

$$\mathcal{A}[[e_1/e_2]]\varphi = \begin{cases} (a_1 \div a_2, \varphi'') & \mathcal{A}[[e_1]]\varphi = (a_1, \varphi') \\ & \wedge \mathcal{A}[[e_2]]\varphi' = (a_2, \varphi'') \\ & \wedge a_2 \neq 0 \\ \uparrow & \text{otherwise} \end{cases}$$
$$\mathcal{A}[[e_1 \text{ op } e_2]]\varphi = \begin{cases} (a_1 \text{ op } a_2, \varphi'') & \mathcal{A}[[e_1]]\varphi = (a_1, \varphi') \\ & \wedge \mathcal{A}[[e_2]]\varphi' = (a_2, \varphi'') \\ \uparrow & \text{otherwise} \end{cases}$$

$\mathcal{A} : AExp \rightarrow State \hookrightarrow \mathbb{Z} \times State$

$$\mathcal{A}[[x++]]\varphi = (\varphi(x), \varphi[x \mapsto x + 1])$$

$$\mathcal{A}[[++x]]\varphi = \text{let } \varphi' = \varphi[x \mapsto x + 1] \\ \text{in } (\varphi'(x), \varphi')$$

$$\mathcal{A}[[x--]]\varphi = (\varphi(x), \varphi[x \mapsto x - 1])$$

$$\mathcal{A}[[--x]]\varphi = \text{let } \varphi' = \varphi[x \mapsto x - 1] \\ \text{in } (\varphi'(x), \varphi')$$

$$\begin{aligned} BExp ::= & \text{true} \mid \text{false} \mid (b) \mid b_1 \text{ and } b_2 \mid b_1 \text{ or } b_2 \\ & \mid e_1 = e_2 \mid e_1 \neq e_2 \mid e_1 < e_2 \mid e_1 \geq e_2 \\ & \mid e_1 > e_2 \mid e_1 \leq e_2 \end{aligned}$$

The operator  $(\neg \cdot) : \mathbb{T} \rightarrow \mathbb{T}$  is not in the minimal definition: it is defined as syntactic sugar later on.



Operators between booleans short-circuit evaluation:

$$\mathcal{B} : BExp \rightarrow State \hookrightarrow \mathbb{T} \times State$$

$$\mathcal{B}[\![\text{true}]\!] \varphi = (\mathbf{tt}, \varphi)$$

$$\mathcal{B}[\![\text{false}]\!] \varphi = (\mathbf{ff}, \varphi)$$

$$\mathcal{B}[\![b]\!] \varphi = \mathcal{B}[\![b]\!] \varphi$$

$$\mathcal{B}[\![b_1 \text{ and } b_2]\!] \varphi = \begin{cases} (\mathbf{ff}, \varphi') & \mathcal{B}[\![b_1]\!] \varphi = (\mathbf{ff}, \varphi') \\ \mathcal{B}[\![b_2]\!] \varphi' & \mathcal{B}[\![b_1]\!] \varphi = (\mathbf{tt}, \varphi') \\ \uparrow & \text{otherwise} \end{cases}$$

$$\mathcal{B}[\![b_1 \text{ or } b_2]\!] \varphi = \begin{cases} (\mathbf{tt}, \varphi') & \mathcal{B}[\![b_1]\!] \varphi = (\mathbf{tt}, \varphi') \\ \mathcal{B}[\![b_2]\!] \varphi' & \mathcal{B}[\![b_1]\!] \varphi = (\mathbf{ff}, \varphi') \\ \uparrow & \text{otherwise} \end{cases}$$

Comparison operators propagate the state transition(s):

$$\mathcal{B} : BExp \rightarrow State \hookrightarrow \mathbb{T} \times State$$

$$\mathcal{B}[\![e_1 = e_2]\!] \varphi = \begin{cases} (a_1 = a_2, \varphi'') & \mathcal{A}[\![e_1]\!] \varphi = (a_1, \varphi') \\ & \wedge \mathcal{A}[\![e_2]\!] \varphi' = (a_2, \varphi'') \\ \uparrow & \text{otherwise} \end{cases}$$

$$\mathcal{B}[\![e_1 \neq e_2]\!] \varphi = \begin{cases} (a_1 \neq a_2, \varphi'') & \mathcal{A}[\![e_1]\!] \varphi = (a_1, \varphi') \\ & \wedge \mathcal{A}[\![e_2]\!] \varphi' = (a_2, \varphi'') \\ \uparrow & \text{otherwise} \end{cases}$$

$\mathcal{B} : BExp \rightarrow State \hookrightarrow \mathbb{T} \times State$

$$\mathcal{B}[[e_1 < e_2]]\varphi = \begin{cases} (a_1 < a_2, \varphi'') & \mathcal{A}[[e_1]]\varphi = (a_1, \varphi') \\ & \wedge \mathcal{A}[[e_2]]\varphi' = (a_2, \varphi'') \\ \uparrow & \text{otherwise} \end{cases}$$

$$\mathcal{B}[[e_1 \geq e_2]]\varphi = \begin{cases} (a_1 \geq a_2, \varphi'') & \mathcal{A}[[e_1]]\varphi = (a_1, \varphi') \\ & \wedge \mathcal{A}[[e_2]]\varphi' = (a_2, \varphi'') \\ \uparrow & \text{otherwise} \end{cases}$$

$\mathcal{B} : BExp \rightarrow State \hookrightarrow \mathbb{T} \times State$

$$\mathcal{B}[[e_1 > e_2]]\varphi = \begin{cases} (a_1 > a_2, \varphi'') & \mathcal{A}[[e_1]]\varphi = (a_1, \varphi') \\ & \wedge \mathcal{A}[[e_2]]\varphi' = (a_2, \varphi'') \\ \uparrow & \text{otherwise} \end{cases}$$

$$\mathcal{B}[[e_1 \leq e_2]]\varphi = \begin{cases} (a_1 \leq a_2, \varphi'') & \mathcal{A}[[e_1]]\varphi = (a_1, \varphi') \\ & \wedge \mathcal{A}[[e_2]]\varphi' = (a_2, \varphi'') \\ \uparrow & \text{otherwise} \end{cases}$$

## Rule

Since boolean expressions induce state transitions, the evaluation order and quantity must be preserved in the desugared code.

This is the reason why we couldn't model the operators  $(\cdot > \cdot)$ ,  $(\cdot \leq \cdot) : \mathbb{N} \rightarrow \mathbb{N} \rightarrow \mathbb{T}$  as syntactic sugar. There is no way to encode those operators only with  $(\cdot < \cdot)$ ,  $(\cdot \geq \cdot)$ ,  $(\cdot = \cdot)$  and  $(\cdot \neq \cdot)$  respecting this rule.

$$\text{not true} \stackrel{\text{def}}{=} \text{false}$$

$$\text{not false} \stackrel{\text{def}}{=} \text{true}$$

$$\text{not } (b_1 \text{ and } b_2) \stackrel{\text{def}}{=} \text{not } b_1 \text{ or not } b_2$$

$$\text{not } (b_1 \text{ or } b_2) \stackrel{\text{def}}{=} \text{not } b_1 \text{ and not } b_2$$

$$\text{not } e_1 = e_2 \stackrel{\text{def}}{=} e_1 \neq e_2$$

$$\text{not } e_1 \neq e_2 \stackrel{\text{def}}{=} e_1 = e_2$$

$$\text{not } e_1 < e_2 \stackrel{\text{def}}{=} e_1 \geq e_2$$

$$\text{not } e_1 \geq e_2 \stackrel{\text{def}}{=} e_1 < e_2$$

$$\text{not } e_1 > e_2 \stackrel{\text{def}}{=} e_1 \leq e_2$$

$$\text{not } e_1 \leq e_2 \stackrel{\text{def}}{=} e_1 > e_2$$

**While** ::=  $x := e$  | skip |  $\{S\}$  |  $S_1 ; S_2$   
| if  $b$  then  $S_1$  else  $S_2$  | while  $b$  do  $S$

$\mathcal{S}_{ds} : \text{While} \rightarrow \text{State} \hookrightarrow \text{State}$

$$\mathcal{S}_{ds}[\![x := e]\!] \varphi = \begin{cases} \varphi'[x \mapsto a] & \mathcal{A}[\![e]\!] \varphi = (a, \varphi') \\ \uparrow & \text{otherwise} \end{cases}$$

$$\mathcal{S}_{ds}[\![\text{skip}]\!] \varphi = \varphi$$

$$\mathcal{S}_{ds}[\![\{S\}]\!] \varphi = \mathcal{S}_{ds}[\![S]\!] \varphi$$



$\mathcal{S}_{ds} : \text{While} \rightarrow \text{State} \hookrightarrow \text{State}$

$$\mathcal{S}_{ds} \llbracket S_1 ; S_2 \rrbracket \varphi = (\mathcal{S}_{ds} \llbracket S_2 \rrbracket \circ \mathcal{S}_{ds} \llbracket S_1 \rrbracket) \varphi$$

$$\mathcal{S}_{ds} \llbracket \text{if } b \text{ then } S_1 \text{ else } S_2 \rrbracket \varphi = \text{cond}(\mathcal{B} \llbracket b \rrbracket, \mathcal{S}_{ds} \llbracket S_1 \rrbracket, \mathcal{S}_{ds} \llbracket S_2 \rrbracket)$$

$$\mathcal{S}_{ds} \llbracket \text{while } b \text{ do } S \rrbracket \varphi = \text{FIX}(\lambda g. \text{cond}(\mathcal{B} \llbracket b \rrbracket, g \circ \mathcal{S}_{ds} \llbracket S \rrbracket, \text{id}))$$

Where

$$\text{cond}(\text{pred}, g_1, g_2) = \begin{cases} g_1(\varphi') & \text{pred}(\varphi) = (\mathbf{tt}, \varphi') \\ g_2(\varphi') & \text{pred}(\varphi) = (\mathbf{ff}, \varphi') \\ \uparrow & \text{otherwise} \end{cases}$$

We define for any abstract domain  $A$ , which is a complete lattice as well, the abstract state type  $\mathbb{S}_A = \text{Map}(\text{Var}, A)$ .

## Assumption

When a variable is used before its definition, then its value is assumed to be “unknown” ( $\top_A$ ). This is due to the fact that we assume that all referenced variables in the program are initialized.

Moreover,  $\perp_{\mathbb{S}_A}$  represents an abnormal termination (no update operation can be performed over this state):

$$s(x) = \begin{cases} a & (x, a) \in s \\ \top_A & \text{otherwise} \end{cases}$$
$$s[x \mapsto a] = \begin{cases} \perp_{\mathbb{S}_A} & s = \perp_{\mathbb{S}_A} \\ \{(k, v) \mid (k, v) \in s, k \neq x\} & a \neq \top_A, s \neq \perp_{\mathbb{S}_A} \\ \{(k, v) \mid (k, v) \in s, k \neq x\} & \text{otherwise} \end{cases}$$

$\mathbb{S}_A$  is partially ordered

$$s_1 \leq_{\mathbb{S}_A} s_2 \iff s_1(x) \leq_A s_2(x) \quad \forall x \in Var$$

$\mathbb{S}_A$  is a complete lattice

$$\perp_{\mathbb{S}_A} = \{(x, \perp_A) \mid x \in Var\}$$

$$\top_{\mathbb{S}_A} = \emptyset$$

$$s_1 \vee_{\mathbb{S}_A} s_2 = \{(var, a_1 \vee_A a_2) \mid (var, a_1) \in s_1, (var, a_2) \in s_2\}$$

$$s_1 \wedge_{\mathbb{S}_A} s_2 = \{(var, a_1 \wedge_A a_2) \mid (var, a_1) \in s_1, (var, a_2) \in s_2\} \\ \cup \{e \mid e \in s_1, e \notin s_2\} \cup \{e \mid e \notin s_1, e \in s_2\}$$

The abstract semantic functions are:

- $\mathcal{A}^\sharp : AExp \rightarrow \mathbb{S}_A \rightarrow A \times \mathbb{S}_A$ :
  - the first element of the tuple approximates the possible results of the arithmetic expression;
  - the second element approximates the possible states after the transition induced by the expression;
- $\mathcal{B}^\sharp : BExp \rightarrow \mathbb{S}_A \rightarrow \mathbb{S}_A \times \mathbb{S}_A$ :
  - the first element of the tuple approximates the states where the boolean expression can evaluate to **tt**;
  - the second element approximates the states where the boolean expression can evaluate **ff**.
- $\mathcal{D}^\sharp : While \rightarrow \mathbb{S}_A \rightarrow \mathbb{S}_A$ .

$\mathcal{D}^\# : \text{While} \rightarrow \mathbb{S}_A \rightarrow \mathbb{S}_A$

$$\mathcal{D}^\# \llbracket x := e \rrbracket s^\# \stackrel{\text{def}}{=} \begin{cases} s'^\# [x \mapsto a] & (a, s'^\#) = \mathcal{A}^\# \llbracket e \rrbracket s^\# \\ & \wedge a \neq \perp_A \\ \perp_{\mathbb{S}_A} & \text{otherwise} \end{cases}$$

$$\mathcal{D}^\# \llbracket \text{skip} \rrbracket s^\# \stackrel{\text{def}}{=} s^\#$$

$$\mathcal{D}^\# \llbracket S_1 ; S_2 \rrbracket s^\# \stackrel{\text{def}}{=} (\mathcal{D}^\# \llbracket S_1 \rrbracket \circ \mathcal{D}^\# \llbracket S_2 \rrbracket) s^\#$$

$\mathcal{D}^\# : While \rightarrow \mathbb{S}_A \rightarrow \mathbb{S}_A$

$$\mathcal{D}^\# \llbracket \text{if } b \text{ then } S_1 \text{ else } S_2 \rrbracket s^\# \stackrel{\text{def}}{=}$$
$$\mathcal{D}^\# \llbracket \text{while } b \text{ do } S \rrbracket s^\# \stackrel{\text{def}}{=}$$

$$I_{m,n} \subset \wp(\mathbb{Z}) \text{ with } m, n \in \mathbb{Z} \cup -\infty, \infty$$

$$\begin{aligned} I_{m,n} = & \{\mathbb{Z}, \emptyset\} \cup \{\{z\} \mid z \in \mathbb{Z}\} \\ & \cup \{\{x \mid w \leq x \leq z\} \mid x, w, z \in \mathbb{Z} \text{ s.t. } m \leq w \leq z \leq n\} \\ & \cup \{\{x \mid x \leq z\} \mid x, z \in \mathbb{Z} \text{ s.t. } m \leq z \leq n\} \\ & \cup \{\{x \mid x \geq z\} \mid x, z \in \mathbb{Z} \text{ s.t. } m \leq z \leq n\} \end{aligned}$$



# Bounded Intervals - Properties (1)



$I_{m,n}$  is partially ordered

$$i_1 \leq i_2 \iff i_1 \subseteq i_2$$

$I_{m,n}$  is a complete lattice

$$\perp_{I_{m,n}} = \emptyset$$

$$\top_{I_{m,n}} = \mathbb{Z}$$

$$\vee_{I_{m,n}} = \cup$$

$$\wedge_{I_{m,n}} = \cap$$

- $I_{m,n}$  has no infinite ascending chains when  $m \neq -\infty \wedge n \neq \infty$ :
  - when  $m, n \in \mathbb{N}$  the fixed-point iteration sequence induced by  $\mathcal{D}^\# \llbracket S_1 \rrbracket$  converges in finite time;
  - otherwise, we must make use of the widening operator  $\nabla : \mathbb{S}_A \rightarrow \mathbb{S}_A \rightarrow \mathbb{S}_A$  in order to enforce convergence.
- $I_{m,n}$  has no infinite descending chains:
  - any descending greatest fixed-point search converges in finite time;
  - there is no need for a narrowing operator  $\Delta : \mathbb{S}_A \rightarrow \mathbb{S}_A \rightarrow \mathbb{S}_A$ .

Etiam eu interdum ligula

Nunc mi eros, vulputate in ornare a, viverra eget quam

- Morbi **vitae lacus** porta neque tincidunt sodales
- Proin tincidunt, **neque** at tincidunt mollis
- Ut **lacinia sem a nibh** consequat porttitor

## Normal block

Fusce luctus venenatis felis quis semper

## Alert block

$$E = (x_1 \vee \neg x_2 \vee \neg x_3) \wedge (x_1 \vee x_2 \vee x_4)$$

## Example block

Proin tincidunt, neque at tincidunt mollis