Abstract interpretation with bounded numeric intervals

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

Christian Micheletti

January 7, 2024

Università degli Studi di Padova

Outline

1 The Language
- Arithmetic Expressions
- Boolean Expressions

2 Abstract Interpreter
- Abstract States
- Abstract Semantics

3 Implementation
- The Bounded Intervals Domain
- Usage

4 Introduction

5 First section

The Language

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.

Arithmetic Expressions - Syntax

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

The syntax allows arithmetic expression that change the state, such as `x++` and `x--`.
The operator $(\cdot/\cdot) : \mathbb{N} \to \mathbb{N} \to \mathbb{N}$ returns the quotient of the two arguments. It's undefined when the second argument is 0.

Arithmetic Expressions - Semantics (1)

Università degli Studi di Padova

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

$$
\begin{aligned}
\mathcal{A}[\![n]\!]\varphi =& (n_{\mathbb{Z}}, \varphi) \\
\mathcal{A}[\![x]\!]\varphi =& (\varphi(x), \varphi) \\
\mathcal{A}[\![(e)]\!]\varphi =& \mathcal{A}[\![e]\!]\varphi \\
\mathcal{A}[\![-e]\!]\varphi =& \begin{cases} (-a, \varphi') & \mathcal{A}[\![e]\!]\varphi = (a, \varphi') \\ \uparrow & (\mathcal{A}[\![e]\!]\varphi) \uparrow \end{cases}
\end{aligned}
$$

Arithmetic Expressions - Semantics (2)

Università degli Studi di Padova

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

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

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

Arithmetic Expressions - Semantics (3)

Università degli Studi di Padova

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

$$
\begin{aligned}
\mathcal{A}[\![x{+}{+}]\!]\varphi =& (\varphi(x), \varphi[x \mapsto x + 1]) \\
\mathcal{A}[\![{+}{+}x]\!]\varphi =& \mathit{let}\ \varphi' = \varphi[x \mapsto x + 1] \\
& \mathit{in}\ (\varphi'(x), \varphi') \\
\mathcal{A}[\![x{-}{-}]\!]\varphi =& (\varphi(x), \varphi[x \mapsto x - 1]) \\
\mathcal{A}[\![{-}{-}x]\!]\varphi =& \mathit{let}\ \varphi' = \varphi[x \mapsto x - 1] \\
& \mathit{in}\ (\varphi'(x), \varphi')
\end{aligned}
$$

Boolean Expressions - Syntax

$$
\begin{aligned}
\mathit{BExp} ::= & \, \texttt{true} \mid \texttt{false} \mid (b) \mid b_1 \texttt{ and } b_2 \mid b_1 \texttt{ or } b_2 \\
& \mid e_1 \texttt{ = } e_2 \mid e_1 \texttt{ != } e_2 \mid e_1 \texttt{ < } e_2 \mid e_1 \texttt{ >= } e_2 \\
& \mid e_1 \texttt{ > } e_2 \mid e_1 \texttt{ <= } e_2
\end{aligned}
$$

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

Università degli Studi di Padova

Operators between booleans short-circuit evaluation:

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

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

Boolean Expressions - Semantics (2)

Università degli Studi di Padova

Comparison operators propagate the state transition(s):

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

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

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

Boolean Expressions - Semantics (3)

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

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

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

Boolean Expressions - Semantics (4)

Università degli Studi di Padova

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

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

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

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 $(\cdot > \cdot), (\cdot \leq \cdot) : \mathbb{N} \to \mathbb{N} \to \mathbb{T}$ as syntactic sugar. There is no way to encode those operators only with $(\cdot < \cdot), (\cdot >= \cdot), (\cdot = \cdot)$ and $(\cdot ! = \cdot)$ respecting this rule.

Boolean Expressions - Negation

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

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

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

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

$$\texttt{not}\ e_1\ \texttt{=}\ e_2 \stackrel{\text{def}}{=} e_1\ \texttt{!=}\ e_2$$

$$\texttt{not}\ e_1\ \texttt{!=}\ e_2 \stackrel{\text{def}}{=} e_1\ \texttt{=}\ e_2$$

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

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

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

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

Statements - Syntax

$$
\begin{aligned}
\textbf{While} ::= &\ x\ \texttt{:=}\ e \mid \texttt{skip} \mid \{S\} \mid S_1\ \texttt{;}\ S_2 \\
&\mid \texttt{if}\ b\ \texttt{then}\ S_1\ \texttt{else}\ S_2 \mid \texttt{while}\ b\ \texttt{do}\ S
\end{aligned}
$$

Statements - Semantics (1)

$\mathcal{S}_{ds} : \mathit{While} \to \mathit{State} \hookrightarrow \mathit{State}$

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

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

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

Statements - Semantics (2)

$\mathcal{S}_{ds} : \mathit{While} \to \mathit{State} \hookrightarrow \mathit{State}$

$$
\begin{aligned}
\mathcal{S}_{ds}[\![S_1\ ;\ S_2]\!]\varphi &= (\mathcal{S}_{ds}[\![S_2]\!] \circ \mathcal{S}_{ds}[\![S_1]\!])\varphi \\
\mathcal{S}_{ds}[\![\texttt{if}\ b\ \texttt{then}\ S_1\ \texttt{else}\ S_2]\!]\varphi &= \mathit{cond}(\mathcal{B}[\![b]\!], \mathcal{S}_{ds}[\![S_1]\!], \mathcal{S}_{ds}[\![S_2]\!]) \\
\mathcal{S}_{ds}[\![\texttt{while}\ b\ \texttt{do}\ S]\!]\varphi &= \mathsf{FIX}(\lambda g.\mathit{cond}(\mathcal{B}[\![b]\!], g \circ \mathcal{S}_{ds}[\![S]\!], \mathit{id}))
\end{aligned}
$$

Where

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

Abstract States (1)

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

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.

Abstract States (2)

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

$$s(x) = \begin{cases} a & (x, a) \in s \\ \top_A & \textit{otherwise} \end{cases}$$

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

Abstract States (3)

Università degli Studi di Padova

$\mathbb{S}_A$ is partially ordered

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

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

$$\begin{aligned}
\perp_{\mathbb{S}_A} &= \{(x, \perp_A) \mid x \in Var\} \\
\top_{\mathbb{S}_A} &= \emptyset \\
s_1 \vee_{\mathbb{S}_A} s_2 &= \{(var, a_1 \vee_A a_2) \mid (var, a_1) \in s_1, (var, a_2) \in s_2\} \\
s_1 \wedge_{\mathbb{S}_A} s_2 &= \{(var, a_1 \wedge_A a_2) \mid (var, a_1) \in s_1, (var, a_2) \in s_2\} \\
&\quad \cup \{e \mid e \in s_1, e \notin s_2\} \cup \{e \mid e \notin s_1, e \in s_2\}
\end{aligned}$$

Abstract Semantics

The abstract semantic functions are:

- $\mathcal{A}^\sharp : AExp \to \mathbb{S}_A \to A \times \mathbb{S}_A$:
  - 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;
- $\mathcal{B}^\sharp : BExp \to \mathbb{S}_A \to \mathbb{S}_A \times \mathbb{S}_A$;
  - 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**.

  This function returns two states, instead of one, in order to preserve the short circuit behavior of boolean operators along with a compositional definition.
- $\mathcal{D}^\sharp : While \to \mathbb{S}_A \to \mathbb{S}_A$.

Abstract Semantics - Statements (1)

Università degli Studi di Padova

$\mathcal{D}^\sharp : \mathit{While} \to \mathbb{S}_A \to \mathbb{S}_A$

$$\mathcal{D}^\sharp[\![x \;\; \texttt{:=} \;\; e]\!]s^\sharp \stackrel{\text{def}}{=} \begin{cases} s'^\sharp[x \mapsto a] & (a, s'^\sharp) = \mathcal{A}^\sharp[\![e]\!]s^\sharp \\ & \wedge a \neq \bot_A \\ \bot_{\mathbb{S}_A} & \mathit{otherwise} \end{cases}$$

$$\mathcal{D}^\sharp[\![\texttt{skip}]\!]s^\sharp \stackrel{\text{def}}{=} s^\sharp$$

$$\mathcal{D}^\sharp[\![S_1 \;\; ; \;\; S_2]\!]s^\sharp \stackrel{\text{def}}{=} (\mathcal{D}^\sharp[\![S_1]\!] \circ \mathcal{D}^\sharp[\![S_2]\!])s^\sharp$$

$\mathcal{D}^\sharp : \textit{While} \to \mathbb{S}_A \to \mathbb{S}_A$

$$
\begin{aligned}
\mathcal{D}^\sharp[\![\texttt{if } b \texttt{ then } S_1 \texttt{ else } S_2]\!]s^\sharp &\stackrel{\text{def}}{=} (\mathcal{B}^\sharp[\![S_1]\!]s^\sharp_{\mathbf{tt}}) \vee_{\mathbb{S}_A} (\mathcal{B}^\sharp[\![S_2]\!]s^\sharp_{\mathbf{ff}}) \\
\textit{where} \quad & (s^\sharp_{\mathbf{tt}}, s^\sharp_{\mathbf{ff}}) = \mathcal{B}^\sharp[\![b]\!]s^\sharp \\
\mathcal{D}^\sharp[\![\texttt{while } b \texttt{ do } S]\!]s^\sharp &\stackrel{\text{def}}{=} \pi_2(\mathcal{B}^\sharp[\![b]\!](\mathsf{GFP}_{\mathsf{FIX}\ F}(\lambda s.s \wedge_{\mathbb{S}_A} F\ s))) \\
\textit{where} \quad & F : \mathbb{S}_A \to \mathbb{S}_A \\
& F\ s = s^\sharp \vee_{\mathbb{S}_A} (\mathcal{D}^\sharp[\![S]\!] \circ \pi_1 \circ \mathcal{B}^\sharp[\![b]\!]s)
\end{aligned}
$$

Where FIX F refers to the fixed point of the function F and $\mathsf{GFP}_s\ f$ is the greatest fixed point of f found starting from s.

Bounded Intervals - Definition

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

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

Bounded Intervals - Properties (1)

$I_{m,n}$ is partially ordered

$$i_1 \leq i_2 \Longleftrightarrow i_1 \subseteq i_2$$

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

$$\bot_{I_{m,n}} = \emptyset$$
$$\top_{I_{m,n}} = \mathbb{Z}$$
$$\vee_{I_{m,n}} = \cup$$
$$\wedge_{I_{m,n}} = \cap$$

Bounded Intervals - Properties (2)

- $I_{m,n}$ 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 $\mathcal{D}^{\#}[\![S_1]\!]s\ \forall s \in \mathbb{S}_A,\ \ S_1 \in \textbf{While}$ 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 $\Delta : \mathbb{S}_A \rightarrow \mathbb{S}_A \rightarrow \mathbb{S}_A$.

Usage

The program runs with the command

`$ cabal run ai --` *path/to/file.whl*

This command will read the file given as input and as output:

- it will output the invariant after the last program point;
- it will rewrite the input into a file called as the input plus `.inv`, with the invariants as comments at any program point.

Program Points

The program points are located along with the statements:

- the terminals $x{:=}e$ and `skip` are followed by one program point;
- the `then` and `else` sub-statements in the branch statement are preceded by one program point each;
- while statements are preceded by a program point, whose invariant is the loop invariant of that loop;
- the `do` sub-statement in the loop statement is preceded by one program point;
- while statements are followed by one program point, which is the invariant after the loop exit.

Examples (1)

Input

```
x := 0;
while x < 10 do {
    x := x + 2
}
```

Output

```
x := 0; // {"x": [0, 0]}
skip; // {"x": [0, 11]}
while x < 10 do {
skip; // {"x": [0, 9]}
x := (x + 2); // {"x": [2, 11]}
};
skip; // {"x": [10, 11]}
```

Examples (2)

Input

```
x := 10;
while x > 0 do x := x + 1;
y := 0
```

Output

```
x := 10; // {"x": [10, 10]}
skip; // {"x": [10, Inf]}
while x > 0 do {
skip; // {"x": [10, Inf]}
x := (x + 1); // {"x": [11, Inf]}
};
skip; // BOTTOM STATE
y := 0; // BOTTOM STATE
```

Examples (3)

Input

```
x := [-10, 10];
if x / 2 = x then y := x else y := 0
```

Output

```
x := [(-10), 10]; // {"x": [-10, 10]}
if (x / 2) = x then {
skip; // {"x": [-1, 0]}
y := x; // {"y": [-1, 0],"x": [-1, 0]}
} else {
skip; // {"x": [-10, 10]}
y := 0; // {"y": [0, 0],"x": [-10, 10]}
};
skip; // {"y": [-1, 0],"x": [-10, 10]}
```

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