Abstract interpretation with bounded numeric intervals

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

Christian Micheletti
January 20, 2024

Università degli Studi di Padova

Outline

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

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

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

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

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

$$
\begin{aligned}
\mathcal{A}[\![n]\!]\varphi &= (n_{\mathbb{Z}}, \varphi) \\
\mathcal{A}[\![x]\!]\varphi &= (\varphi(x), \varphi) \\
\mathcal{A}[\![(e)]\!]\varphi &= \mathcal{A}[\![e]\!]\varphi \\
\mathcal{A}[\![-e]\!]\varphi &= \begin{cases} (-a, \varphi') & \mathcal{A}[\![e]\!]\varphi = (a, \varphi') \\ \uparrow & (\mathcal{A}[\![e]\!]\varphi) \uparrow \end{cases} \\
\mathcal{A}[\![[e_1, e_2]]\!]\varphi &= (a, \varphi'') \\
&\quad \text{where } (a_1, \varphi') = \mathcal{A}[\![e_1]\!]\varphi \\
&\quad\quad (a_2, \varphi'') = \mathcal{A}[\![e_2]\!]\varphi' \\
&\quad\quad a \text{ is a random integer between } a_1 \text{ and } a_2
\end{aligned}
$$

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)

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

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

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 $(\neg\cdot) : \mathbb{T} \to \mathbb{T}$ is defined as syntactic sugar later on.

Boolean Expressions - Semantics (1)

Università degli Studi di Padova

Conjuntion and disjunction short-circuit evaluation:

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

$$
\begin{aligned}
\mathcal{B}[\![\texttt{true}]\!]\varphi &= (\mathbf{tt}, \varphi) \\
\mathcal{B}[\![\texttt{false}]\!]\varphi &= (\mathbf{ff}, \varphi) \\
\mathcal{B}[\![(b)]\!]\varphi &= \mathcal{B}[\![b]\!]\varphi
\end{aligned}
$$

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

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

Boolean Expressions - Semantics (2)

Comparison operators propagate the state transition(s):

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

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

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

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)

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

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

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

Boolean Expressions - Syntactic Sugar

Università degli Studi di Padova

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

$$
\begin{aligned}
\texttt{not true} &\stackrel{\text{def}}{=} \texttt{false} \\
\texttt{not false} &\stackrel{\text{def}}{=} \texttt{true} \\
\texttt{not}\ (b_1\ \texttt{and}\ b_2) &\stackrel{\text{def}}{=} \texttt{not}\ b_1\ \texttt{or not}\ b_2 \\
\texttt{not}\ (b_1\ \texttt{or}\ b_2) &\stackrel{\text{def}}{=} \texttt{not}\ b_1\ \texttt{and not}\ b_2 \\
\texttt{not}\ e_1\ \texttt{=}\ e_2 &\stackrel{\text{def}}{=} e_1\ \texttt{!=}\ e_2 \\
\texttt{not}\ e_1\ \texttt{!=}\ e_2 &\stackrel{\text{def}}{=} e_1\ \texttt{=}\ e_2 \\
\texttt{not}\ e_1\ \texttt{<}\ e_2 &\stackrel{\text{def}}{=} e_1\ \texttt{>=}\ e_2 \\
\texttt{not}\ e_1\ \texttt{>=}\ e_2 &\stackrel{\text{def}}{=} e_1\ \texttt{<}\ e_2 \\
\texttt{not}\ e_1\ \texttt{>}\ e_2 &\stackrel{\text{def}}{=} e_1\ \texttt{<=}\ e_2 \\
\texttt{not}\ e_1\ \texttt{<=}\ e_2 &\stackrel{\text{def}}{=} e_1\ \texttt{>}\ e_2
\end{aligned}
$$

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)

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

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

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

Where

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

Bounded Intervals - Definition

Università degli Studi di Padova

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

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

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

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

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 $\forall s \in \mathbb{S}_{I_{m,n}},\ S_1 \in \textbf{While}.\mathcal{D}^{\#}[\![S_1]\!]s$ converges in finite time;
  - otherwise, we must make use of the widening operator $\nabla : \mathbb{S}_{I_{m,n}} \to \mathbb{S}_{I_{m,n}} \to \mathbb{S}_{I_{m,n}}$ 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 $\Delta : \mathbb{S}_{I_{m,n}} \to \mathbb{S}_{I_{m,n}} \to \mathbb{S}_{I_{m,n}}$

Abstract States (1)

Università degli Studi di Padova

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

Assumption

We assume that all the variables referenced in the program have been initialized. Therefore, the value of a non initialized variable is assumed to be "unknown" ($\top_{I_{m,n}}$).

Abstract States (2)

Università degli Studi di Padova

$\mathbb{S}_{I_{m,n}}$ is partially ordered

$$s_1 \sqsubseteq_{\mathbb{S}_{I_{m,n}}} s_2 \iff \forall x \in Var\,.s_1(x) \sqsubseteq_{I_{m,n}} s_2(x)$$

$\mathbb{S}_{I_{m,n}}$ is a complete lattice

$$\begin{aligned}
\bot_{\mathbb{S}_{I_{m,n}}} &= \{(x, \bot_{I_{m,n}}) \mid x \in Var\} \\
\top_{\mathbb{S}_{I_{m,n}}} &= \emptyset \\
s_1 \vee_{\mathbb{S}_{I_{m,n}}} s_2 &= \{(var, a_1 \vee_{I_{m,n}} a_2) \mid (var, a_1) \in s_1, (var, a_2) \in s_2\} \\
s_1 \wedge_{\mathbb{S}_{I_{m,n}}} s_2 &= \{(var, a_1 \wedge_{I_{m,n}} a_2) \mid (var, a_1) \in s_1, (var, a_2) \in s_2\} \\
&\quad \cup \{e \mid e \in s_1, e \notin s_2\} \cup \{e \mid e \notin s_1, e \in s_2\}
\end{aligned}$$

Abstract States (3)

$\perp_{\mathbb{S}_{I_{m,n}}}$ represents an abnormal termination or sure divergence $\uparrow$. Hence, such a state is propagated through the remaining execution and no recover is possible:

$$s(x) = \begin{cases} a & (x, a) \in s \\ \top_{I_{m,n}} & otherwise \end{cases}$$

$$s[x \mapsto a] = \begin{cases} \perp_{\mathbb{S}_{I_{m,n}}} & s = \perp_{\mathbb{S}_{I_{m,n}}} \\ \{(k, v) \mid (k, v) \in s,\ k \neq x\} & a \neq \top_{I_{m,n}},\ s \neq \perp_{\mathbb{S}_{I_{m,n}}} \\ \{(k, v) \mid (k, v) \in s,\ k \neq x\} & otherwise \end{cases}$$

Abstract Semantics

The abstract semantic functions are:

- $\mathcal{A}^\sharp : \mathit{AExp} \to \mathbb{S}_{I_{m,n}} \to A \times \mathbb{S}_{I_{m,n}}$
  - 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;
- $\mathcal{B}^\sharp : \mathit{BExp} \to \mathbb{S}_{I_{m,n}} \to \mathbb{S}_{I_{m,n}} \times \mathbb{S}_{I_{m,n}}$
  - 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.
- $\mathcal{D}^\sharp : \mathit{While} \to \mathbb{S}_{I_{m,n}} \to \mathbb{S}_{I_{m,n}}$.

Abstract Semantics - AExp (1)

$\mathcal{A}^\sharp : AExp \to \mathbb{S}_{I_{m,n}} \to I_{m,n} \times \mathbb{S}_{I_{m,n}}$

$$
\begin{aligned}
\mathcal{A}^\sharp[\![n]\!]s^\sharp &= (\{n_\mathbb{Z}\}, s^\sharp) \\
\mathcal{A}^\sharp[\![x]\!]s^\sharp &= (s^\sharp(x), s^\sharp) \\
\mathcal{A}^\sharp[\![(e)]\!]s^\sharp &= \mathcal{A}^\sharp[\![e]\!]s^\sharp \\
\mathcal{A}^\sharp[\![-e]\!]s^\sharp &= (-a^\sharp, s_1^\sharp) \\
&\quad \text{where } (a^\sharp, s_1^\sharp) = \mathcal{A}^\sharp[\![e]\!]s^\sharp \\
\mathcal{A}^\sharp[\![[e_1, e_2]]\!]s^\sharp &= (\{a_1^\sharp, \ldots, a_2^\sharp\}, s_2^\sharp) \\
&\quad \text{where } (a_1^\sharp, s_1^\sharp) = \mathcal{A}^\sharp[\![e_1]\!]s^\sharp \\
&\quad\phantom{\text{where }} (a_2^\sharp, s_2^\sharp) = \mathcal{A}^\sharp[\![e_2]\!]s_1^\sharp
\end{aligned}
$$

Abstract Semantics - AExp (2)

$\mathcal{A}^\sharp : AExp \to \mathbb{S}_{I_{m,n}} \to I_{m,n} \times \mathbb{S}_{I_{m,n}}$

$$
\begin{aligned}
\mathcal{A}^\sharp[\![e_1 \ \mathbf{op} \ e_2]\!]s^\sharp = &(a_1^\sharp \ \ op_{I_{m,n}} \ \ a_2^\sharp, s_2^\sharp) \\
\text{where } &(a_1^\sharp, s_1^\sharp) = \mathcal{A}^\sharp[\![e_1]\!]s^\sharp \\
&(a_2^\sharp, s_2^\sharp) = \mathcal{A}^\sharp[\![e_2]\!]s_1^\sharp
\end{aligned}
$$

$\mathcal{A}^\sharp : AExp \to \mathbb{S}_{I_{m,n}} \to I_{m,n} \times \mathbb{S}_{I_{m,n}}$

$$
\begin{aligned}
\mathcal{A}^\sharp[\![x\texttt{++}]\!]s^\sharp =& (s^\sharp(x), s^\sharp[x \mapsto x +_{I_{m,n}} 1]) \\
\mathcal{A}^\sharp[\![\texttt{++}x]\!]s^\sharp =& (s_1^\sharp(x), s_1^\sharp) \\
& \text{where } s^\sharp[x \mapsto x +_{I_{m,n}} 1] = s_1^\sharp \\
\mathcal{A}^\sharp[\![x\texttt{--}]\!]s^\sharp =& (s^\sharp(x), s^\sharp[x \mapsto x -_{I_{m,n}} 1]) \\
\mathcal{A}^\sharp[\![\texttt{--}x]\!]s^\sharp =& (s_1^\sharp(x), s_1^\sharp) \\
& \text{where } s^\sharp[x \mapsto x -_{I_{m,n}} 1] = s_1^\sharp
\end{aligned}
$$

Abstract Semantics - BExp (1)

Università degli Studi di Padova

$\mathcal{B}^\sharp : BExp \to \mathbb{S}_{I_{m,n}} \to \mathbb{S}_{I_{m,n}} \times \mathbb{S}_{I_{m,n}}$

$$
\begin{aligned}
\mathcal{B}^\sharp[\![\texttt{true}]\!]s^\sharp =& (s^\sharp, \bot_{\mathcal{S}_{I_{m,n}}})\\
\mathcal{B}^\sharp[\![\texttt{false}]\!]s^\sharp =& (\bot_{\mathcal{S}_{I_{m,n}}}, s^\sharp)\\
\mathcal{B}^\sharp[\![(b)]\!]s^\sharp =& \mathcal{B}^\sharp[\![b]\!]s^\sharp\\
\mathcal{B}^\sharp[\![b_1 \texttt{ and } b_2]\!]s^\sharp =& (s_2^{\sharp(then)}, s_1^{\sharp(else)} \vee_{\mathbb{S}_{I_{m,n}}} s_2^{\sharp(else)})\\
& \text{where } (s_1^{\sharp(then)}, s_1^{\sharp(else)}) = \mathcal{B}^\sharp[\![b_1]\!]s^\sharp\\
& (s_2^{\sharp(then)}, s_2^{\sharp(else)}) = \mathcal{B}^\sharp[\![b_2]\!]s_1^{\sharp(then)}\\
\mathcal{B}^\sharp[\![b_1 \texttt{ or } b_2]\!]s^\sharp =& (s_1^{\sharp(then)} \vee_{\mathbb{S}_{I_{m,n}}} s_2^{\sharp(then)}, s_2^{\sharp(else)})\\
& \text{where } (s_1^{\sharp(then)}, s_1^{\sharp(else)}) = \mathcal{B}^\sharp[\![b_1]\!]s^\sharp\\
& (s_2^{\sharp(then)}, s_2^{\sharp(else)}) = \mathcal{B}^\sharp[\![b_2]\!]s_1^{\sharp(else)}
\end{aligned}
$$

Abstract Semantics - BExp (2)

Università degli Studi di Padova

$\mathcal{B}^\sharp : BExp \to \mathbb{S}_{I_{m,n}} \to \mathbb{S}_{I_{m,n}} \times \mathbb{S}_{I_{m,n}}$

$$\mathcal{B}^\sharp[\![e_1 \texttt{ = } e_2]\!]s^\sharp = \begin{cases} (\bot_{\mathbb{S}_{I_{m,n}}}, \bot_{\mathbb{S}_{I_{m,n}}}) & a_1 = \bot_{I_{m,n}} \vee a_2 = \bot_{I_{m,n}} \\ (\bot_{\mathbb{S}_{I_{m,n}}}, s_2^\sharp) & a_1 \wedge_{I_{m,n}} a_2 = \bot_{I_{m,n}} \\ (s_2^\sharp, \bot_{\mathbb{S}_{I_{m,n}}}) & a_1 = a_2 \wedge |a_1| = |a_2| = 1 \\ (trans(s^{\sharp(=)}), trans(s)) & otherwise \end{cases}$$

$$\text{where } (a_1, s_1^\sharp) = \mathcal{A}^\sharp[\![e_1]\!]s^\sharp$$

$$(a_2, s_2^\sharp) = \mathcal{A}^\sharp[\![e_2]\!]s_1^\sharp$$

$$trans = \pi_2 \circ \mathcal{A}^\sharp[\![e_2]\!] \circ \pi_2 \circ \mathcal{A}^\sharp[\![e_1]\!]$$

$$\mathcal{B}^\sharp[\![e_1 \texttt{ != } e_2]\!]s^\sharp = (s_2^\sharp, s_1^\sharp)$$

$$\text{where } (s_1^\sharp, s_2^\sharp) = \mathcal{B}^\sharp[\![e_1 \texttt{ = } e_2]\!]s^\sharp$$

Variable refinements

When one or both arithmetic expressions are variable, then we can be more precise:

$$
\begin{aligned}
s^{\sharp(=)} &= \mathrm{GFP}_{s^\sharp} f \\
&\text{where } f(s) = s[x \mapsto a_1 \wedge_{I_{m,n}} a_2] \\
&\qquad \forall\,(\mathrm{Var}\ x) \in \{e_1, e_2\}, (a_1, s_1) = \mathcal{A}^\sharp[\![e_1]\!]s, (a_2, \_) = \mathcal{A}^\sharp[\![e_2]\!]s_1
\end{aligned}
$$

Since f is descending monotone (it changes the abstract values with the $\wedge_{I_{m,n}}$ operator of up to two variables) and $I_{m,n}$ has no infinite descending chains, the GFP of f converges in finite time.

Abstract Semantics - BExp (4)

$\mathcal{B}^\sharp : BExp \rightarrow \mathbb{S}_{I_{m,n}} \rightarrow \mathbb{S}_{I_{m,n}} \times \mathbb{S}_{I_{m,n}}$

$$\mathcal{B}^\sharp[\![e_1 \ < \ e_2]\!]s^\sharp = \begin{cases} (\bot_{\mathbb{S}_{I_{m,n}}}, \bot_{\mathbb{S}_{I_{m,n}}}) & a_1 = \bot_{I_{m,n}} \vee a_2 = \bot_{I_{m,n}} \\ (\bot_{\mathbb{S}_{I_{m,n}}}, s_2^\sharp) & a_1 <_{I_{m,n}} a_2 \\ (s_2^\sharp, \bot_{\mathbb{S}_{I_{m,n}}}) & a_1 \geq_{I_{m,n}} a_2 \\ (trans(s^{\sharp(<)}), trans(s^{\sharp(\geq)})) & otherwise \end{cases}$$

$$\text{where } (a_1, s_1^\sharp) = \mathcal{A}^\sharp[\![e_1]\!]s^\sharp$$

$$(a_2, s_2^\sharp) = \mathcal{A}^\sharp[\![e_2]\!]s_1^\sharp$$

$$trans = \pi_2 \circ \mathcal{A}^\sharp[\![e_2]\!] \circ \pi_2 \circ \mathcal{A}^\sharp[\![e_1]\!]$$

$$\mathcal{B}^\sharp[\![e_1 \ \texttt{>=} \ e_2]\!]s^\sharp = (s_2^\sharp, s_1^\sharp)$$

$$\text{where } (s_1^\sharp, s_2^\sharp) = \mathcal{B}^\sharp[\![e_1 \ < \ e_2]\!]s^\sharp$$

Abstract Semantics - BExp (5)

$\mathcal{B}^\sharp : BExp \to \mathbb{S}_{I_{m,n}} \to \mathbb{S}_{I_{m,n}} \times \mathbb{S}_{I_{m,n}}$

$$\mathcal{B}^\sharp[\![e_1 \texttt{ > } e_2]\!]s^\sharp = \begin{cases} (\bot_{\mathbb{S}_{I_{m,n}}}, \bot_{\mathbb{S}_{I_{m,n}}}) & a_1 = \bot_{I_{m,n}} \vee a_2 = \bot_{I_{m,n}} \\ (\bot_{\mathbb{S}_{I_{m,n}}}, s_2^\sharp) & a_1 >_{I_{m,n}} a_2 \\ (s_2^\sharp, \bot_{\mathbb{S}_{I_{m,n}}}) & a_1 \leq_{I_{m,n}} a_2 \\ (trans(s^{\sharp(>)}), trans(s^{\sharp(\leq)})) & otherwise \end{cases}$$

$$\text{where } (a_1, s_1^\sharp) = \mathcal{A}^\sharp[\![e_1]\!]s^\sharp$$

$$(a_2, s_2^\sharp) = \mathcal{A}^\sharp[\![e_2]\!]s_1^\sharp$$

$$trans = \pi_2 \circ \mathcal{A}^\sharp[\![e_2]\!] \circ \pi_2 \circ \mathcal{A}^\sharp[\![e_1]\!]$$

$$\mathcal{B}^\sharp[\![e_1 \texttt{ <= } e_2]\!]s^\sharp = (s_2^\sharp, s_1^\sharp)$$

$$\text{where } (s_1^\sharp, s_2^\sharp) = \mathcal{B}^\sharp[\![e_1 \texttt{ > } e_2]\!]s^\sharp$$

Abstract Semantics - Statements (1)

Università degli Studi di Padova

$\mathcal{D}^\sharp$: While $\to \mathbb{S}_{I_{m,n}} \to \mathbb{S}_{I_{m,n}}$

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

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

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

Abstract Semantics - Statements (2)

$\mathcal{D}^\sharp$: While $\to \mathbb{S}_{I_{m,n}} \to \mathbb{S}_{I_{m,n}}$

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

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

Abstract Semantics - Statements (3)

Widened invariant refinition

Since $I_{m,n}$ has infinitely ascending chains, FIX F might diverge.
Therefore, in the implementation, we make use of a widened iteration sequence.
The **widened invariant** resulting from (possibly) widened FIX F is later refined with the GFP of $\lambda s.s \wedge_{\mathbb{S}_{I_{m,n}}} F\, s$.

This is sound:

- the widened invariant $s^{*\sharp}$ is a sound over-approximation of the smallest loop invariant s^*;
- $s^* = F\ s^*$, so $s^* = s^* \wedge_{\mathbb{S}_{I_{m,n}}} F\ s^*$: therefore $\lambda s.s \wedge_{\mathbb{S}_{I_{m,n}}} F\, s$ (descending monotone) is a sound filtering of those states not in s^*.

Therefore, GFP F starting from $s^{*\sharp}$ is the most precise refinement of the widened invariant.

Usage

The program runs with the command

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

This command will read the file given as input and:

- it will output the invariant after the last program point;
- it will rewrite the input into a file called just 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]}
```