EC3204: Programming Languages and Compilers

Lecture 11 — Semantic Analysis (2)

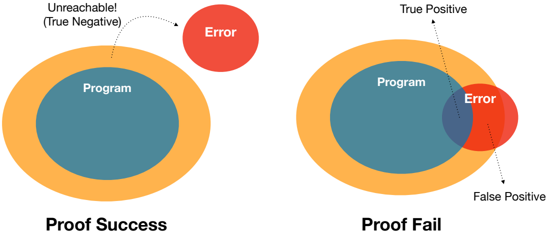
Introduction to Abstract Interpretation

Sunbeom So

Fall 2024

Overview

- **Abstract Interpretation** (AI) is a cost-effective analysis technology.
- Many useful static (compile-time) analyzers are based on AI.
  - Infer (Meta): a tool for detecting memory leaks in Android applications.
  - Astrée (Airbus): a static analyzer for aircraft software.
- **Key idea**: compute over-approximations (safe approximations).
  - "safe": the analysis result describes all possible runs of the program.
- AI can prove correctness by obtaining safe approximations.
  - **Yes** (proof success): the given program is guaranteed to be safe.
  - **No** (proof fail): the given program **may** contain errors.



Abstract Interpretation Recipe

To use abstract interpretation, follow the steps below.

1. **Abstract Domain** : define the abstract values that each variable can have (i.e., fixes "shape" of the invariants).
   - $c_1 \leq x \leq c_2$ (interval), $\pm x \pm y \leq c$ (octagon)
2. **Abstract Semantics** (abstract transformers): define how to execute each statement in the chosen abstract domain.

Concrete Semantics		**Abstract Semantics**
$[x \mapsto 7]$		$[x \mapsto \mathbf{odd}]$
$\downarrow$		$\downarrow$
$F(\boxed{\texttt{x := x+2;}})$	**VS.**	$\widehat{F}(\boxed{\texttt{x := x+2;}})$
$\downarrow$		$\downarrow$
$[x \mapsto 9]$		$[x \mapsto \mathbf{odd}]$

3. Run the analysis, i.e., execute the program using **abstract semantics**.

Setting: Simple Imperative Language

We will design an abstract interpreter for the simple language.

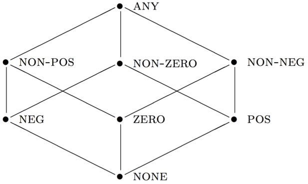
$$
\begin{array}{rcl}
a & \rightarrow & n \mid x \mid a_1 + a_2 \mid a_1 \star a_2 \mid a_1 - a_2 \\
b & \rightarrow & \texttt{true} \mid \texttt{false} \mid a_1 = a_2 \mid a_1 \leq a_2 \mid \neg b \mid b_1 \wedge b_2 \\
c & \rightarrow & x := a \mid \texttt{skip} \mid c_1; c_2 \mid \texttt{if}\ b\ c_1\ c_2 \mid \texttt{while}\ b\ c
\end{array}
$$

Step 1: Abstract Domain (Abstract Integers)

- Suppose we aim to infer invariants of the form $x \prec 0$ where $\prec \in \{>, \geq, <, \leq, =, \neq\}$.
- The abstract domain is defined as a pair $(\mathbf{Sign}, \sqsubseteq)$:

$$\mathbf{Sign} = \{\top, \bot, \text{Pos}, \text{Neg}, \text{Zero}, \text{Non-Pos}, \text{Non-Neg}, \text{Non-Zero}\}$$

where $\top$=ANY, $\bot$=NONE, and the partial order ($\sqsubseteq$) is defined as:



Intuitively, $a \sqsubseteq b$ indicates b contains more information.

- Pos $\sqsubseteq$ Non-Zero holds since $\{z \in \mathbb{Z} \mid z > 0\} \subseteq \{z \in \mathbb{Z} \mid z \neq 0\}$.

Step 1: Abstract Domain (Abstract Integers)

- **Important Requirement**: $(D, \sqsubseteq)$ must be a **complete lattice**.
- A partially ordered set (poset) $(D, \sqsubseteq)$ is a **complete lattice**, iff every subset $Y \subseteq D$ has $\bigsqcup Y \in D$ (the least upper bound of Y).
- $(\textbf{Sign}, \sqsubseteq)$ is a complete lattice. For example:
  - Given $Y = \{\text{Neg}, \text{Zero}\}$, $\bigsqcup Y = \text{Non-Pos}$ where Non-Pos $\in$ **Sign**.
  - Given $Y = \{\text{Neg}, \text{Zero}, \text{Pos}\}$, $\bigsqcup Y = \top$ where $\top \in$ **Sign**.

Step 1: Abstract Domain (Abstract Integers)

- **Important Requirement**: $(D, \sqsubseteq)$ must be a **complete lattice**.
- A partially ordered set (poset) $(D, \sqsubseteq)$ is a **complete lattice**, iff every subset $Y \subseteq D$ has $\bigsqcup Y \in D$ (the least upper bound of Y).
- $(\textbf{Sign}, \sqsubseteq)$ is a complete lattice. For example:
  - Given $Y = \{\text{Neg}, \text{Zero}\}$, $\bigsqcup Y = \text{Non-Pos}$ where $\text{Non-Pos} \in \textbf{Sign}$.
  - Given $Y = \{\text{Neg}, \text{Zero}, \text{Pos}\}$, $\bigsqcup Y = \top$ where $\top \in \textbf{Sign}$.

Q. Why should $(D, \sqsubseteq)$ be a complete lattice?

Step 1: Abstract Domain (Abstract Integers)

- **Important Requirement**: $(D, \sqsubseteq)$ must be a **complete lattice**.
- A partially ordered set (poset) $(D, \sqsubseteq)$ is a **complete lattice**, iff every subset $Y \subseteq D$ has $\bigsqcup Y \in D$ (the least upper bound of Y).
- $(\textbf{Sign}, \sqsubseteq)$ is a complete lattice. For example:
  - Given $Y = \{\text{Neg}, \text{Zero}\}$, $\bigsqcup Y =$ Non-Pos where Non-Pos $\in$ **Sign**.
  - Given $Y = \{\text{Neg}, \text{Zero}, \text{Pos}\}$, $\bigsqcup Y = \top$ where $\top \in$ **Sign**.

Q. Why should $(D, \sqsubseteq)$ be a complete lattice?
A. To combine abstract values from multiple branches, while ensuring **safe approximations**. Consider the snippet where the conditional expression depends on some external inputs.

$$\text{if}(\ldots)\ \{\ x := \text{Neg}\ \}\ \text{else}\ \{\ x := \text{Zero}\ \}$$

If the least upper bound between Neg and Zero is not defined, we cannot compute over-approximations.

Step 1: Abstract Domain (Abstract Integers)

The join ($\sqcup$) operator, the least upper bound between two elements, is defined as follows.

$$\boldsymbol{a} \sqcup \boldsymbol{b} = \begin{cases} \boldsymbol{a} & \cdots \text{ if } \boldsymbol{b} \sqsubseteq \boldsymbol{a} \\ \boldsymbol{b} & \cdots \text{ if } \boldsymbol{a} \sqsubseteq \boldsymbol{b} \\ \text{Non-Zero} & \cdots \text{ if } (\boldsymbol{a}, \boldsymbol{b}) = (\text{Neg}, \text{Pos}) \text{ or } (\boldsymbol{b}, \boldsymbol{a}) = (\text{Neg}, \text{Pos}) \\ \text{Non-Pos} & \cdots \text{ if } (\boldsymbol{a}, \boldsymbol{b}) = (\text{Neg}, \text{Zero}) \text{ or } (\boldsymbol{b}, \boldsymbol{a}) = (\text{Neg}, \text{Zero}) \\ \text{Non-Neg} & \cdots \text{ if } (\boldsymbol{a}, \boldsymbol{b}) = (\text{Zero}, \text{Pos}) \text{ or } (\boldsymbol{b}, \boldsymbol{a}) = (\text{Zero}, \text{Pos}) \\ \top & \cdots \text{ otherwise} \end{cases}$$

Step 1: Abstract Domain (Abstract Integers)

We can extend the lattice of abstract integers into that of abstract states.

- The complete lattice of abstract states $(\widehat{\textbf{State}}, \sqsubseteq)$:

$$\widehat{\textbf{State}} = Var \to \textbf{Sign}$$

  with the pointwise ordering:

$$\hat{s}_1 \sqsubseteq \hat{s}_2 \iff \forall x \in Var.\ \hat{s}_1(x) \sqsubseteq \hat{s}_2(x).$$

  - $[x \mapsto \text{Pos}, y \mapsto \text{Zero}] \sqsubseteq [x \mapsto \top, y \mapsto \top]$
  - $[x \mapsto \text{Pos}, y \mapsto \text{Zero}] \not\sqsubseteq [x \mapsto \top, y \mapsto \text{Pos}]$

- The least upper bound of $Y \subseteq \widehat{\textbf{State}}$,

$$\bigsqcup Y = \lambda x.\ \bigsqcup_{\hat{s} \in Y} \hat{s}(x).$$

  i.e., $\hat{s_1} \sqcup \hat{s_2} = \lambda x.\ s_1(x) \sqcup s_2(x)$.

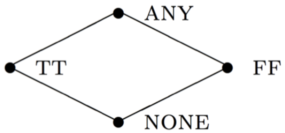
Step 1: Abstract Domain (Abstract Booleans)

The truth values $\mathbf{T} = \{true, false\}$ are abstracted by the complete lattice $(\widehat{\mathbf{T}}, \sqsubseteq)$:

$$\widehat{\mathbf{T}} = \{\top, \bot, \widehat{true}, \widehat{false}\}$$

where $\top$ = ANY, $\bot$ = NONE, $\widehat{true}$ = TT, and $\widehat{false}$ = FF.

$$\widehat{b_1} \sqsubseteq \widehat{b_2} \iff \widehat{b_1} = \widehat{b_2} \ \vee \ \widehat{b_1} = \bot \ \vee \ \widehat{b_2} = \top$$



Step 2: Abstract Semantics

- After defining the abstract domain, we should define abstract transformers for each statement.
- A counter-part of concrete semantics.
  - In concrete execution, each statement changes concrete memory states.
  - In abstract execution, each statement changes abstract memory states.

Concrete Semantics		**Abstract Semantics**
$[x \mapsto 7]$		$[x \mapsto \mathbf{odd}]$
$\downarrow$		$\downarrow$
$F(\boxed{\textbf{x := x+2;}})$	**VS.**	$\widehat{F}(\boxed{\textbf{x := x+2;}})$
$\downarrow$		$\downarrow$
$[x \mapsto 9]$		$[x \mapsto \mathbf{odd}]$

Design Principle: abstract semantics must be **conservative** with respect to the concrete semantics (i.e., over-approximate the concrete semantics).

Step 2: Abstract Semantics

$$
\begin{array}{rcl}
\widehat{\mathcal{A}}[\![a]\!] & : & \widehat{\mathsf{State}} \to \mathsf{Sign} \\
\widehat{\mathcal{A}}[\![n]\!](\hat{s}) & = & \alpha_{\mathsf{Sign}}(n) \\
\widehat{\mathcal{A}}[\![x]\!](\hat{s}) & = & \hat{s}(x) \\
\widehat{\mathcal{A}}[\![a_1 + a_2]\!](\hat{s}) & = & \widehat{\mathcal{A}}[\![a_1]\!](\hat{s}) +_{\mathsf{Sign}} \widehat{\mathcal{A}}[\![a_2]\!](\hat{s}) \\
\widehat{\mathcal{A}}[\![a_1 \star a_2]\!](\hat{s}) & = & \widehat{\mathcal{A}}[\![a_1]\!](\hat{s}) \star_{\mathsf{Sign}} \widehat{\mathcal{A}}[\![a_2]\!](\hat{s}) \\
\widehat{\mathcal{A}}[\![a_1 - a_2]\!](\hat{s}) & = & \widehat{\mathcal{A}}[\![a_1]\!](\hat{s}) -_{\mathsf{Sign}} \widehat{\mathcal{A}}[\![a_2]\!](\hat{s})
\end{array}
$$

where α_{Sign} abstracts the integer values:

$$
\alpha_{\mathsf{Sign}}(z) = \begin{cases} \mathsf{Neg} & \cdots \ z < 0 \\ \mathsf{Zero} & \cdots \ z = 0 \\ \mathsf{Pos} & \cdots \ z > 0 \end{cases}
$$

Step 2: Abstract Semantics

$+_S$	NONE	NEG	ZERO	POS	NON-POS	NON-ZERO	NON-NEG	ANY
NONE	NONE	NONE	NONE	NONE	NONE	NONE	NONE	NONE
NEG	NONE	NEG	NEG	ANY	NEG	ANY	ANY	ANY
ZERO	NONE	NEG	ZERO	POS	NON-POS	NON-ZERO	NON-NEG	ANY
POS	NONE	ANY	POS	POS	ANY	ANY	POS	ANY
NON-POS	NONE	NEG	NON-POS	ANY	NON-POS	ANY	ANY	ANY
NON-ZERO	NONE	ANY	NON-ZERO	ANY	ANY	ANY	ANY	ANY
NON-NEG	NONE	ANY	NON-NEG	POS	ANY	ANY	NON-NEG	ANY
ANY	NONE	ANY	ANY	ANY	ANY	ANY	ANY	ANY

$\star_S$	NEG	ZERO	POS
NEG	POS	ZERO	NEG
ZERO	ZERO	ZERO	ZERO
POS	NEG	ZERO	POS

$-_S$	NEG	ZERO	POS
NEG	ANY	NEG	NEG
ZERO	POS	ZERO	NEG
POS	POS	POS	ANY

Exercise) complete the definitions of the abstract multiplication ($\star_{\mathbf{Sign}}$) and the abstract subtraction ($-_{\mathbf{Sign}}$).

Step 2: Abstract Semantics

$$
\begin{array}{rcl}
\widehat{\mathcal{B}}[\![\ b\ ]\!] & : & \widehat{\textsf{State}} \to \widehat{\textsf{T}} \\
\widehat{\mathcal{B}}[\![\ \texttt{true}\ ]\!](\hat{s}) & = & \widehat{true} \\
\widehat{\mathcal{B}}[\![\ \texttt{false}\ ]\!](\hat{s}) & = & \widehat{false} \\
\widehat{\mathcal{B}}[\![\ a_1 = a_2\ ]\!](\hat{s}) & = & \widehat{\mathcal{A}}[\![\ a_1\ ]\!](\hat{s}) =_{\textsf{Sign}} \widehat{\mathcal{A}}[\![\ a_2\ ]\!](\hat{s}) \\
\widehat{\mathcal{B}}[\![\ a_1 \leq a_2\ ]\!](\hat{s}) & = & \widehat{\mathcal{A}}[\![\ a_1\ ]\!](\hat{s}) \leq_{\textsf{Sign}} \widehat{\mathcal{A}}[\![\ a_2\ ]\!](\hat{s}) \\
\widehat{\mathcal{B}}[\![\ \neg b\ ]\!](\hat{s}) & = & \neg_{\widehat{\textsf{T}}} \widehat{\mathcal{B}}[\![\ b\ ]\!](\hat{s}) \\
\widehat{\mathcal{B}}[\![\ b_1 \wedge b_2\ ]\!](\hat{s}) & = & \widehat{\mathcal{B}}[\![\ b_1\ ]\!](\hat{s}) \wedge_{\widehat{\textsf{T}}} \widehat{\mathcal{B}}[\![\ b_2\ ]\!](\hat{s})
\end{array}
$$

Step 2: Abstract Semantics

$=_S$	NEG	ZERO	POS
NEG	ANY	FF	FF
ZERO	FF	TT	FF
POS	FF	FF	ANY

$\leq_S$	NEG	ZERO	POS
NEG	ANY	TT	TT
ZERO	FF	TT	TT
POS	FF	FF	ANY

$\neg_T$	
NONE	NONE
TT	FF
FF	TT
ANY	ANY

$\wedge_T$	NONE	TT	FF	ANY
NONE	NONE	NONE	NONE	NONE
TT	NONE	TT	FF	ANY
FF	NONE	FF	FF	FF
ANY	NONE	ANY	FF	ANY

Exercise) complete the definitions of the abstract boolean operators.

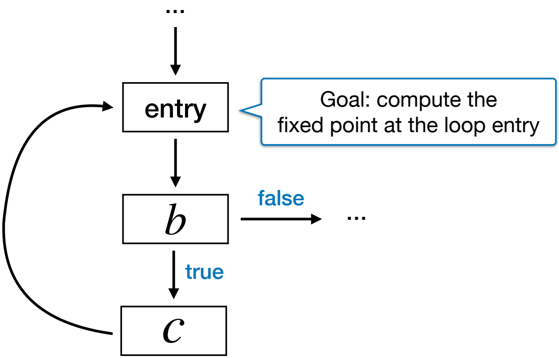
Step 2: Abstract Semantics

$$
\begin{array}{rcl}
\widehat{\mathcal{C}}[\![\, c \,]\!] & : & \widehat{\textsf{State}} \to \widehat{\textsf{State}} \\
\widehat{\mathcal{C}}[\![\, x := a \,]\!] & = & \lambda\hat{s}.\hat{s}[x \mapsto \widehat{\mathcal{A}}[\![\, a \,]\!](\hat{s})] \\
\widehat{\mathcal{C}}[\![\, \texttt{skip} \,]\!] & = & \textsf{id} \\
\widehat{\mathcal{C}}[\![\, c_1; c_2 \,]\!] & = & \widehat{\mathcal{C}}[\![\, c_2 \,]\!] \circ \widehat{\mathcal{C}}[\![\, c_1 \,]\!] \\
\widehat{\mathcal{C}}[\![\, \texttt{if}\ b\ c_1\ c_2 \,]\!] & = & \widehat{\textsf{cond}}(\widehat{\mathcal{B}}[\![\, b \,]\!], \widehat{\mathcal{C}}[\![\, c_1 \,]\!], \widehat{\mathcal{C}}[\![\, c_2 \,]\!]) \\
\widehat{\mathcal{C}}[\![\, \texttt{while}\ b\ c \,]\!] & = & \lambda\hat{s}.\mathit{fix}(\lambda\hat{x}.\hat{s} \sqcup \widehat{\mathcal{C}}[\![\, c \,]\!](\hat{x}))
\end{array}
$$

$$
\widehat{\textsf{cond}}(f, g, h)(\hat{s}) = \begin{cases} \bot & \cdots f(\hat{s}) = \bot \\ g(\hat{s}) & \cdots f(\hat{s}) = \widehat{\mathit{true}} \\ h(\hat{s}) & \cdots f(\hat{s}) = \widehat{\mathit{false}} \\ g(\hat{s}) \sqcup h(\hat{s}) & \cdots f(\hat{s}) = \top \end{cases}
$$

Step 2: Abstract Semantics

- The abstract semantics for the while-loop over-approximates the states of the **loop entry**.
- That is, it aims to compute "stable" abstract memory states that are conservative over possibly infinite number of loop iterations.



Example: Sign Analysis

Suppose y and z are input parameters.

```
x = 0;
y = 0;
while (y <= n) {
  if (z == 0) {
    x = x+1;
  }
  else {
    x = x+y;
  }
  y = y+1;
}
assert (x >= 0); /* Goal: prove the assertion */
```

	0	1	2
x	Zero	Non-Neg	Non-Neg
y	Zero	Non-Neg	Non-Neg
n	$\top$	$\top$	$\top$
z	$\top$	$\top$	$\top$

Summary

Abstract interpretation is a framework for automatically computing **safe approximations** of program states.

1. **Abstract Domain** : define the abstract values that each variable can have (i.e., fixes "shape" of the invariants).
   - $c_1 \leq x \leq c_2$ (interval), $\pm x \pm y \leq c$ (octagon)
2. **Abstract Semantics** (abstract transformers): define how to execute each statement in the chosen abstract domain.

Concrete Semantics **Abstract Semantics**

$$\begin{array}{ccc} [x \mapsto 7] & & [x \mapsto \mathbf{odd}] \\ \downarrow & & \downarrow \\ F(\boxed{\texttt{x := x+2;}}) & \textbf{VS.} & \widehat{F}(\boxed{\texttt{x := x+2;}}) \\ \downarrow & & \downarrow \\ [x \mapsto 9] & & [x \mapsto \mathbf{odd}] \end{array}$$

3. Run the analysis, i.e., execute the program using **abstract semantics**.