Introduction to static analysis #3

Seminar @ Gondow Lab.

Goal of This Chapter

- The construction of a *static analysis framework*.
  - feature : general, can be used with different abstraction.
  - goal : compute program invariants by static abstraction
- How to construct a static analysis step by step.
  - We use basic programming language that operates over numerical states.

Outline of the book

> - 3.1 : fix the language and its semantics.(6p)
> - 3.2 : select an abstraction and fix their representation.(9p)
> - 3.3 : derive the abstract semantics of programs from their semantics and abstractions.(18p)
> - 3.4 : design of the interpreter.(2p)

Overview

- Semantics (3.1)
  - Programming Language
  - Concrete Semantics
    - Concrete Semantics
    - Properties of Interest
    - Input-Output Semantics
- Abstraction (3.2)
- Computable Abstract Semantics (3.3)
- Interpreter (3.4)

A Simple Programming Language (1/2)

We use simple programming language to illustrate the concepts of static analysis.

Some preparations:

- $\mathbb{X}$: a finite set of variable(which is fixed)
- $\mathbb{V}$: a set of scalar value
- $\mathbb{B}$: a set of boolean value
  - $\mathbb{B} = \{\mathbf{true}, \mathbf{false}\}$

A Simple Programming Language (2/2)

Syntax of our language is:

- $n \quad \in \quad \mathbb{V}$
- $\mathrm{x} \quad \in \quad \mathbb{X}$
- $\odot \quad ::= \quad + \mid - \mid * \mid \ldots$
- $\lessdot \quad ::= \quad < \mid \leq \mid == \mid \ldots$
- $E \quad ::= \quad n \mid \mathrm{x} \mid E \odot E$
- $B \quad ::= \quad \mathrm{x} \lessdot n$
  - returns an element of $\mathbb{B}(= \{\textbf{true}, \textbf{false}\})$
- $C \quad ::= \quad \textbf{skip} \mid C; C \mid \mathrm{x} := E \mid \textbf{input}(\mathrm{x}) \mid \textbf{if}(B)\{C\}\textbf{else}\{C\}$
- $P \quad ::= \quad C$

A Simple Programming Language (3/2)

- $n \quad \in \quad \mathbb{V}$
  - scalar values
- $\mathbf{x} \quad \in \quad \mathbb{X}$
  - program variables
- $\odot \quad ::= \quad + \mid - \mid * \mid \ldots$
  - binary operators
- $\lessdot \quad ::= \quad < \mid \leq \mid == \mid \ldots$
  - comparison operators
- $E \quad ::= \quad n \mid \mathbf{x} \mid E \odot E$
  - scalar expressions

A Simple Programming Language (4/2)

- $B \quad ::= \quad \mathrm{x} \lessdot n$
  - returns an element of $\mathbb{B}(= \{\mathbf{true}, \mathbf{false}\})$
  - Boolean expressions
- $C \quad ::= \quad \mathbf{skip} \mid C; C \mid \mathrm{x} := E \mid \mathbf{input}(\mathrm{x}) \mid \mathbf{if}(B)\{C\}\mathbf{else}\{C\}$
  - commands
- $P \quad ::= \quad C$
  - program

Concrete Semantics

There're several kind of semantics. For instance, **trace semantics**, **denotational semantics**.

- **trace semantics** : describes program execution as a sequence of program state
- **denotational semantics** : describes only input-output relation

Before we can select which semantics to use, we discuss the family of properties of interest.

Properties of Interest

As in chapter 2, we focus on **reachability** properties.

Examples:

1. absence of run-time errors
2. verification of user assertions
   - execution should reach assertion point but should not meet the assertion condition

More general properties will be addressed in chapter 9.

Properties of Interest - reachability

Checking reachability properties would be:

1. pre-condition $\longrightarrow$ post-condition ($\longleftarrow$We need a semantic that capture this)
2. check post-condition

So we use ***input-output semantics***(one of denotational semantics).

An Input-Output Semantics

- Input-output semantics :
  - set of input states $\longmapsto$ set of output states
  - use mathematical function to map
  - output is a set of states because:
    - of the non-deterministic execution of **input**
    - we may observe infinitely many output states from one input
  - input is also a set of states
    - for the sake of compositionality

An Input-Output Semantics - compositionality

- Input-Output Semantics ***compositional***.

> 📓 compositional : the semantics of a command can be defined by composing the semantics of its sub-commands.
>
> e.g
>
> > $C := C_1; C_2$
>
> Semantics of C is defined by that of C_1 and C_2.

An Input-Output Semantics vs Interpreter

Input-output Semantics and Interpreter have much in common:

- input-output : set of input state**s** $\longmapsto$ set of output state**s**
- interpreter : a program and an input state $\longmapsto$ an output state

The main difference is:

- interpreter : inputs a ***single*** state and returns a ***single*** state

Essentially, interpreter implements the input-output semantics.

Memory States(1/2)

- *program state* should include:
  - *memory state* : contents of the memory
  - *control state* : a value of "program counter"(or next command to be executed)
- a state is defined by a memory state:
  - we use input-output semantics
  - input(output) state is fully determined by the contents of memory

Memory States(2/2)

- memory state $\mathbb{M}$ is defined by:
  - $\mathbb{M} = \mathbb{X} \longrightarrow \mathbb{V}$

example:

- $\mathbb{X} = \{\mathrm{x}, \mathrm{y}\}$
  - $\mathrm{x} : 2, \mathrm{y} : 7$
- $m \in \mathbb{M}$ is:
  - $m = \{\mathrm{x} \mapsto 2, \mathrm{y} \mapsto 7\}$

> 📓 簡単に言うと、memory stateとは変数からスカラー値のマッピングのこと

Semantics of Scalar Expressions

How scalar expressions are evaluated.

- $[\![E]\!](m)$: semantics of expression E, in the memory state m.
  - $[\![E]\!] : \mathbb{M} \longrightarrow \mathbb{V}$
    - This is a function from memory states to scalar values

Semantics of each scalar expression is as follows:

- $[\![n]\!](m) = n$
- $[\![\mathrm{x}]\!](m) = m(\mathrm{x})$
  - $m(\mathrm{x})$: value of x in the memory state m
- $[\![E_0 \odot E_1]\!](m) = f_\odot([\![E_0]\!](m), [\![E_1]\!](m))$
  - $f_\odot$: mathematical function associated to the binary operator $\odot$

Semantics of Boolean Expressions

How Boolean expressions are evaluated.

- $[\![B]\!] = \mathbb{M} \longrightarrow \mathbb{B}$
  - This is a function from memory states to boolean values
- $[\![\mathrm{x} \lessdot n]\!] = f_{\lessdot}(m(\mathrm{x}), n)$
  - $f_{\lessdot}$: mathematical function associated to the comparison operator $\lessdot$

Semantics of Commands (1/6)

- $[\![C]\!]_{\mathscr{P}}$: semantics of a command C
  - a set of input states to a set of output states(which is observed **after** the command)
    - non-terminating executions are not observed
- $\wp(\mathbb{M})$: power set of memory states
  - intuitive explanation : "whether or not each variable is defined"
  - M : an element of $\wp(\mathbb{M})$, that is:
    - $M \in \wp(\mathbb{M})$

As a result, semantics of commands can be written as follows:

- $[\![C]\!]_{\mathscr{P}} : \wp(\mathbb{M}) \longrightarrow \wp(\mathbb{M})$

Semantics of Commands (2/6)

Semantics of commands is:

- $[\![\texttt{slip}]\!]_{\mathscr{P}}(M) = M$
  - identity function
- $[\![C_0; C_1]\!]_{\mathscr{P}}(M) = [\![C_1]\!]_{\mathscr{P}}([\![C_0]\!]_{\mathscr{P}}(M))$
  - composition of the semantics of each commands
- $[\![\mathrm{x} := E]\!]_{\mathscr{P}}(M) = \{m[\mathrm{x} \mapsto [\![E]\!](m)] \mid m \in M\}$
  - the evaluation of assignment updates the value of $\mathtt{x}$ in the memory states with the result of the evaluation of E.
- $[\![\texttt{input}(\mathrm{x})]\!]_{\mathscr{P}}(M) = \{m[\mathrm{x} \mapsto n] \mid m \in M, n \in \mathbb{V}\}$
  - replace the value of x with any possible scalar value n.

Quite easy.

Semantics of Commands (3/6)

Before we define semantics of **`if-else`** or **`while`**, we need some preparations.

- $\mathscr{F}_B$: filtering function. We need to define this first.
  - This function filter out memory states

Definition is as follows:

- $\mathscr{F}_B(M) = \{m \in M \mid [\![B]\!](m) = \mathbf{true}\}$
  - intuitive explanation : filter out memory states m in which B doesn't hold or can't be defined

Semantics of Commands (4/6)

Semantics of **`if-else`**:

- $[\![\texttt{if}(B)\{C_0\}\texttt{else}\{C_1\}]\!]_{\mathscr{P}}(M) = [\![C_0]\!]_{\mathscr{P}}(\mathscr{F}_B(M)) \cup [\![C_1]\!]_{\mathscr{P}}(\mathscr{F}_{\neg B}(M))$
  - union of the results of each branch

Semantics of Commands (5/6)

Semantics of **`while`**:

- $[\![\texttt{while}(B)\{C\}]\!]_{\mathscr{P}}(M) = \mathscr{F}_{\neg B}\big(\cup_{i\geq 0}\left([\![C]\!]_{\mathscr{P}} \circ \mathscr{F}_B\right)^i(M)\big)$
  - complicated...

Let M_i be as follows:

- $M_i = \mathscr{F}_{\neg B}\big(([\![C]\!]_{\mathscr{P}} \circ \mathscr{F}_B)^i(M)\big)$
  - intuitive explanation : B evaluates to **true** i times and to **false** for the last.
  - $[\![C]\!]_{\mathscr{P}} \circ \mathscr{F}_B$: filter memory states with B, then execute the command.

Semantics of Commands (6/6)

> Semantics of **`while`**:
>
> - $[\![\mathtt{while}(B)\{C\}]\!]_{\mathscr{P}}(M) = \mathscr{F}_{\neg B}\Big(\cup_{i\geq 0}\left([\![C]\!]_{\mathscr{P}} \circ \mathscr{F}_B\right)^i(M)\Big)$
>   - complicated...

Then, the set of output states would be $M_0 \cup M_1 \cup M_2 \ldots$, that is :

- $[\![\mathtt{while}(B)\{C\}]\!]_{\mathscr{P}}(M) = \cup_{i\geq 0} M_i = \cup_{i\geq 0}\mathscr{F}_{\neg B}\Big(\left([\![C]\!]_{\mathscr{P}} \circ \mathscr{F}_B\right)^i(M)\Big)$

$\mathscr{F}_B$ commutes with the union, thus:

- $\cup_{i\geq 0} M_i = \mathscr{F}_{\neg B}\Big(\cup_{i\geq 0}\left([\![C]\!]_{\mathscr{P}} \circ \mathscr{F}_B\right)^i(M)\Big)$

Therefore,

- $[\![\mathtt{while}(B)\{C\}]\!]_{\mathscr{P}}(M) = \mathscr{F}_{\neg B}\Big(\cup_{i\geq 0}\left([\![C]\!]_{\mathscr{P}} \circ \mathscr{F}_B\right)^i(M)\Big)$