A Note on Abstract Interpreter Using Compositional Semantics

by Chew Feng

§1. Background

I always believe that scalability is the key for verification. I have been aware of an analysis technique called *abstract interpretation* for a while. People say it is a technique that scales very well. But there are not so many holistic materials on this topic, expect a recent textbook by Rival [1]. Therefore, I requested the school library to find it. Luckily, they find it from the library of University of Queensland and get it back to me after some 3 months. I was surprised by how this inter-library loan works.

This is a note of the Chapter 3 from Rival's textbook, which focuses on building an abstract interpreter of a language with *compositional semantics*.

§2. Compositional Semantics

A language is equipped with a **compositional semantics** if the meaning of a complex expression is determined by the meanings of its sub-expressions. An example of compositional semantics is the **denotational semantics**, which view the meaning of program as an mathematical function in the form of

$$\text{program} : \text{input} \mapsto \text{output}$$

In this view, programs have no side effect, thus it is compositional: suppose g and f are two programs and $g \circ f$ is the composed program, we have :

$$\forall x, g \circ f(x) = g(f(x))$$

Rival uses the classical *WHILE* language for an example to illustrate its denotational semantics. It would be too tedious to list the syntax of its language. Let's just look at some examples:

```
// max(x, y)
input(x);
input(y);
if(x >= y) {
  skip;
} else {
  x := y
}

// Fibonacci Sequence
a := 0;
b := 1;
while b <= 1000000 {
  b := a + b;
  a := b - a
}
```

This language only has two types for value: natural numbers $\mathbb{N}$ and booleans $\mathbb{B}$. While minimalist, this language has (nondeterministic) input, if-else, assignment, while commands. In addition, there are two types expressions to consider:

- Scalar Expression $\odot : \mathbb{N} \to \mathbb{N} \to \mathbb{N}$. This includes "`+`", "`-`" in the example.
- Boolean Expression: $\circledcirc : \mathbb{N} \to \mathbb{N} \to \mathbb{B}$. This includes "`<=`" in the example

Representing expressions in denotational semantics is fairly easy: just define mathematical functions for operators `+`, `-`, etc.

For commands, it needs some thoughts. Firstly, this is an imperative language and it works fundamentally on reading from and writing to the memory. To assign it with denotational semantics, we need a formalism of program state $\mathbb{M}$ (for memory). The straightforward one is

$$\mathbb{M} : \mathbb{X} \to \mathbb{V}$$

where $\mathbb{X}$ is a set of variables and $\mathbb{V}$ is a set of values, and we view a program as a function:

$$[\![C]\!] : \mathscr{P}(\mathbb{M}) \to \mathscr{P}(\mathbb{M})$$

where $\mathscr{P}$ is the power set function. That is to say, a program takes a set of input states, return a set of output states. By doing so, we interpret the composition of programs by

$$[\![C_1; C_2]\!](M) = [\![C_1]\!]([\![C_2]\!](M))$$

Problem 2.1.
Why the input and output is a set of states, instead of a single state? This is because the language has a nondeterministic command `input`. Consider this program:

```
input(x)
if(x > 10) {
  y = true;
} else {
  y = false;
}
```

What should be the output of the semantics? Intuitively, it should be two states:

$$\{\{x > 10, y \mapsto T\}, \{x \leq 10, y \mapsto F\}\}$$

Now you should agree that output should be a set of states, not a single state. Besides, to make the semantics composable, the input should be of the same type of the output.

Semantics for input Input is nondeterministic. The command `input(x)` has the semantics that assign x to a symbolic value that represent all possible result.[1] Let n represents this magical value, Hence we have:

$$[\![\text{input}(x)]\!](M) = \{m[x \mapsto n] \mid \forall m \in M\}$$

Semantics for branching. Let's design the semantics for the if-else statement. It needs to execute different branch based on if the program state $\mathbb{M}$ satisfy a boolean condition $B : \mathbb{M} \to \mathbb{B}$. Let $\mathscr{F}_B$ be a filter function on B, which filters the state based on whether a state satisfies B:

[1] We only assume input natural numbers

$$\mathscr{F}_B : \mathscr{P}(\mathbb{M}) \to \mathscr{P}(\mathbb{M})$$
$$\mathscr{F}_{B(M)} := \{m \in M \mid [\![B]\!](m) = T\}$$

We now can use the filter function for semantics of branching

$$[\![\text{if}(B)\{C_0\}\text{else}\{C_1\}]\!](M) = [\![C_0]\!](\mathscr{F}_B(M)) \cup [\![C_1]\!](\mathscr{F}_{\neg B}(M))$$

This definition is accurate by taking the union of all states produced two possible branches. However, it is hard for analyzer to simulate : If there are n states in M, the branching command can create $2n$ states.

Semantics For Loops The final block of this language is the loop command while$(B)\{C\}$. Informally, the semantics should be composed by a number of Cs until condition B turns false.

As an aside, by viewing the program as a function input $\mapsto$ output, we already assume that the program halts eventually.

The textbook gives such interesting semantics for loops [1]:

$$[\![\text{while}(B)\{C\}]\!](M) = \mathscr{F}_{\neg B}\left(\bigcup_{i\geq 0} ([\![C]\!] \circ \mathscr{F}_B)^i\right)(M)$$

What makes it interesting is that it does not calculate the iteration of the loops. Instead, it exploit the filter function $\mathscr{F}$ in such a way that allows unbounded loops:

- The number of iteration is unbounded. Hence, it unions all $i > 0$.
- After compose the loop command C, it filters out the states where the condition B does not hold. Hence, for each iteration, the semantics is $[\![C]\!] \circ \mathscr{F}_B$.
- The loops end when the condition B becomes false. The outer filter function $\mathscr{F}_{\neg B}$ guards such requirement.

Apart from the trick of exploiting the filter function, it is also interesting because it shows that it is not necessary for the semantics to be computable: this semantics rule is valid even it assume the loop is unbounded.

§3. Analysis by Abstraction

§3.1. An Abstract Interpreter

Properties of Interest Now consider what we want to know about a program before it executes. One fundamental property of programs is the *reachability*, which ask what states can be reached in execution and what can not. This is a fundamental property. For example, people want to show that a program never goes to some error state, and this reduces to the reachability property.

Programs are hard to statically simulated Most useful programs contain nondeterministic components. It might from user input, a random number generator, etc. Consider this program:

```
input(x1); input(x2);
if x1 > 0 then
  if x2 > 0 then ...
```

There're 2^2 possible paths to track. More generally, when there is n nondeterministic variables being tested by branching commands, 2^n possible paths need to be tracked. People have an cool name for this problem: *path explosion*.

Abstraction To deal with path explosion, abstraction comes into play. While states in the *concrete domain* is hard to simulate, we can design an *abstract domain* that over-approximate it. One abstract domain of concrete real numbers $\mathbb{R}$ is the domain $\{\geq 0, \leq 0, 0\}$ that represent the sign of an variable.

For example,

$$\{\{x \mapsto 1\}, \{x \mapsto 2\}, \{x \mapsto 3\}, ...\}$$

can be simply abstracted by

$$\{x \mapsto [\geq 0]\}$$

In addition, we usually require the abstract domain to be a lattice. For the abstract sign domain, we add $\top$ for all signs and $\bot$ for no result. This allows us to use the join $\sqcup$ and meet $\sqcap$ operator to simplify the analysis. We have this lattice:

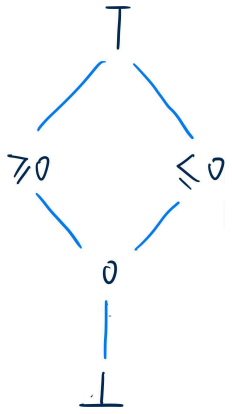

Figure 1: The lattice of abstract sign domain

The textbook define the formal relation of abstraction.

- $\mathbb{C}$ is the concrete domain, and $\mathbb{A}$ is the abstract domain.
- It is required that there is an order relation $\subseteq$ defined on $\mathbb{C}$, and an order relation $\sqsubseteq$ defined on $\mathbb{A}$.
- $\forall c \in \mathbb{C}, a \in \mathbb{A}, c \vDash a$ when a describes c.[2]
- A function $\gamma : \mathbb{A} \to \mathbb{C}$ is the concretization function if $\gamma(a) \vDash a$ and $\gamma(a)$ is the maximum element in $\mathbb{C}$ satisfies a.

Finally, we call define abstraction:

Definition 3.1.1 (Abstraction).
Domain $\mathbb{A}$ is an abstraction of $\mathbb{C}$ if

- $\forall c \in \mathbb{C}, \forall (a_0, a_1 \in \mathbb{A}), c \vDash a_0 \to a_0 \sqsubseteq a_1 \to c \vDash a_1$
- $\forall (c_0, c_1 \in \mathbb{C}), \forall a \in \mathbb{A}, c_0 \subseteq c_1 \to c_1 \vDash a \to c_0 \vDash a$

[2] I read "$\vDash$" as "abstract to". Not sure why they use this entailment symbol.

This definition essentially requires $\vDash$ to be compatible with both order relations. Consider the abstract domain of signs Figure 1 and concrete domains of real numbers. An example of the first assertion is[3]

$$\{0\} \vDash \{[= 0]\} \wedge \{[= 0]\} \sqsubseteq \{[\geq 0]\} \rightarrow \{0\} \vDash \{[\geq 0]\}$$

And similarly, an example of the second assertion:

$$\{1, 2\} \subseteq \{1, 2, 3\} \wedge \{1, 2, 3\} \vDash \{[\geq 0]\} \rightarrow \{1, 2\} \vDash \{[\geq 0]\}$$

Hence, we have this lattice:

•

Having the abstraction, we can simulate the program behaviors on the abstract domain, instead of on the concrete domain. The process is like **executing the program on the abstract domain**. Like $[\![C]\!]$, We use $[\![C]\!]_\sharp$ for interpreting program the C on the abstract domain.

Abstract Semantics for Branching Consider following program that computes the absolute value of x, we can use the abstract sign domain (Figure 1) to simulate it:

```
input(x);
if(x > 0) {y = x} else {y = -x}
```

Before branching, the state is

$$\{x \mapsto \top\}$$

For the branch where $x > 0$, we have

$$\{x \mapsto [\geq 0], y \mapsto [\geq 0]\}$$

similarly, for the branch where $x \leq 0$, we have

$$\{x \mapsto [\leq 0], y \mapsto [\geq 0]\}$$

To join these two states by $\sqcup$, we have

$$\{x \mapsto \top, y \mapsto [\geq 0]\}$$

Thus, we prove that this program always produce a non-negative `y`.

Recall that the concrete semantics of branching: for n input states, there will be $2n$ states be produced. However when abstractly analysed, we use the join operator on the lattice to avoid the increase of states, and it makes analysis more scalable, at the cost of losing precision (we give up all information about x).

Abstract Semantics for Loops Another important trick about abstract interpretation is the handling of loops. Recall that the concrete semantics treat loops as unbounded iteration. While accurate, it is not a computable procedure. For analysis, we need to find a way out of simulating a loop by finite steps.

One key observation is that, if the loop *converges*, we can stop the iteration. Convergence of loops means that for some natural number k, the state at $k + 1$ iteration is the same as k. This fact is fairly intuitive and we omit further proof.

So, we want analysis of a loop to converge rapidly. The technique for convergence is called *widening*.

[3]Here, $\{1, 2, 3\}$ means the set of program states $\{\{x \mapsto 1\}, \{x \mapsto 2\}, \{x \mapsto 3\}\}$.

Definition 3.1.2 (Widening).
An operator $\triangledown : \mathbb{A} \to \mathbb{A} \to \mathbb{A}$ is called widening if
1. $\forall(a_0, a_1 \in \mathbb{A}), \gamma(a_0) \cup \gamma(a_1) \subseteq \gamma(a_0 \triangledown a_1)$
2. This sequence of a'_n is ultimate stationary:

$$a'_0 = a_0$$
$$a'_{n+1} = a'_n \triangledown a_n$$

Consider analysing the command $[\![\text{while}(B)\{C\}]\!]_\sharp$, we can apply widening of the every i and $i+1$ state. The first obligation of widening ensures that it indeed "widens" the concrete elements, not shrinking, otherwise the analysis will be unsound. The second one is more important: it ensures that after applying finite widening, the loops will be converged.

The textbook takes an abstract domain of intervals as an example. The domain describes the concrete domain by the interval of variables. Consider $\{x \mapsto [n, p]\}$: (suppose $p - n > 1$)

$$\{x \mapsto n + 0\} \vDash \{x \mapsto [n, p]\}$$
$$\{x \mapsto n + 1\} \vDash \{x \mapsto [n, p]\}$$
$$\cdots$$
$$\{x \mapsto p\} \vDash \{x \mapsto [n, p]\}$$

A straightforward widening of two intervals is:

$$[n, p] \triangledown [n, q] = \begin{cases} [n, p] & \text{if } p \geq q \\ [n, +\infty) & \text{if } p < q \end{cases}$$

That is to say, if the iteration starts in $[n, p]$ and ends in $[n, q]$, if the iteration increases the bound, we simply set the bound to infinite. For example,

```
x := 0;
while(x < 50) {
  x := x + 1;
}
```

We mark the abstract state of nth iteration as $M^\sharp_n$:

$$M^\sharp_0 = \{x \mapsto [0, 0]\}$$
$$M^\sharp_1 = \{x \mapsto [0, +\infty)\}$$
$$M^\sharp_2 = \{x \mapsto [0, +\infty)\}$$

The analysis converges on the second iteration. Again, this will lose precision, but we have a fairly scalable analysis on loops.

§3.2. Soundness of Analysis

The most fundamental property of an analysis is that it should be sound. For reachability, soundness means that all states that are reachable in execution will be captured by the analysis. Hence, a general soundness property of abstract interpretation is:

Theorem 3.2.1.
For all commands C and all abstract states $M^\sharp$, the computation of $[\![C]\!]_\sharp(M^\sharp)$ is a sound analysis if

$$[\![C]\!](\gamma(M^\sharp)) \subseteq \gamma\big([\![C]\!]_\sharp(M^\sharp)\big)$$

This theorem says that the states of concrete execution is a subset of abstract interpretation. Apparently this theorem is true since it basically reiterate the definition of soundness on reachability problem.

The textbook uses an illustration (Figure 2) to express this visually.

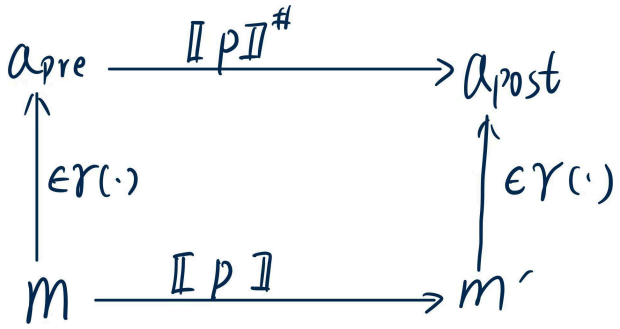

Figure 2: Sound Analysis by Abstraction

On the bottom, it shows the concrete execution of program P on input state m gets m'. On the top, the figure says the precondition (as an abstract state a_{pre}) is transformed to the the postcondition by abstractly interpreting program P. The concretization function γ acts as the connection between two procedure: the concrete states is always described by the abstract states. Hence, the analysis is sound.

§4. Summary

Let's summary what has been learned. The design of an abstract interpreter always follows this scheme:

1. Find a concrete semantics for the language.
2. Select an abstraction domain $\mathbb{A}$ of $\mathbb{C}$.
3. Define the abstract semantics based on $\mathbb{A}$ and prove it is sound.

When the analysis is not precise enough, one needs to look at the three steps.

Abstract interpretation uses two important trick to improve scalability.

1. Use the join operator $\sqcup$ to replace the concrete union $\cup$. This prevent the branching command from creating exponential paths.
2. Use the widening operator $\triangledown$ for rapid converge of loops. This avoid the unbounded loops.

Now we have an (somehow over-simplified) theoretical abstract interpreter. Congratulations!

Bibliography

[1] X. Rival and K. Yi, *Introduction to static analysis: an abstract interpretation perspective*. Mit Press, 2020.