Software Verification and Analysis

Siddhant Chaudhary

Abstract

These are some supplementary notes which I made during a course I took in CMI. There was no reference book as such, but I took some reference from the book *Software Verification and Analysis* by Janusz Laski and William Stanley.

Contents

January 2022

1. Syntax and Semantics of our language

1.1 The Syntax Grammar

In this section, we will mention the syntax of the programming language that we will analyze. This syntax will be given by a CFG.

1.1.1 Arithmetic and Boolean Expressions. Let $\mathbf{B} = \{\mathbf{true}, \mathbf{false}\}$ denote the set of boolean constants. Let $\mathbf{Z}$ denote the usual set of integers. Let $\mathbf{V} = \{x_0, y, z, x_0, x_1, ...\}$ be the set of *variables*. Next, we will define a grammar which will have these elements as terminals.

Let the set of *arithmetic expressions*, denoted by AEXP, be the language defined by the following grammar.

$$\alpha := x \mid n \mid (\alpha_1 + \alpha_2) \mid (\alpha_1 - \alpha_2) \mid (\alpha_1 * \alpha_2) \mid \alpha_1 \div \alpha_2$$

Here, $x \in \mathbf{V}$ is some variable, $n \in \mathbf{Z}$ is an integer, and α_1, α_2 are any elements in AEXP.

Similarly, we define the set of *boolean expressions*, denoted by BEXP, to be the language defined by the following grammar.

$$\beta := x \mid b \mid \alpha_1 = \alpha_2 \mid \alpha_1 \leq \alpha_2 \mid (\mathbf{not}\ \beta) \mid (\beta_1\ \mathbf{or}\ \beta_2) \mid (\beta_1\ \mathbf{and}\ \beta_2)$$

Above, $x \in \mathbf{V}$ is any variable, $b \in \mathbf{B}$ is a boolean constant, and again α_1, α_2 are any elements in BEXP.

1.1.2 Commands. Let the set of commands, denoted by COM, by the language defined by the following grammar.

$$\begin{aligned} \text{COM} :: = & \ \mathbf{skip} \\ \mid & \ x := \alpha \\ \mid & \ y := \beta \\ \mid & \ \text{COM}_1; \text{COM}_2 \\ \mid & \ \mathbf{if}\ \beta\ \mathbf{then}\ \text{COM}_1\ \mathbf{else}\ \text{COM}_2\ \mathbf{end\,if} \\ \mid & \ \mathbf{while}\ \beta\ \mathbf{do}\ \text{COM}\ \mathbf{end\,while} \end{aligned}$$

Above, $x, y \in \mathbf{V}$ are any variables. The first two statements are just like assignments in the usual programming languages. The third command is used to write multiple commands in the same program; this is just like Java/C++ syntax. The last two commands are the usual if-then-else and while constructs.

1.2 Structural Operational Semantics (SOS).

In this section, we will introduce a set of rules which will specify the execution *semantics* of our programming language.

1.2.1 Environments. First, we define the notion of an *environment*.

Definition 1.1. Consider a program with variables in the set $\mathbf{V}$ and a value domain Val (for example, $Val = \mathbf{Z} \cup \mathbf{B}$).

- An *environment* is a function $\sigma : \mathbf{V} \to Val$.
- The *substitution operation* $\sigma[v/x]$ maps an environment σ to an environment σ', where σ' is defined as follows.

$$\sigma'(y) = \begin{cases} v & , \quad \text{if } y = x \\ \sigma(y) & , \quad \text{otherwise} \end{cases}$$

- The set of all possible environments for a program is denoted by ENV.

In this terminology, one can also define *initial values* in a program. At an entry point in a program, variables can either have arbitrary values (for instance in many languages, an uninitialized string is often the empty string). Another way of specifiying initial values is by introducing a new value $Undef \in Val$ (which represents undefined), and we let the initial environment be the map $x \mapsto Undef$.

1.2.2 Evaluation. We now define the *evaluation semantics* of our language.

Definition 1.2. Let σ be any environment. Let α_1, α_2 be arithmetic expressions (i.e members of AEXP). Let β, β_1, β_2 be arbitrary boolean expressions (i.e members of BEXP). Let $x \in V$ be a variable, $n \in \mathbf{Z}$ be an integer and $b \in \mathbf{B}$ be a boolean. We define a function $eval : \text{EXP} \times \text{EXP} \to \mathbf{Z} \cup \mathbf{B}$, where $\text{EXP} = \text{AEXP} \cup \text{BEXP}$ inductively as follows.

(1) For $\bowtie \in \{+, -, *, \div, \leq, =\}$, we define

$$eval(\alpha_1 \bowtie \alpha_2, \sigma) = eval(\alpha_1, \sigma) \bowtie eval(\alpha_2, \sigma)$$

(2) For $\Delta \in \{\mathbf{or}, \mathbf{and}\}$, we define

$$eval(\beta_1 \,\Delta\, \beta_2, \sigma) = eval(\beta_1, \sigma) \,\Delta\, eval(\beta_2, \sigma)$$

(3) $eval(\mathbf{not}\, \beta, \sigma) = \neg eval(\beta, \sigma)$
(4) $eval(x, \sigma) = \sigma(x)$
(5) $eval(n, \sigma) = n$
(6) $eval(b, \sigma) = b$

Using the above definition, any expression (arithmetic or boolean) can be reduced to a value in $\mathbf{Z} \cup \mathbf{B}$.

1.2.3 Configurations. Formally, a *configuration* γ is a pair $\langle S, \sigma \rangle$ where S is either some program or ✓ and σ is an environment. A pair $\langle ✓, \sigma \rangle$ is used to denote a *final configuration* (i.e a stage in which the program has finished executing). A configuration $\langle S, \sigma \rangle$ is said to be stuck if

- $S \neq ✓$ and
- there are no more subsequent transitions. For instance, this could be the case in evaluation errors or other types of errors.

1.2.4 Defining the SOS. We will now define the *operational semantics* of our language by specifying a bunch of inference rules. The first rule will simply be

$$\langle \textbf{skip}, \sigma \rangle \longrightarrow \langle \checkmark, \sigma \rangle \tag{1}$$

In other words, an empty program will always terminate. The second rule is an inference rule specifying how assignment works.

$$\langle x := e, \sigma \rangle \longrightarrow \langle \checkmark, \sigma[eval(e, \sigma)/x] \rangle \tag{2}$$

Next, we have

$$\frac{\langle S_1, \sigma_1 \rangle \longrightarrow \langle \checkmark, \sigma_2 \rangle}{\langle S_1; S_2, \sigma_1 \rangle \longrightarrow \langle S_2, \sigma_2 \rangle} \tag{3}$$

The above inference rule specifies that our programming language is *sequential*, i.e commands execute one after another. Next, we have

$$\frac{\langle S_1, \sigma_1 \rangle \longrightarrow \langle S_3, \sigma_2 \rangle}{\langle S_1; S_2, \sigma_1 \rangle \longrightarrow \langle S_3; S_2, \sigma_2 \rangle} \tag{4}$$

Next, we will define two inference rule for the if-then-else construct. The first rule is

$$\frac{eval(B, \sigma) = \textbf{true}}{\langle \textbf{if } B \textbf{ then } S_1 \textbf{ else } S_2 \textbf{ end if}, \sigma \rangle \longrightarrow \langle S_1, \sigma \rangle} \tag{5}$$

and the second one is

$$\frac{eval(B, \sigma) = \textbf{false}}{\langle \textbf{if } B \textbf{ then } S_1 \textbf{ else } S_2 \textbf{ end if}, \sigma \rangle \longrightarrow \langle S_2, \sigma \rangle} \tag{6}$$

Finally, we specify the semantics of while loops. First, we have the followign rule.

$$\frac{eval(B, \sigma) = \textbf{true}}{\langle \textbf{while } B \textbf{ do } S \textbf{ end while}, \sigma \rangle \longrightarrow \langle S; \textbf{while } B \textbf{ do } S \textbf{ end while}, \sigma \rangle} \tag{7}$$

Next, we have

$$\frac{eval(B, \sigma) = \textbf{false}}{\langle \textbf{while } B \textbf{ do } S \textbf{ end while}, \sigma \rangle \longrightarrow \langle \checkmark, \sigma \rangle} \tag{8}$$

With the eight rules above, we have fully specified the operational semantics of our language.

1.2.5 Derivation Sequences. Given a program S starting with environment σ, a *derivation sequence* is either

- A finite sequence $\gamma_0, ..., \gamma_k$ of configurations satisfying $\gamma_0 = \langle S, \sigma \rangle$, $\gamma_i \longrightarrow \gamma_{i+1}$ for all $0 \leq i \leq k$, and γ_k is either a final configuration or a stuck configuration.
- An infinite sequence $\gamma_0, \gamma_1, \gamma_2, ...$ of configurations satisfying only the first two conditions in the previous point.

1.2.6 Specification of Errors. By introducing special values in Val which indicate undefined/erroneous behavior (for example division by zero) and by introducing a special configuration called ERROR, we can use inference rules to specify errors in our language. I won't write all the details here, but this is fairly easy to do.

2. Logic Fundamentals

2.1 Propositional Logic

2.1.1 Syntax. We start with an infinite set of atomic propositions A. Using these atomic propositions, logical formulae are created inductively; equivalently, we describe this using the following grammar.

$$\begin{aligned}\phi ::= \quad & \top \\ \mid \quad & \bot \\ \mid \quad & p \\ \mid \quad & \neg\phi_1 \\ \mid \quad & \phi_1 \wedge \phi_2 \\ \mid \quad & \phi_1 \vee \phi_2 \\ \mid \quad & \phi_1 \implies \phi_2 \\ \mid \quad & \phi_1 \iff \phi_2 \\ \mid \quad & \phi_1 \oplus \phi_2\end{aligned}$$

Above, $p \in A$ is any atom.

2.1.2 Semantics. Having defined the syntax of formulae in predicate logic, we now define the semantics of this logic. For any formula ϕ, let $atoms(\phi)$ denote the set of atoms contained in ϕ. Let $\sigma : atoms(\phi) \to \{\mathbf{true}, \mathbf{false}\}$ be any *valuation*. With respect to this valuation, every formula ϕ evaluates to either true or follows as per the following rules.

$$\begin{aligned}[\![\top]\!]_\sigma &::= \mathbf{true} \\ [\![\bot]\!]_\sigma &::= \mathbf{false} \\ [\![p]\!]_\sigma &::= \sigma(p) \quad \forall p \in A \\ [\![\neg\phi_1]\!]_\sigma &::= \neg\,[\![\phi_1]\!]_\sigma \\ [\![\phi_1 \wedge \phi_2]\!]_\sigma &::= [\![\phi_1]\!]_\sigma \wedge [\![\phi_2]\!]_\sigma \\ [\![\phi_1 \vee \phi_2]\!]_\sigma &::= [\![\phi_1]\!]_\sigma \vee [\![\phi_2]\!]_\sigma \\ [\![\phi_1 \implies \phi_2]\!]_\sigma &::= \begin{cases} \mathbf{true} & , \text{ if } [\![\phi_1]\!]_\sigma = \mathbf{false} \text{ or } [\![\phi_2]\!]_\sigma = \mathbf{true} \\ \mathbf{false} & , \text{ otherwise} \end{cases} \\ [\![\phi_1 \iff \phi_2]\!]_\sigma &::= \begin{cases} \mathbf{true} & , \text{ if } [\![\phi_1]\!]_\sigma = [\![\phi_2]\!]_\sigma \\ \mathbf{false} & , \text{ otherwise} \end{cases} \\ [\![\phi_1 \oplus \phi_2]\!]_\sigma &::= [\![\phi_1]\!]_\sigma \oplus [\![\phi_2]\!]_\sigma\end{aligned}$$

Definition 2.1. For a given formula ϕ, if $[\![\phi]\!]_\sigma = \mathbf{true}$, then σ is said to be a *model* (or *satisfying assignment*) for ϕ. ϕ is said to be

(1) *valid*, if all valuations σ are models for ϕ.
(2) SAT, if there exists a model for ϕ.
(3) UNSAT, if there exists no model for ϕ.

2.2 Predicate Logic

2.2.1 Syntax. As before, let $\mathbf{V}$ denote an infinite set of variables. Let (F, P) be a pair of sets of *symbols* (the meanings of which will be made clear in a moment); this pair will be called a *signature*. A *term* in predicate logic is any element belonging to the following grammar.

$$\begin{aligned} term := \quad & v \\ | \quad & f(term_1, ..., term_n) \end{aligned}$$

Above, $f \in F$ is any symbol. Any formula in predicate logic is any member element belonging to the following grammar (such formulae are also called *well-defined* formulae).

$$\begin{aligned} \phi := \quad & \top \\ | \quad & \bot \\ | \quad & p(term_1, ..., term_n) \\ | \quad & \neg\phi_1 \\ | \quad & \phi_1 \wedge \phi_2 \\ | \quad & \cdots \\ | \quad & \forall v.\phi_1 \\ | \quad & \exists v.\phi_1 \end{aligned}$$

Above, the dots $(\cdots)$ mean that the usual logical operators are included in the grammar. Also, $p \in P$ is any symbol.

Now, a word about the sets F and P. The elements of F are said to be *function symbols*, and the elements of P are said to be *predicate symbols*. Any symbol $f \in F$ should be thought of as an n-ary function (for some n) $f : D^n \to D$, where D is the domain in which the variables in $\mathbf{V}$ take values. Any symbol $p \in P$ should be thought of as a predicate on D^n (for some n); more formally, a symbol $p \in P$ represents some subset of D^n. Evaluating $p(x_1, ..., x_n)$ for $x_1, ..., x_n \in D$ will then be equivalent to checking whether $(x_1, ..., x_n)$ belongs to that subset; if it does, the predicate is true, otherwise it is false. We will have more to say about these symbols in the next section.

2.2.2 Semantics. We begin first with the definition of an *interpretation* (also referred to as a *model*).

Definition 2.2. An *interpretation* (or *model*) is a triple (D, i_F, i_P) where

- D is a set, the *domain* of interpretation.
- i_F maps every element of F to an n-ary function on D, where n is some positive integer.
- i_P maps every element of P to a subset of D^n.

The above definition combined with the *semantics* (which we're about to define) will justify why elements of F and P are called *function* and *predicate symbols* respectively.

Definition 2.3. Given a formula ϕ and some interpretation/model $M = (D, i_F, i_P)$ and

a valuation σ of the variables in ϕ over the domain D, we define the following.

$$
\begin{aligned}
[\![v]\!]_{M,\sigma} &::= \sigma(v) \\
[\![f(t_1,...,t_n)]\!]_{M,\sigma} &::= i_F(f)([\![t_1]\!]_{M,\sigma},...,[\![t_n]\!]_{M,\sigma}) \\
[\![\top]\!]_{M,\sigma} &::= \textbf{true} \\
[\![\bot]\!]_{M,\sigma} &::= \textbf{false} \\
[\![p(t_1,...,t_n)]\!]_{M,\sigma} &::= ([\![t_1]\!]_{M,\sigma},...,[\![t_n]\!]_{M,\sigma}) \in i_P(p) \\
[\![\neg\phi_1]\!]_{M,\sigma} &::= \neg\,[\![\phi_1]\!]_{M,\sigma} \\
[\![\phi_1 \wedge \phi_2]\!]_{M,\sigma} &::= [\![\phi_1]\!]_{M,\sigma} \wedge [\![\phi_2]\!]_{M,\sigma} \\
&\dots \\
[\![\forall v.\phi_1]\!]_{M,\sigma} &::= \begin{cases} \textbf{true} & , \text{ if } [\![\phi_1]\!]_{M,\sigma[t/v]} = \textbf{true} \quad \forall t \in D \\ \textbf{false} & , \text{ otherwise} \end{cases} \\
[\![\exists v.\phi_1]\!]_{M,\sigma} &::= \begin{cases} \textbf{true} & , \text{ if } [\![\phi_1]\!]_{M,\sigma[t/v]} = \textbf{true} \quad \text{for some } t \in D \\ \textbf{false} & , \text{ otherwise} \end{cases}
\end{aligned}
$$

These definitions are nothing but a formalization of the usual meanings of true/false assignments to formulae.

Definition 2.4. A model M is said to *satisfy* a formula ϕ, which is written $M \models \phi$ if $[\![\phi]\!]_{M,\sigma}$ holds for all valuations σ. ϕ is said to be *satisfiable* if there is a model/interpretation which satisfies it. ϕ is said to be *valid* if it holds for all models and all valuations.

Definition 2.5. Suppose Σ is some signature. Let Γ be a set of Σ-formulas. We write $M \models \Gamma$ to signify that $M \models \phi$ for all $\phi \in \Gamma$. Moreover, if Γ is a set of Σ-formulas and ϕ is another Σ-formula, then Γ is said to *logically imply* ϕ, written $\Gamma \models \phi$, iff for every model M of Σ, if $M \models \Gamma$ then $M \models \phi$. This is just formalizing the notion of logical implication of a set of formulas. With this definition, two formulas ψ and ϕ are said to be *logically equivalent* if $\psi \models \phi$ and $\phi \models \psi$ (note that $\psi \models \phi$ is a shorthand for $\{\psi\} \models \phi$).

2.3 Formal reasoning of programs: Hoare Logic

2.3.1 Hoare Triples. First, we introduce *Hoare Triples*. In all of this discussion, we will assume that our programming language has the syntax and semantics which we defined before, and we will be working with a predicate logic on a given domain.

A *Hoare Triple* is a triple $\{P\}\,Q\,\{R\}$, where P and R are predicates, and Q is a program (i.e some code generated out of the syntax grammar), such that the following holds.

$$\forall x, y : (P(x) \wedge \text{program } Q \text{ terminates on } x \text{ with output } y) \implies R(y)$$

In simple words, for all valuations for the input variables x for which the predicate P holds and the program Q terminates with output y, $R(y)$ must be true. Note the *termination condition*; in particular, if there is no values for the input variables for which Q terminates, the implication follows vacuously (as the left hand side of the implication will be false).

2.3.2 Inference rules for Hoare Triples. In this section, we will introduce inference rules for Hoare logic, which will form the basis of deriving proofs for program correctness.

The first inference rule is for the **skip** command.

$$\frac{}{\{P\}\,\textbf{skip}\,\{P\}} \tag{9}$$

The next inference rule is the *backwards assignment* inference rule.

$$\frac{}{\{P[e/x]\}\, x := e\, \{P(x)\}} \tag{10}$$

Let us now describe the intent behind introducing this rule. We want the postcondition P to be true; clearly, the *weakest precondition* for which P will certainly be true is $P[e/x]$, i.e the predicate P with all occurences of x replaced by e. Certainly, this forms a valid Hoare triple.

Example 2.1. Suppose our program is just a single assignment statement, $x := x + 1$. Suppose the postcondition we want to hold is $x > 0$. So, by the inference rule (10), we see that the *weakest* precondition for which the Hoare triple will be valid is $P[e/x] = (x + 1) > 0$, which is the same as the predicate $x \geq 0$. So, we obtain the Hoare triple $\{x \geq 0\}\, x := x + 1\, \{x > 0\}$.

Next, we introduce the *forward assignment* inference rule. The motivation here is pretty similar to the backwards assignment rule, except that here, given a precondition and a program which is a single assignment statement, we want to determine the *strongest postcondition* which will give us a valid Hoare triple. We now state the inference rule.

$$\frac{}{\{P(x)\}\, x := e[x]\, \{\exists v\, :\, P(v) \wedge x = e[v/x]\}} \tag{11}$$

Let us look at an example for this rule.

Example 2.2. Suppose $P(x)$ is the predicate $x = 5$, and our program is $x := x + 1$. According to the above rule, the *strongest postcondition* is $\exists v\, :\, v = 5 \wedge x = v + 1$; this postcondition implies $x = 5 + 1 = 6$.

Note that the forward assignment inference rule above involves an existential quantifier. We can this inference rule without the existential quantifier by explicitly writing the initial value of the variable x as follows.

$$\frac{}{\{PreCond(x) : x = x_0 \wedge P(x)\}\, x := e[x]\, \{PreCond(x_0) \wedge x_1 = e[x_0/x] \wedge x = x_1\}} \tag{12}$$

This demands an explanation. Above, we have introduced an explicit value x_0, using which the existential quantifier can be dealt with. We let $PreCond(x)$ be the predicate $x = x_0 \wedge P(x)$; in the postcondition, we instantiate the predicate $PreCond$ with the intial value x_0, introduce a new variable $x_1 = e[x_0/x]$ and set $x = x_1$.

Example 2.3. As an example, suppose again that the predicate $P(x)$ is $x = 5$ and the program is $x := x+1$. So, we have $PreCond(x) : x = x_0 \wedge x = 5$. Using the second version of the forward assignment inference rule, we obtain the postcondition $PreCond(x_0) \wedge x_1 = x_0 + 1 \wedge x = x_1$, which is the same as $x_0 = x_0 \wedge x_0 = 5 \wedge x_1 = x_0 + 1 \wedge x = x_1$, which clearly implies that $x = 6$.

Ofcourse, this inference rule can be extended to multiple variables; all we need to remember is to instantiate $PreCond$ using the initial value of the variable which is being assigned to a new value by the program (in the above case, this variable is x).

Next, we will look at the *sequential composition* inference rule. This is straightforward.

$$\frac{\{P\}\, Q\, \{R\} \quad \{R\}\, S\, \{T\}}{\{P\}\, Q; S\, \{T\}} \tag{13}$$

This is just saying that a for a program consisting of two subprograms Q and S, we can derive a Hoare triple using Hoare triples of the two subprograms.

Next, we introduce the *consequence inference* rules. These are also pretty straightforward. The first is the following.

$$\frac{\{P\}\,Q\,\{R\} \qquad P' \implies P}{\{P'\}\,Q\,\{R\}} \tag{14}$$

I.e, if P' is a stronger precondition than P, then any Hoare triple with P as it's precondition can be used to another Hoare triple with P' as its precondition. There is a similar inference rule concerning postconditions.

$$\frac{\{P\}\,Q\,\{R\} \qquad R \implies R'}{\{P\}\,Q\,\{R'\}} \tag{15}$$

2.3.3 Automation via Static Single Assignment (SSA). In this section we will briefly describe how a given program is checked automatically via *static single assignments* (SSA). This is the technique used by bounded model checkers like CBMC. We will explain this using a simple program, whose pseudocode is given below.

$$\begin{aligned} &assume(x > y);\\ &x := x + 1;\\ &x := x + 2;\\ &assert(x > y + 3); \end{aligned}$$

Here, think of $assume(x > y)$ as the CBMC `__CPROVER_assume()` function, and think of $assert(x > y+3)$ as a usual assert. For any program, a bounded model checker generates an SSA formula; for the given problem, the SSA formula will be the following.

$$SSA(x_0, x_1, x_2, y_0) = (x_0 > y_0) \wedge (x_1 = x_0 + 1) \wedge (x_2 = x_1 + 2)$$

Let us now explain this formula. The subscripts below the variables x and y are time stamps for the variable values; for instance, the initial value of x when the program starts is x_0, and the values x_1, x_2 represent the values of x after the corresponding assignment statements. Similarly, y_0 represents the initial value of the variable y. Since y is never reassigned another value, the final value of y after the execution of the program is still y_0. So, the SSA formula is just a *conjunction* of all the assumptions of the program as well as the *static single assignment* formulas, i.e assignment statements in the program.

Having generated the SSA formula for the program, a bounded model checker then generates a *verification condition* (VC) for *every assertion* in the program. In the above program, the VC formula will be the following.

$$SSA(x_0, x_1, x_2, y_0) \implies (x_2 > y_0 + 3)$$

Putting the actual value of the SSA formula, the VC formula becomes

$$[(x_0 > y_0) \wedge (x_1 = x_0 + 1) \wedge (x_2 = x_1 + 2)] \implies (x_2 > y_0 + 3)$$

So, the VC formula can be thought of as being a formula of the form

$$\text{assumptions} \implies \text{assertion}$$

After generating the VC formula for the program, a bounded model checker will then check if the VC formula is *valid* (recall **Definition** 2.4). Checking validity of VC is the same as checking whether ¬VC is UNSAT, i.e checking whether there is *no* model and valuation which satisfies VC. Note that the negation of VC is the formula

$$\text{assumptions} \implies \neg\text{assertion}$$

So, a bounded model checker will check whether UNSAT[assumptions $\implies$ ¬assertion] is true; this work is deligated to an SMT solver.

2.3.4 Conditional Forward Rule. Next, we will see an inference rule for conditional statements.

$$\frac{\{P \wedge B\}\, Q\, \{R\} \qquad \{P \wedge \neg B\}\, S\, \{R\}}{\{P\}\, \textbf{if } B \textbf{ then } Q \textbf{ else } S \textbf{ end if}\, \{R\}} \tag{16}$$

2.3.5 SSA for programs with conditional commands. We will now see via an example how an SSA for a program with conditional commands looks like. Suppose the program is the following.

$assume(\textbf{true})$
if $x > y$ **then**
    $x := y;$
else
    $x := x;$
end if
$assert(x \leq y);$

The SSA for this program will look something like the following.

$$SSA = (x_1 = y_0) \wedge (x_2 = x_0) \wedge (x_3 = (x_0 > y_0)?x_1 : x_2)$$

Note that third conjunct in the above formula; it is called a *merge*. For programs which are branched, i.e programs will can follows different execution branches induced via conditions, the bounded model checker adds a *merge* for every variable changed in a condition block. The VC for the above program will be the following.

$$(x_1 = y_0) \wedge (x_2 = x_0) \wedge (x_3 = (x_0 > y_0)?x_1 : x_2) \implies x_3 \leq y_0$$

This formula will then be delegated to an SMT solver, which will check whether it is UNSAT.

2.3.6 Handling loops. Next, we introduce another inference rule, called the *iteration* inference rule.

$$\frac{\{P \wedge B\}\, Q\, \{P\}}{\{P\}\, \textbf{while } B \textbf{ do } Q \textbf{ end while}\, \{P \wedge \neg B\}} \tag{17}$$

This rule is also straightforward and needs little explanation.

The next question is, how do bounded model checkers handle a program with loops? The answer is something known as *unwinding a loop*. A bounded model checker will *unwind* a given loop to a certain depth, known as the *unwind depth*; *unwinding* means flattening out the loop into a straight program. Hence these model checkers are called *bounded* model checkers, because the unwind depth is usually passed as a parameter to the model checker. Moreover, model checkers use the notion of an *inductive loop invariant* (LI), which we will next introduce.

A predicate LI inside a loop is said to be *inductive* if it is true for all iterations, i.e $\{LI \wedge LC\}\, Body\, \{LI\}$ must be a valid Hoare Triple. Here LC is the loop condition and $Body$ is the body of the loop. Additionally, LI is said to be an *inductive loop invariant* if it also holds before the loop, i.e $\{PreCond\}\, PreLoopCode\, \{LI\}$ is a valid Hoare triple. For a given property P, LI is said to be a *safe* inductive invariant if it is able to prove P on loop exit, i.e the formula $(LI \wedge \neg LC) \implies P$ is valid.

2.3.7 Inductive Loop Invariants for Transition Systems. Suppose T is a transition system (i.e, T relates a state s to a state s', where a *state* is just a state of the program in question. For most purposes we will assume that a transition system will be represented as a predicate formula. To see examples of this, see solutions to HW-2). A predicate P is said to be *inductive* for T if

$$P(s) \wedge T(s, s') \implies P(s')$$

where s, s' are states. The above formula is also expressed by saying that *the image of set of all P states is closed under T.*

In the upcoming section, we will find a systematic way of computing inductive loop invariants for loops using *fixed point theory.*

3. Lattices and Fixed Point Theory

3.1 Overview

3.1.1 Total and Partial Orders. For the sake of completeness we will include the definitions of total and partial orders here.

Definition 3.1. A *totally ordered set* is a tuple $(P, \leq)$ where $\leq \subseteq P \times P$ is a total order, i.e it holds that:
(1) $\leq$ is *total*, i.e $a \leq b$ or $b \leq a$ for all $a, b \in P$.
(2) $\leq$ is *transitive*, i.e $a \leq b$ and $b \leq c$ implies $a \leq c$ for all $a, b, c \in P$.
(3) $\leq$ is *anti-symmetric*, i.e $a \leq b$ and $b \leq a$ implies $a = b$ for all $a, b, c \in P$.

Definition 3.2. A *partially ordered set* is a tuple $(P, \preceq)$ where $\preceq \subseteq P \times P$ is a partial order, i.e it holds that:
(1) $\preceq$ is *reflexive*, i.e $a \preceq a$ for all $a \in P$.
(2) $\preceq$ is *transitive*, i.e $a \preceq b$ and $b \preceq c$ implies that $a \preceq c$ for all $a, b, c \in P$.
(3) $\preceq$ is *anti-symmetric*, i.e $a \preceq b$ and $b \preceq a$ implies $a = b$ for all $a, b \in P$.

Definition 3.3 (Join). Let $(P, \preceq)$ be a poset. An **upper bound** of a set $M \subseteq P$ is an element $u \in P$ such that $x \preceq u$ for all $x \in M$. The *least upper bound*, denoted $\sqcup M$, is an element $u \in P$ such that
(1) u is an upper bound of M.
(2) $u \preceq u'$ for all upper bounds u' of M.
The operator $\sqcup$ is known as the *join operator.*

Definition 3.4 (Dual of Join). A *lower bound* and a *greatest lower bound* of a set $M \subseteq P$ are defined analogously. The greatest lower bound of M is denoted by $\sqcap M$. The operator $\sqcap$ is called the *meet operator.*

3.1.2 Lattices. We will now define *lattices.*

Definition 3.5. A *semi lattice* is a tuple $(A, \preceq, \sqcup)$ where $(A, \preceq)$ is a poset with join $\sqcup$. A *complete semi-lattice* is a lattice $(A, \preceq, \sqcup)$ such that the join $\sqcup B$ of every (non-empty) subset $B \subseteq A$ exists in A. A *lattice* is a semi-lattice $(A, \preceq, \sqcup)$ with additionally a meet $\sqcap$ operator, and analogously a *complete lattice* is defined.

Usually, given a lattice A, when $\sqcup A$ exists, we call it the *top element*, and when $\sqcap A$ exists, we call it the *bottom element.* These are examples of *maximal* and *minimal* elements in a poset.

Definition 3.6 (**Monotone Function**)**.** Let $(X, \preceq_X)$ and $(Y, \preceq_Y)$ be posets and let $f : X \to Y$ be a total function. f is said to be monotone if for all $a, b \in X$

$$a \preceq_X b \implies f(a) \preceq_Y f(b)$$

Definition 3.7 (**Fixed Points**)**.** Let X be a poset. For a function $f : X \to X$, $Fix(f)$, the set of *fixed points* of f, is defined to be

$$Fix(f) = \{x \in X \mid f(x) = x\}$$

3.1.3 Knaster-Tarski Theorem. In this section, we will mention (without proof) a theorem in lattice theory that characterises fixed points of a monotone function.

Theorem 3.1 (**Knaster-Tarski**)**.** *Let $(L, \preceq)$ be a complete lattice. Then, the least fixed point of a monotone function $f : L \to L$ always exists and is given by*

$$lfp(f) = \bigsqcap Red(f)$$

where $Red(f) := \{x \in L \mid f(x) \preceq x\}$. Similarly, the greatest fixed point of f always exists and is given by

$$gfp(f) = \bigsqcup Ext(f)$$

where $Ext(f) := \{x \in L \mid x \preceq f(x)\}$.

Proof. We will only prove the first part of the theorem, as the second part can be proven similarly.

First, define

$$lfp(f) = \bigsqcap Fix(f)$$

We will do the proof in two parts, wherein we will prove that

- $lfp(f) \in Red(f)$.
- $\bigsqcap Red(f)$ is a fixed point.

Let us show the first part. Note that $lfp(f)$ is a lower bound of $Fix(f)$ by definition. So, for any $y \in Fix(f)$, we have that

$$lfp(f) \preceq y$$

Because f is monotonic, it follows that

$$f(lfp(f)) \preceq f(y) = y$$

Clearly, this implies that $f(lfp(f))$ is a lower bound of $Fix(f)$ too. But because $lfp(f)$ is the *greatest* lower bound, it must be the case that $f(lfp(f)) \preceq lfp(f)$, which implies that $lfp(f) \in Red(f)$.

Now, let's prove the second bullet point, namely that $\bigsqcap Red(f)$ is a fixed point of f. To prove this, we will show that $f\left(\bigsqcap Red(f)\right) \preceq \bigsqcap Red(f)$ and $\bigsqcap Red(f) \preceq f\left(\bigsqcap Red(f)\right)$.

(1) Note that $\bigsqcap Red(f) \preceq x$ for every $x \in \bigsqcap Red(f)$, which is true by definition. Since f is monotone, this means that $f\left(\bigsqcap Red(f)\right) \preceq f(x) \preceq x$ for every $x \in Red(f)$. Since $\bigsqcap Red(f)$ is the greatest lower bound of $Red(f)$, this implies that $f\left(\bigsqcap Red(f)\right) \preceq \bigsqcap Red(f)$.

(2) The above inequality implies that $f\left(\bigsqcap Red(f)\right) \in Red(f)$. But this clearly means that $\bigsqcap Red(f) \preceq f\left(\bigsqcap Red(f)\right)$.

Hence, $\bigsqcap Red(f)$ is indeed a fixed point of f.

Finally, note that because $\bigsqcap Red(f) \in Fix(f)$, we have that $lfp(f) \preceq \bigsqcap Red(f)$. And, since $lfp(f) \in Red(f)$, we have that $\bigsqcap Red(f) \preceq lfp(f)$. This proves that $lfp(f) = \bigsqcap Red(f)$, and completes the proof of the theorem. ■

Example 3.1. Let us see an example of this theorem in the reachability problem. Suppose we are given a transition system T with state space S, a set of initial states I and a transition relation $R \subseteq S \times S$. We will work with the lattice $(2^S, \subseteq)$, which is clearly a complete lattice. Define the *post* operator $post : 2^S \to 2^S$ as follows, where $X \in 2^S$.

$$X \mapsto \left\{s' \in S \mid \exists\ s \in X \text{ s.t } R(s, s')\right\}$$

In other words, $post(X)$ is precisely the set containing all states reachable from a state in X in one step of the transition system. In general, the set of k-reachable states will be $post^k(I)$. Then consider the monotonic function f on this lattice defined by

$$f(X) = I \cup post(X)$$

Note that

$$\begin{aligned}
f^0(\phi) &= \phi \\
f^1(\phi) &= I \cup post(\phi) = I \\
f^2(\phi) &= I \cup post(I) \\
f^3(\phi) &= I \cup post(I \cup post(I)) = I \cup post(I) \cup post^2(I)
\end{aligned}$$

and in general one can observe that $f^k(I) = I \cup post(I) \cup \cdots \cup post^{k-1}(I)$. In particular, if one can find k such that $f^k(\phi) \subseteq f^{k-1}(\phi)$, one will have found a fixed point of f. We will explore this method of repeated iteration to compute fixed points in the next section.

3.1.4 Continuous Functions and Kleene's Fixed Point Theorem. First, we will define *continuous functions* in the context of lattices.

Definition 3.8. Let X, Y be posets and let $f : X \to Y$ be some function. Assume that f is monotone (though this is strictly not needed).

(1) f is said to be *semi-$\sqcup$-continuous* if it preserves upper bounds of ascending chains, i.e for each ascending chain C of X, we have

$$\bigsqcup_Y \{f(X) \mid X \in C\} = f\left(\bigsqcup_X C\right)$$

(2) f is said to be *semi-$\sqcap$-continuous* if it preserves upper bounds of ascending chains, i.e for each ascending chain C of X, we have

$$\bigsqcap_Y \{f(X) \mid X \in C\} = f\left(\bigsqcap_X C\right)$$

Theorem 3.2 (**Kleene's Fixed Point Theorem**). *Let S be a complete lattice and let $f, g : S \to S$ be monotone functions (w.r.t the partial order on S). Further, suppose f is semi-$\sqcup$-continuous and g is semi-$\sqcap$-continuous. Then,*

$$\begin{aligned}
lfp(f) &= \bigsqcup_{k \geq 0} f^k(\bot) \\
gfp(g) &= \bigsqcap_{k \geq 0} g^k(\top)
\end{aligned}$$

Proof. Again, we will only prove this for the lfp, as the proof of the gfp is similar.

Let $\tilde{x} = \bigsqcap Red(f)$ as per Tarski's characterisation of lfp. We show that $\tilde{x} \preceq \bigsqcup_{k\geq 0} \{f^k(\bot)\}$ and that $\bigsqcup_{k\geq 0} \{f^k(\bot)\} \preceq \tilde{x}$, and that will prove the claim.

(1) First, let us show that $\tilde{x} \preceq \bigsqcup_{k\geq 0} \{f^k(\bot)\}$. Note that for every $k \geq 0$, we have that $f^k(\bot) \preceq f^{k+1}(\bot)$; this follows from the monotonicity of f and the fact that $\bot$ is the smallest element. This implies by the continuity of f that

$$f\left(\bigsqcup_{k\geq 0}\left\{f^k(\bot)\right\}\right) = \bigsqcup_{k\geq 0}\left\{f^{k+1}(\bot)\right\} \preceq \bigsqcup_{k\geq 0}\left\{f^k(\bot)\right\}$$

The above inequality implies that $\bigsqcup_{k\geq 0} \{f^k(\bot)\} \in Red(f)$. And so, naturally we get that $\tilde{x} \preceq \bigsqcup_{k\geq 0} \{f^k(\bot)\}$.

(2) Next, let us show that $\bigsqcup_{k\geq 0} \{f^k(\bot)\} \preceq \tilde{x}$. To show this, we will show that for every $k \geq 0$, it is the case that $f^k(\bot) \preceq \tilde{x}$. The proof will be by induction on k.
- The base case $k = 0$ is trivial, since $\bot$ is the smallest element.
- Now suppose $f^k(\bot) \preceq \tilde{x}$ for some k. By the monotonicity of f, this implies that

$$f(f^k(\bot)) \preceq f(\tilde{x}) = \tilde{x}$$

where the last equality is true because $\tilde{x}$ is a fixed point of f. This proves the inductive case.

Hence, the inequality follows.

From the above two inequalities, the claim follows. ■

Example 3.2. In **Example** 3.1 we saw a monotonic increasing function f on the lattice $(2^S, \subseteq)$ corresponding to a transition system. Now, suppose P is some predicate, and we interpret P as an element of 2^S (it contains all those states which satisfy the predicate). Now, define a function $g : 2^S \to 2^S$ as follows (here $\top = S$ is the greatest element).

$$X \mapsto (P \cap \top) \cap \left\{s \in X \mid \forall s' \in S, \neg R(s, s') \vee s' \in P\right\}$$

In other words, the function g removes from a set X all those elements which violate P and all those elements whose one-step successor in the transition system violate P.

4. Abstract Interpretation

4.1 Abstractions

4.1.1 Introduction. The idea of *abstractions* is one of the central ideas in static analysis. We will formalize this notion in the upcoming section, but we can look at an example right now.

Example 4.1. Suppose our program only has one variable (let's call it x) which can take values in the domain $\{1, 2, 3\}$. This is to say that at any point in the program, the variable is *guaranteed* to have a value in this set. Now, we want to keep track of the *range* of values in which x can lie; for this, we will use intervals. Let A be the set of all intervals of the form $[a, b]$, where a, b are integers and $1 \leq a \leq b \leq 3$. Consider the map $\alpha : 2^S \to A$ given by

$$X \mapsto [\min X, \max X]$$

Here, $S = \{1, 2, 3\}$ and $X \subseteq S$. In this case, α is said to be an *abstract interpretation*. We will further formalize this notion in the next section.

In the above example, the set $S = \{1, 2, 3\}$ is called the *concrete space*, and the set A (the set of intervals) is called the *abstract space*.

4.1.2 Galois Connections. Let C, A be partial orders (where we will assume C is the concrete space and A is the abstract space). Let $\alpha : C \to A$ and $\gamma : A \to C$ be mappings such that the following hold.

- α is a monotone function; it is called a monotone *abstraction function*.
- γ is a monotone function; this is also called the *concretizer function*.
- For all $c \in C$, it must be the case that $c \preceq \gamma(\alpha(c))$; in other words, γ bounds from above.
- For all $a \in A$, it must be the case that $\alpha(\gamma(a)) \preceq a$; in other words, α bounds from below.

In such a case, the maps α and γ are set to form a *Galois connection*.

Spelled out in words, the above definition is saying the following.

- γ maps an abstract element a to a concrete element c that is greater or equal to than every element that gets abstracted to the same a.
- α maps a concrete c to an abstract a that is less than or equal to every element that gets concretized to the same c.

Definition 4.1 (**Over-approximate transformers**)**.** Let C, A be partial orders as above. Let $f : C \to C$, $f^\# : A \to A$ be monotone functions. $f^\#$ is said to be an *over-approximation* of f if and only if for all $a \in A$, it is true that

$$f(\gamma(a)) \preceq \gamma(f^\#(a))$$

Now, suppose $f^\#$ is an over-approximation of f. Then we know that $f(\gamma(a)) \preceq \gamma(f^\#(a))$. Applying α to both sides, we see that

$$(\alpha \circ f \circ \gamma)(a) \preceq \alpha(\gamma(f^\#(a)))$$

But because α-γ form a Galois connection, we know that $\alpha(\gamma(f^\#(a))) \preceq f^\#(a)$. So, we see that

$$(\alpha \circ f \circ \gamma)(a) \preceq f^\#(a)$$

This shows that the *most precise* over-approximation of the function f is the function $\alpha \circ f \circ \gamma$.

4.1.3 Ingredients of Abstract Interpretation for Interval Domains. Having formalized the notion of abstract interpretation and over-approximate transformers, we will now build these elements over the domain of intervals.

Our abstract domain will be the lattice of intervals $\mathbf{I}(\mathbf{R}^n)$ which is defined as the set

$$\mathbf{I}(\mathbf{R}^n) = \{\bot\} \cup \{[I, u] \mid I \in \mathbf{R} \cup \{-\infty\}, u \in \mathbf{R} \cup \{\infty\}, I \leq u\}^n$$

Also, we use the notation $\top = (-\infty, \infty)^n$ and $\bot = (\infty, -\infty)^n$. For the notation, notation like X, Y will denote elements of the concrete space, while symbols like $X^\#, Y^\#$ will be used to denote elements of the abstract domain.

The partial ordering on the interval domain is defined in the most straightforward way.

$$X^\# \preceq Y^\# = \begin{cases} \textbf{true} & \text{if } X^\# = \bot \\ \textbf{false} & \text{if } X^\# \neq \bot \wedge Y^\# = \bot \\ \bigwedge_{i \in [1,n]} I_i^Y \leq I_i^X \leq u_i^X \leq u_i^Y & \text{otherwise} \end{cases}$$

The meet operator $\sqcap$ on two intervals returns the largest interval contained in both the operands, and the join operator $\sqcup$ returns the smallest interval that contains both the operands. Due to time constraints, I wasn't able to complete this section.