

## Contents

<b>1</b>	<b>Question - Logical Formula Classification</b>	<b>2</b>
1.1	Theory . . . . .	2
1.2	Solution . . . . .	2
<b>2</b>	<b>Question Equisatisfiability-Equivalence</b>	<b>3</b>
2.1	Theory . . . . .	3
2.2	Solution . . . . .	3
<b>3</b>	<b>Question - Theory of Equality</b>	<b>4</b>
3.1	Theory . . . . .	4
3.2	Solution . . . . .	5
<b>4</b>	<b>Question - Theory of Arrays</b>	<b>6</b>
4.1	Theory . . . . .	6
4.2	Solution . . . . .	7
<b>5</b>	<b>Question - Structures</b>	<b>7</b>
5.1	Theory . . . . .	7
5.2	Solution . . . . .	8
<b>6</b>	<b>Question - Language Semantics</b>	<b>9</b>
6.1	Theory . . . . .	9
6.2	Solution . . . . .	9
<b>7</b>	<b>Question - Hoare Logic</b>	<b>10</b>
7.1	Theory . . . . .	10
7.2	Solution . . . . .	10
<b>8</b>	<b>Question - Hoare Rules</b>	<b>11</b>
8.1	Theory . . . . .	11
8.2	Solution . . . . .	12
<b>9</b>	<b>Question - Verification Condition Generation</b>	<b>13</b>
9.1	Theory . . . . .	13
9.2	Solution . . . . .	13
<b>10</b>	<b>Question - Horn SAT/UNSAT</b>	<b>13</b>
10.1	Theory . . . . .	13
10.2	Solution . . . . .	14
<b>11</b>	<b>Question - Proving Correctness</b>	<b>15</b>
11.1	Solution . . . . .	16
<b>12</b>	<b>Question - Solving Horn Clauses</b>	<b>16</b>
12.1	Theory . . . . .	16
12.2	Solution . . . . .	17
<b>13</b>	<b>Question - IFC Types</b>	<b>18</b>
13.1	Theory . . . . .	18
13.2	Solution . . . . .	19
<b>14</b>	<b>Question - Security Attacker Model</b>	<b>20</b>
14.1	Theory . . . . .	20
14.2	Solution . . . . .	20

# 1 Question - Logical Formula Classification

Consider the following formula in propositional logic

$$F \triangleq (p \rightarrow q) \wedge (\neg q \rightarrow p)$$

Is the formula contingent, unsatisfiable, or valid? If the formula is satisfiable, give satisfying assignment.

## 1.1 Theory

The central thing to remember for this question revolves around how a formula evaluates with respect to an interpretation. As a reminder, an interpretation is just a mapping from propositional variables to truth values (True or False). So in this instance an interpretation could be

$$I = \{p \mapsto \text{True}, q \mapsto \text{False}\}$$

Naturally under certain assignments the formula will evaluate to True, and under others it will evaluate to False. We classify these kinds of assignments as follows:

- If a formula evaluates to True for a *specific interpretation*, we say that the formula is **satisfiable**.
- If a formula evaluates to False for a *specific interpretation*, we say that the formula is **unsatisfiable**.

Likewise we can ask ourselves for *how many* interpretations a formula evaluates to True or False. This leads us to the following classifications:

- If a formula evaluates to True for *every possible interpretation*, we say that the formula is **valid**.
- If a formula evaluates to False for *every possible interpretation*, we say that the formula is **unsatisfiable**.

Finally if a formula evaluates to True for *some* interpretations and False for *others*, we say that the formula is **contingent**. If there exists a satisfying assignment or interpretation for a formula we denote this as

$$I \models F \quad (\text{Entails})$$

We can express these classifications as a piecewise function:

$$\text{classifications}(F) = \begin{cases} \text{satisfying} & \text{if } \exists I, I \models F \\ \text{unsatisfying} & \text{if } \exists I, I \not\models F \\ \text{valid} & \text{if } \forall I, I \models F \\ \text{unsatisfiable} & \text{if } \forall I, I \not\models F \\ \text{contingent} & \text{otherwise} \end{cases}$$

## 1.2 Solution

To determine whether the formula  $F$  is contingent, unsatisfiable, or valid, we can evaluate it under all possible interpretations of the propositional variables  $p$  and  $q$ . There are four possible interpretations:

- $I_1 : p = \text{True}, q = \text{True}$
- $I_2 : p = \text{True}, q = \text{False}$
- $I_3 : p = \text{False}, q = \text{True}$
- $I_4 : p = \text{False}, q = \text{False}$

Now we evaluate  $F$  under each interpretation:

- Under  $I_1$ :

$$F = (\text{True} \rightarrow \text{True}) \wedge (\neg \text{True} \rightarrow \text{True}) = \text{True} \wedge \text{True} = \text{True}$$

- Under  $I_2$ :

$$F = (\text{True} \rightarrow \text{False}) \wedge (\neg \text{False} \rightarrow \text{True}) = \text{False} \wedge \text{True} = \text{False}$$

- Under  $I_3$ :

$$F = (\text{False} \rightarrow \text{True}) \wedge (\neg \text{True} \rightarrow \text{False}) = \text{True} \wedge \text{False} = \text{False}$$

- Under  $I_4$ :

$$F = (\text{False} \rightarrow \text{False}) \wedge (\neg \text{False} \rightarrow \text{False}) = \text{True} \wedge \text{False} = \text{False}$$

From the evaluations, we see that  $F$  is True under  $I_1$  and False under  $I_2, I_3$ , and  $I_4$ . Therefore, the formula  $F$  is **contingent**, as it evaluates to True for some interpretations and False for others. A satisfying assignment is  $I_1 : p = \text{True}, q = \text{True}$ .

## 2 Question Equisatisfiability-Equivalence

Consider the following two formulas

$$F \triangleq (p \rightarrow q) \quad \text{and} \quad G \triangleq (\neg p \vee q)$$

Which of the following statements are true?

- A.  $F$  and  $G$  are equisatisfiable, but not equivalent.
- B.  $F$  and  $G$  are equivalent, but not equisatisfiable.
- C.  $F$  and  $G$  are both equisatisfiable and equivalent.
- D.  $F$  and  $G$  are neither equisatisfiable nor equivalent.

### 2.1 Theory

A general question which frequently appears in different branches of mathematics is the ways in which we can relate two objects. Typically, these relationships range from very weak relationships (e.g., sharing a common property) to very strong relationships (e.g., being identical). In propositional logic, two important relationships between formulas are equisatisfiability and equivalence.

#### Definition 2.1. Equisatisfiable

Two formulas  $F$  and  $G$  are said to be equisatisfiable if either both formulas are satisfiable or both formulas are unsatisfiable. In other words, there exists an interpretation  $I$  such that  $I \models F$  if and only if there exists an interpretation  $I'$  such that  $I' \models G$ . Formally

$$F \text{ and } G \text{ are equisatisfiable} \iff (\exists I, I \models F) \iff (\exists I', I' \models G) \quad (\text{Equisatisfiability})$$

The intuitive idea behind equisatisfiability is that both formulas have the same "satisfiability status" - either both can be satisfied by some interpretation, or neither can be satisfied by any interpretation.

#### Definition 2.2. Equivalence

Two formulas  $F$  and  $G$  are said to be equivalent if they evaluate to the same truth value under every possible interpretation. Formally,  $F$  and  $G$  are equivalent if for every interpretation  $I$ ,

$$I \models F \iff I \models G \quad (\text{Equivalence})$$

The key difference between equisatisfiability and equivalence is that equisatisfiability only requires that both formulas have the same satisfiability status (either both satisfiable or both unsatisfiable), while equivalence requires that the formulas have the same truth value for every possible interpretation.

### 2.2 Solution

To determine the relationship between the formulas  $F \triangleq (p \rightarrow q)$  and  $G \triangleq (\neg p \vee q)$ , we can analyze their truth tables.

- For  $F$ :

$p$	$q$	$F = (p \rightarrow q)$
True	True	True
True	False	False
False	True	True
False	False	True

- For G:

p	q	$G = (\neg p \vee q)$
True	True	True
True	False	False
False	True	True
False	False	True

From the truth tables, we can see that both F and G evaluate to the same truth values for all possible interpretations of p and q. Therefore, F and G are equivalent. Since equivalent formulas are always equisatisfiable, we conclude that F and G are both equisatisfiable and equivalent. Thus, the correct answer is:

C. F and G are both equisatisfiable and equivalent.

### 3 Question - Theory of Equality

Consider the first-order theory of equality  $T_E$ . Let's say our universe is

$$U \triangleq \{a, b, c\}$$

Besides the built-in equality predicate, our signature contains the function symbol f. We are given the following interpretation I over =.

$$I(=) \triangleq \{(a, a), (b, b), (c, c), (a, c), (c, a)\}$$

and the following interpretation for f:

$$I(f) \triangleq \{a \rightarrow c, c \rightarrow c, b \rightarrow a\}$$

Finally, we're given the following formula

$$G \triangleq \exists y. (f(x) = y \wedge x = f(y))$$

and a variable assignment  $\sigma$

$$\sigma(x) = c \quad \text{and} \quad \sigma(y) = b$$

We define a model  $M \triangleq (U, I, \sigma)$ . Which of the following statements is true?

- A. The interpretation I violates the axioms of  $T_E$ , and formula G holds on the model M.
- B. The interpretation I satisfies the axioms of  $T_E$ , and formula G does not hold on the model M.
- C. The interpretation I satisfies the axioms of  $T_E$ , and formula G holds on the model M.
- D. The interpretation I violates the axioms of  $T_E$ , and formula G does not hold on the model M.

#### 3.1 Theory

As a reminder, *first-order theories* are formalizations of different kinds of mathematical structures using first-order logic. The *theory of equality*, denoted as  $T_E$ , is a fundamental first-order theory that deals with the properties of equality. The signature  $\Sigma_E$ , which is defined by a set of *constants*, *function*, and *predicate symbols*, contains the equality predicate = and additional symbols as needed. So

$$\Sigma_E = \{=, a, b, c, \dots\}$$

A theory must also define a set of axioms to enable reasoning about our structures and interpretations. The most important axioms of the theory of equality  $T_E$  are

$$\begin{aligned} \forall x. x = x & \quad \text{(Reflexivity)} \\ \forall x. \forall y. (x = y) \rightarrow (y = x) & \quad \text{(Symmetry)} \\ \forall x. \forall y. \forall z. (x = y \wedge y = z) \rightarrow (x = z) & \quad \text{(Transitivity)} \end{aligned}$$

We also have function and predicate congruence axioms

$$\forall \bar{x}. \forall \bar{y}. \left( \bigwedge_{i=1}^n \bar{x}_i = \bar{y}_i \right) \rightarrow (f(\bar{x}) = f(\bar{y})) \quad (\text{Function Congruence})$$

$$\forall \bar{x}. \forall \bar{y}. \left( \bigwedge_{i=1}^n \bar{x}_i = \bar{y}_i \right) \rightarrow (P(\bar{x}) \leftrightarrow P(\bar{y})) \quad (\text{Predicate Congruence})$$

Some important related notions are

**Definition 3.1.**  $\Sigma$ -formula

A  $\Sigma$ -formula is a formula constructed using the symbols from the signature  $\Sigma$  along with logical connectives and quantifiers. In the context of the theory of equality  $T_E$ , a  $\Sigma_E$ -formula may include the equality predicate  $=$ , constants, function symbols, and variables. For example

$$\forall x. \exists y. (f(x) = y \wedge x = f(y))$$

Another good question to ask is what constitutes a valid formula in a particular theory.

**Definition 3.2.** Valid in the theory  $T$

A  $\Sigma$ -formula  $F$  is said to be valid in the theory  $T$  if it holds true in every model of the theory  $T$ . In other words, for every interpretation  $I$  that satisfies the axioms of the theory  $T$ , the formula  $F$  evaluates to true under that interpretation. If we denote the set of axioms as  $\mathcal{A}$  we can denote this as a predicate in the form

$$\text{valid}_T(F) \iff \forall I. (\forall A \in \mathcal{A}. I \models A) \rightarrow (I \models F) \quad (\text{Validity in } T)$$

### 3.2 Solution

To determine the truth of the statements regarding the interpretation  $I$  and the formula  $G$  in the model  $M$ , we need to analyze both the axioms of the theory of equality  $T_E$  and the evaluation of the formula  $G$  under the given interpretation and variable assignment.

**Checking the Axioms of  $T_E$ :**

- *Reflexivity*: For all  $x \in U$ , we have  $x = x$ . This holds true for  $a, b, c$  since  $I(=)$  includes  $(a, a), (b, b), (c, c)$ .
- *Symmetry*: For all  $x, y \in U$ , if  $x = y$ , then  $y = x$ . This holds true since  $I(=)$  includes  $(a, c)$  and  $(c, a)$ .
- *Transitivity*: For all  $x, y, z \in U$ , if  $x = y$  and  $y = z$ , then  $x = z$ . This holds true as there are no violations in the provided pairs.

Since all axioms of  $T_E$  are satisfied by the interpretation  $I$ , we conclude that  $I$  satisfies the axioms of  $T_E$ . From this we can also observe two equivalence classes

$$[a] = \{a, c\} \quad \text{and} \quad [b] = \{b\}$$

**Evaluating the Formula  $G$ :**

The formula  $G$  is given by

$$G \triangleq \exists y. (f(x) = y \wedge x = f(y))$$

We need to evaluate  $G$  under the model  $M$  with the variable assignment  $\sigma(x) = c$ .

- First, we compute  $f(x)$  under the assignment  $\sigma$ :

$$f(c) = c \quad (\text{from } I(f))$$

- Next, we need to find a  $y \in U$  such that

$$f(c) = y \wedge c = f(y)$$

Substituting  $f(c) = c$ , we have

$$c = y \wedge c = f(y)$$

- Now we check each  $y \in U$ :

$$y = a. c = a \wedge c = f(a) \quad (\text{False})$$

$$y = b. c = b \wedge c = f(b) \quad (\text{False})$$

$$y = c. c = c \wedge c = f(c) \quad (\text{True})$$

Since there exists a  $y \in U$  (specifically  $y = c$ ) that satisfies the formula  $G$  under the model  $M$ , we conclude that  $G$  holds on the model  $M$ .

**Final Conclusion:** Based on the analysis, we find that the interpretation  $I$  satisfies the axioms of  $T_E$ , and the formula  $G$  holds on the model  $M$ . Therefore, the correct statement is:

C. The interpretation  $I$  satisfies the axioms of  $T_E$ , and formula  $G$  holds on the model  $M$ .

## 4 Question - Theory of Arrays

Consider the following formula in the combined theory of arrays and integer arithmetic

$$(\forall j. a[j] = 2) \rightarrow (b = a\langle j \triangleleft 3 \rangle) \rightarrow a[i] = 2$$

Which of the following statements is true?

- A. The formula is not a sentence and is not valid.
- B. The formula is a sentence but is not valid.
- C. The formula is a sentence and is valid.
- D. The formula is not a sentence but is valid.

### 4.1 Theory

We already know what validity means from the previous questions, so we just have to understand two things here, what a sentence is in first-order logic, and what the theory of arrays is about.

**Definition 4.1.** Sentence

A *sentence* as defined in the course textbook is quite literally just a formula. Or more pedantically a sequence of symbols and connectives following the syntax rules of first-order logic.

**Definition 4.2.** Theory of Arrays

As we talked about before, a first-order theory always has two key components, a signature and a set of axioms. The signature for the theory of arrays  $T_A$  typically includes:

$$\Sigma_A = \{-[-], -\langle - \triangleleft - \rangle, =\} \quad (\text{Signature of Arrays})$$

where the ‘-’ just represents a placeholder for the array variable. So our signature denotes 3 key operations

- *Read* ( $a[i]$ ): This operation retrieves the value stored at index  $i$  in array  $a$ .
- *Write* ( $a\langle i \triangleleft v \rangle$ ): This operation creates a new array that is identical to array  $a$  except that the value at index  $i$  is updated to  $v$ .
- *Equality* ( $=$ ): This operation checks whether two arrays are equal, meaning they have the same values at all indices.

The axioms of the theory of arrays include the equality axioms we discussed earlier, along with the following array-specific axioms:

$$\forall a, i, j. i = j \rightarrow a[i] = a[j] \quad (\text{Array congruence})$$

$$\forall a, i, j, v. i \neq j \rightarrow a\langle i \triangleleft v \rangle[j] = a[j] \quad (\text{Read-Over-Write 1})$$

$$\forall a, i, v. a\langle i \triangleleft v \rangle[i] = v \quad (\text{Read-Over-Write 2})$$

The intuition here should be somewhat obvious, the first axiom states that equal indices in an array should yield equal values. The second axiom states that if we write a value to an index  $i$ , reading from a different index  $j$  should yield the original value at  $j$ . The third axiom states that if we write a value to an index  $i$ , reading from index  $i$  should yield the newly written value.

## 4.2 Solution

To determine the truth of the statements regarding the formula in the combined theory of arrays and integer arithmetic, we need to analyze both whether the formula is a sentence and whether it is valid.

**Checking if the Formula is a Sentence:** The given formula is:

$$(\forall j. a[j] = 2) \rightarrow (b = a[j \triangleleft 3]) \rightarrow a[i] = 2$$

This formula is constructed using logical connectives and quantifiers, following the syntax rules of first-order logic. Therefore, it is indeed a sentence.

**Checking Validity:** To determine if the formula is valid, we need to check if it holds true under all possible interpretations in the combined theory of arrays and integer arithmetic.

- The antecedent  $\forall j. a[j] = 2$  states that all elements of array  $a$  are equal to 2.
- The second part  $b = a[j \triangleleft 3]$  indicates that array  $b$  is obtained by updating array  $a$  at index  $j$  to the value 3.
- The consequent  $a[i] = 2$  states that the element at index  $i$  of array  $a$  is equal to 2.

Now, if the antecedent  $\forall j. a[j] = 2$  is true, then all elements of array  $a$  are indeed 2. However, when we update array  $a$  at index  $j$  to 3 to form array  $b$ , it *does not affect the original array  $a$* . Therefore, the element at index  $i$  of array  $a$  remains 2, regardless of the update made to form array  $b$ . Thus, the formula holds true under all possible interpretations, making it valid. Therefore, the correct statement is:

C. The formula is a sentence and is valid.

## 5 Question - Structures

Consider the universe

$$U \triangleq \{a, b, c\}$$

where we have a single unary function  $f$ , a predicate  $p$ , and a single constant  $c$ . Assume we have a variable assignment

$$\sigma(x) = b$$

and the following interpretation  $I$ :

$$I(a) \triangleq a$$

$$I(f) \triangleq \{c \rightarrow b, b \rightarrow a, a \rightarrow c\}$$

$$I(p) \triangleq \{(c, a), (b, b), (a, a), (b, a)\}$$

Now consider the formulas

$$F \triangleq p(x, a) \rightarrow p(f(x), a)$$

$$G \triangleq \exists x. (p(x, x) \rightarrow p(f(x), x))$$

We define a structure  $S \triangleq (U, I)$  which of the following statements are true? You may choose multiple answers.

- A.  $S, \sigma \models F$ , that is  $F$  is true
- B.  $S, \sigma \not\models F$ , that is  $F$  is false
- C.  $S, \sigma \models G$ , that is  $G$  is true
- D.  $S, \sigma \not\models G$ , that is  $G$  is false

### 5.1 Theory

**Definition 5.1.** Structure

A structure is just a pair  $S = (\mathcal{U}, I)$  where  $\mathcal{U}$  is the universe (or domain) of discourse, and  $I$  is an interpretation function that assigns meanings to the symbols in the signature.

So given a structure  $S = (\mathcal{U}, I)$  and a variable assignment  $\sigma$ , we can evaluate the truth of formulas within that structure. We denote this as  $S, \sigma \models F$  if the formula  $F$  is true in the structure  $S$  under the variable assignment  $\sigma$ , and  $S, \sigma \not\models F$  if it is false.

## 5.2 Solution

To determine the truth of the statements regarding the formulas  $F$  and  $G$  in the structure  $S$  with the variable assignment  $\sigma$ , we need to evaluate both formulas under the given interpretation and variable assignment.

**Evaluating Formula F:** The formula  $F$  is given by

$$F \triangleq p(x, a) \rightarrow p(f(x), a)$$

We need to evaluate  $F$  under the structure  $S$  with the variable assignment  $\sigma(x) = b$ .

- First, we compute  $p(x, a)$  under the assignment  $\sigma$ :

$$p(b, a)$$

From the interpretation  $I(p)$ , we see that  $(b, a) \in I(p)$ , so  $p(b, a)$  is true.

- Next, we compute  $f(x)$  under the assignment  $\sigma$ :

$$f(b) = a \quad (\text{from } I(f))$$

- Now we compute  $p(f(x), a)$ :

$$p(a, a)$$

From the interpretation  $I(p)$ , we see that  $(a, a) \in I(p)$ , so  $p(a, a)$  is true.

Since both  $p(x, a)$  and  $p(f(x), a)$  are true, the implication  $p(x, a) \rightarrow p(f(x), a)$  is true. Therefore, we conclude that  $S, \sigma \models F$ .

**Evaluating Formula G:**

The formula  $G$  is given by

$$G \triangleq \exists x. (p(x, x) \rightarrow p(f(x), x))$$

We need to evaluate  $G$  under the structure  $S$ . To satisfy the existential quantifier, we need to find at least one  $x \in \mathcal{U}$  such that

$$p(x, x) \rightarrow p(f(x), x)$$

We will check each element in the universe  $\mathcal{U} = \{a, b, c\}$ :

- For  $x = a$ :

$$p(a, a) \rightarrow p(f(a), a)$$

From  $I(p)$ ,  $p(a, a)$  is true. Now,  $f(a) = c$ , so we need to check  $p(c, a)$ . From  $I(p)$ ,  $(c, a) \in I(p)$ , so  $p(c, a)$  is true. Thus, the implication is true.

- For  $x = b$ :

$$p(b, b) \rightarrow p(f(b), b)$$

From  $I(p)$ ,  $p(b, b)$  is true. Now,  $f(b) = a$ , so we need to check  $p(a, b)$ . From  $I(p)$ ,  $(a, b) \notin I(p)$ , so  $p(a, b)$  is false. Thus, the implication is false.

- For  $x = c$ :

$$p(c, c) \rightarrow p(f(c), c)$$

From  $I(p)$ ,  $p(c, c)$  is false. Thus, the implication is true.

Since there exists at least one  $x \in \mathcal{U}$  (specifically  $x = a$  or  $x = c$ ) that satisfies the formula  $G$ , we conclude that  $S, \sigma \models G$ .

**Final Conclusion:**

Based on the analysis, we find that:

- $S, \sigma \models F$  (Statement A is true)
- $S, \sigma \models G$  (Statement C is true)

## 6 Question - Language Semantics

We consider the standard Nano language, as discussed in class, *without* the addition of the statements `assume`, `assert`, and `havoc`.

Your friend wants to add the following statement to the language

$$s_1 \circ s_2$$

They suggest defining its semantics via the following two rules:

$$\frac{(s_1, \sigma) \Downarrow \sigma'}{(s_1 \circ s_2, \sigma) \Downarrow \sigma'} \circ_1 \qquad \frac{(s_2, \sigma) \Downarrow \sigma'}{(s_1 \circ s_2, \sigma) \Downarrow \sigma'} \circ_2$$

Which statements about the  $\Downarrow$  relation are true? You may choose multiple answers.

- A. For a statement  $s$  and an initial state  $\sigma$  in our extended language, there always exists a state  $\sigma'$  such that  $(s, \sigma) \Downarrow \sigma'$  contains no while loops.
- B. For a statement  $s$  and an initial state  $\sigma$  in our extended language, there might not exist a state  $\sigma'$  such that  $(s, \sigma) \Downarrow \sigma'$  contains no while loops.
- C. For a statement  $s$  and an initial state  $\sigma$  in our extended language, if  $(s, \sigma) \Downarrow \sigma'$  and  $(s, \sigma) \Downarrow \sigma''$  then  $\sigma' = \sigma''$ .
- D. For a statement  $s$  and an initial state  $\sigma$  in our extended language, if  $(s, \sigma) \Downarrow \sigma'$  and  $(s, \sigma) \Downarrow \sigma''$  then it might be the case that  $\sigma' \neq \sigma''$ .

### 6.1 Theory

There are a few things to deconstruct here, first it makes sense to understand what the relationship here  $\Downarrow$  is.

**Definition 6.1.** Big-step operational semantics

The big-step operational semantics, also known as natural semantics, is a way to define the meaning of programming languages by describing how statements and expressions evaluate to final results in a single step. The relation  $(s, \sigma) \Downarrow \sigma'$  indicates that starting from an initial state  $\sigma$ , executing the statement  $s$  results in a final state  $\sigma'$ .

There are two key properties of the big-step operational semantics that are relevant to this question:

- **Determinism:** For a given statement  $s$  and initial state  $\sigma$ , there is at most one final state  $\sigma'$  such that  $(s, \sigma) \Downarrow \sigma'$ . This means that the execution of a statement is deterministic, leading to a unique outcome.
- **Termination:** Not all statements are guaranteed to terminate. Some statements, such as infinite loops, may not lead to a final state  $\sigma'$ . In such cases, there may not exist any  $\sigma'$  such that  $(s, \sigma) \Downarrow \sigma'$ .

### 6.2 Solution

To determine the truth of the statements regarding the extended Nano language with the new statement  $s_1 \circ s_2$  and its semantics, we need to analyze the implications of the provided rules.

**Analyzing Statements A and B:**

The rules for the new statement  $s_1 \circ s_2$  allow for non-deterministic execution, as either  $s_1$  or  $s_2$  can be executed to produce the final state  $\sigma'$ . This non-determinism means that there may be cases where neither  $s_1$  nor  $s_2$  leads to termination, especially if one or both of them contain while loops that do not terminate. Therefore, it is possible that for a statement  $s$  and an initial state  $\sigma$ , there might not exist a state  $\sigma'$  such that  $(s, \sigma) \Downarrow \sigma'$ . Thus, statement B is true, and statement A is false.

**Analyzing Statements C and D:**

Due to the non-deterministic nature of the new statement  $s_1 \circ s_2$ , it is possible for the execution to lead to different final states  $\sigma'$  and  $\sigma''$  depending on whether  $s_1$  or  $s_2$  is executed. Therefore, if  $(s, \sigma) \Downarrow \sigma'$  and  $(s, \sigma) \Downarrow \sigma''$ , it might be the case that  $\sigma' \neq \sigma''$ . Thus, statement D is true, and statement C is false.

## 7 Question - Hoare Logic

Let precondition  $P \triangleq x \geq 3$ , and postcondition  $Q \triangleq y \geq 1$ . Consider the following two Nano programs:

$$S \triangleq \text{if } (x \geq 0) \text{ then } y := 1 \text{ else } y := 0$$

$$T \triangleq y := 0; \text{ while } (x \geq 0) \text{ do } x := x + 1$$

Which of the following statements are true, you may choose multiple answers.

- A. The Hoare triple  $\{P\} S \{Q\}$  is valid, that is,  $\models \{P\} S \{Q\}$
- B. The Hoare triple  $\{P\} S \{Q\}$  is not valid, that is,  $\not\models \{P\} S \{Q\}$
- C. The Hoare triple  $\{P\} T \{Q\}$  is valid, that is,  $\models \{P\} T \{Q\}$
- D. The Hoare triple  $\{P\} T \{Q\}$  is not valid, that is,  $\not\models \{P\} T \{Q\}$

### 7.1 Theory

#### Definition 7.1. Hoare Triple

A *Hoare Triple* is a basic unit of reasoning or judgement in Hoare logic. It is typically written in the form

$$\{P\} S \{Q\}$$

where  $P$  is the precondition,  $S$  is a program statement, and  $Q$  is the postcondition. The Hoare triple expresses that if the precondition  $P$  holds before executing the statement  $S$ , then the postcondition  $Q$  will hold after executing  $S$ , provided that  $S$  terminates.

#### Definition 7.2. Hoare Rules

Hoare logic provides a set of inference or derivation rules which can be used to deduce or prove the validity of Hoare triples. Some of the key Hoare rules include:

$$\frac{\{P \wedge B\} S_1 \{Q\} \quad \{P \wedge \neg B\} S_2 \{Q\}}{\{P\} \text{ if } B \text{ then } S_1 \text{ else } S_2 \{Q\}} \text{IF} \qquad \frac{\{I \wedge B\} s \{I\} \quad P \rightarrow I \quad I \wedge \neg B \rightarrow Q}{\{P\} \text{ while } B \text{ do } s \{Q\}} \text{WHILE}$$

$$\frac{\{P\} S_1 \{R\} \quad \{R\} S_2 \{Q\}}{\{P\} S_1; S_2 \{Q\}} \text{SEQ} \qquad \frac{}{\{Q[x \mapsto E]\} x := E \{Q\}} \text{ASSIGN}$$

#### Definition 7.3. Partial Validity

A Hoare triple  $\{P\} S \{Q\}$  is said to be partially valid, denoted as  $\models \{P\} S \{Q\}$ , if whenever the precondition  $P$  holds before executing the statement  $S$ , and if  $S$  terminates, then the postcondition  $Q$  holds after executing  $S$ . Formally we can express this as

$$\sigma \models P \wedge \exists \sigma'. (S, \sigma) \Downarrow \sigma' \implies \sigma' \models Q \quad (\text{Partial Validity})$$

### 7.2 Solution

To determine the truth of the statements regarding the Hoare triples  $\{P\} S \{Q\}$  and  $\{P\} T \{Q\}$ , we need to analyze both programs  $S$  and  $T$  with respect to the given precondition  $P$  and postcondition  $Q$ .

#### Analyzing Hoare Triple $\{P\} S \{Q\}$ :

The program  $S$  is given by

$$S \triangleq \text{if } (x \geq 0) \text{ then } y := 1 \text{ else } y := 0$$

The precondition  $P$  is  $x \geq 3$ , and the postcondition  $Q$  is  $y \geq 1$ .

- If  $x \geq 0$  (which is true since  $x \geq 3$ ), then  $y := 1$ , which satisfies the postcondition  $Q$  since  $1 \geq 1$ .
- If  $x < 0$  (which cannot happen since  $x \geq 3$ ), then  $y := 0$ , which does not satisfy the postcondition  $Q$ .

Since the precondition  $P$  guarantees that  $x \geq 0$ , the program  $S$  will always execute the then-branch, resulting in  $y := 1$ . Therefore, the Hoare triple  $\{P\} S \{Q\}$  is valid. Thus, statement A is true, and statement B is false.

### Analyzing Hoare Triple $\{P\} T \{Q\}$ :

The program  $T$  is given by

$$T \triangleq y := 0; \text{ while } (x \geq 0) \text{ do } x := x + 1$$

The precondition  $P$  is  $x \geq 3$ , and the postcondition  $Q$  is  $y \geq 1$ .

- The first statement  $y := 0$  sets  $y$  to 0, which does not satisfy the postcondition  $Q$ .
- The while loop  $\text{while } (x \geq 0) \text{ do } x := x + 1$  will continue to increment  $x$  indefinitely since  $x$  starts at a value greater than or equal to 3. This means that the program  $T$  does not terminate.

Since the program  $T$  does not terminate, the Hoare triple  $\{P\} T \{Q\}$  is not valid. Thus, statement D is true, and statement C is false.

**Final Conclusion:** Based on the analysis, we find that:

- The Hoare triple  $\{P\} S \{Q\}$  is valid (Statement A is true).
- The Hoare triple  $\{P\} T \{Q\}$  is not valid (Statement D is true).

## 8 Question - Hoare Rules

Your friend suggests the following proof rule for assignment in Hoare logic.

$$\{(x = e) \rightarrow Q\} x := e \{Q\} \text{ ASSIGN}$$

is the proof rule correct? If yes, prove its soundness. If no, provide a counterexample showing that it is unsound.

### 8.1 Theory

The central thing to understand here is the notion of soundness within Hoare logic.

#### Definition 8.1. Soundness

As a reminder hoare triples can be either valid or derivable. A hoare triple  $\{P\} s \{Q\}$  is said to be *valid* if it holds true in the semantics of the programming language, i.e. if executing statement  $s$  from any state satisfying precondition  $P$  will always terminate in a state satisfying postcondition  $Q$ . A hoare triple is said to be *derivable* if it can be proven using the rules of hoare logic.

- **SOUNDNESS** - If a hoare triple  $\{P\} s \{Q\}$  is derivable using the hoare rules, then it is valid. In other words if we can prove a hoare triple using the rules of hoare logic, then it must be true in the semantics of the programming language. Formally

$$\vdash \{P\} s \{Q\} \rightarrow \models \{P\} s \{Q\}$$

- **COMPLETENESS** - If a hoare triple  $\{P\} s \{Q\}$  is valid, then it is derivable using the hoare rules. In other words if a hoare triple is true in the semantics of the programming language, then we can prove it using the rules of hoare logic. Formally

$$\models \{P\} s \{Q\} \rightarrow \vdash \{P\} s \{Q\}$$

#### Definition 8.2. Proof of soundness

A proof of soundness aims to show that if a hoare triple  $\{P\} s \{Q\}$  is derivable using the hoare rules, then it preserves or exhibits the property of partial validity.

Conceptually the idea here is that if a rule is syntactically correct (i.e. we can derive a hoare triple using the rule), but *not* semantically correct (i.e. the hoare triple is not valid), then the rule is unsound. Thus to prove soundness we need to show that any hoare triple derived using the rule is also valid.

Soundness represents a meta-theoretical property in that we are reasoning about the relationship between the syntax (the hoare rules) and the semantics (the meaning of the programming language). Whereas partial validity is a property of individual hoare triples within the system.

**Definition 8.3.** Assignment Rule

The standard assignment rule in Hoare logic is given by:

$$\frac{}{\{Q[x \mapsto E]\} x := E \{Q\}} \text{ ASSIGN}$$

This rule states that to prove a Hoare triple for an assignment statement  $x := E$ , we need to ensure that the postcondition  $Q$  holds after substituting  $E$  for  $x$  in  $Q$ . The proof of soundness in case amounts to proving partial validity for all states so

$$\forall \sigma. \sigma \models Q[x \mapsto E] \wedge \exists \sigma'. (x := E, \sigma) \Downarrow \sigma' \implies \sigma' \models Q$$

We know that under the semantics of assignment the state after execution should be the state before execution but with  $x$  now mapping to the value of  $E$  in the original state. Formally

$$(x := E, \sigma) \Downarrow \sigma' \iff \sigma' = \sigma[x \mapsto \sigma(E)]$$

Our proof of soundness is then

$$\sigma \models Q[x \mapsto E] \implies \sigma' \models Q$$

The key lemma used here is

$$\sigma \models Q[x \mapsto E] \iff \sigma[x \mapsto \sigma(E)] \models Q \quad (\text{Substitution Lemma})$$

this lemma is typically proved by structural induction on  $Q$ . In our original proof of soundness we can then conclude that because  $\sigma' = \sigma[x \mapsto \sigma(E)]$  we have that  $\sigma' \models Q$  as required.

## 8.2 Solution

To determine whether the proposed proof rule for assignment in Hoare logic is correct, we need to analyze its soundness. The proposed rule is:

$$\{(x = e) \rightarrow Q\} x := e \{Q\} \text{ ASSIGN}$$

To prove the soundness of this rule, we need to show that if the precondition  $(x = e) \rightarrow Q$  holds before executing the assignment  $x := e$ , then the postcondition  $Q$  will hold after executing the assignment, provided that the assignment terminates.

**Proof of Soundness:**

Let  $\sigma$  be the initial state before executing the assignment  $x := e$ . We need to show that if  $\sigma \models (x = e) \rightarrow Q$ , then after executing  $x := e$ , the resulting state  $\sigma'$  satisfies  $\sigma' \models Q$ .

- Before the assignment, we have  $\sigma \models (x = e) \rightarrow Q$ . This means that if  $\sigma(x) = \sigma(e)$ , then  $\sigma \models Q$ .
- After executing the assignment  $x := e$ , the new state  $\sigma'$  is defined as:

$$\sigma'(x) = \sigma(e)$$

and for all other variables  $y \neq x$ ,  $\sigma'(y) = \sigma(y)$ .

- Now, we need to check whether  $\sigma' \models Q$ . Since  $\sigma'(x) = \sigma(e)$ , we have two cases to consider:
  - If  $\sigma(e) = \sigma(x)$ , then by the precondition  $(x = e) \rightarrow Q$ , we have  $\sigma \models Q$ . Since  $\sigma'$  only changes the value of  $x$  to  $\sigma(e)$ , and  $Q$  is satisfied in the original state  $\sigma$ , it follows that  $\sigma' \models Q$ .
  - If  $\sigma(e) \neq \sigma(x)$ , then the precondition  $(x = e) \rightarrow Q$  does not impose any requirement on  $Q$ , and we cannot conclude anything about  $Q$  in this case. However, since the assignment  $x := e$  has been executed, we need to ensure that  $Q$  holds in the new state  $\sigma'$ . In this case, we cannot guarantee that  $\sigma' \models Q$ .

From the above analysis, we see that the proposed proof rule does not guarantee that  $Q$  holds after the assignment  $x := e$  in all cases. Specifically, if  $\sigma(e) \neq \sigma(x)$ , we cannot conclude that  $\sigma' \models Q$ . Therefore, the proposed proof rule is not sound.

As a counterexample take  $Q \equiv (x = 0)$  and  $e = 1$ . Then our hoare triple becomes

$$\{(x = 1) \rightarrow (x = 0)\} x := 1 \{x = 0\}$$

If  $x \neq 1$  before the assignment then the precondition is vacuously true, however after the assignment  $x$  will equal 1 and thus the postcondition will be false. Thus we have a case where the precondition holds but the postcondition does not after execution, showing that the rule is unsound.

## 9 Question - Verification Condition Generation

We have written a program  $s$  and annotated it with *loop invariants*. We want to ensure that the postcondition  $Q$  holds. For this, we use our VCGen procedure to generate a formula  $vc(s, Q)$ . Say we send this formula to an SMT solver, and it lets us know that the formula is not valid. What can we conclude? Explain your reasoning.

### 9.1 Theory

**Definition 9.1.** Verification Conditions (VC)

Verification Conditions (VC) are logical formulas generated from a program and its specifications (preconditions, postconditions, and loop invariants) that need to be proven true to ensure the correctness of the program with respect to its specifications. The VCGen procedure takes a program  $s$  and a postcondition  $Q$  and produces a formula  $vc(s, Q)$  that encapsulates the necessary conditions for the program to satisfy the postcondition  $Q$ .

**Definition 9.2.** Satisfiability module theories (SMT)

Satisfiability Modulo Theories (SMT) is a decision problem for logical formulas with respect to combinations of background theories such as arithmetic, bit-vectors, arrays, and others. An SMT solver is a tool that determines the satisfiability of these formulas. If a formula is satisfiable, there exists an interpretation that makes the formula true; if it is unsatisfiable, no such interpretation exists.

Importantly for our context, if an SMT solver determines that a formula is *not valid*, it means that there exists at least one interpretation under which the formula is false, but it does *not* imply that the program is incorrect.

### 9.2 Solution

If the SMT solver indicates that the formula  $vc(s, Q)$  generated by the VCGen procedure is not valid, we can conclude that there exists at least one interpretation under which the formula is false. However, this does not necessarily imply that the program  $s$  is incorrect or that the postcondition  $Q$  does not hold after executing  $s$ . This is because the VCGen procedure generates conditions that are sufficient but not always necessary for the correctness of the program. In short, we can conclude that our current invariants are insufficient to prove the correctness of the program with respect to the postcondition  $Q$  but not that the program is incorrect.

## 10 Question - Horn SAT/UNSAT

Consider the following set of Horn Clauses

$$x = 2 \rightarrow r(x) \quad (1)$$

$$q(x) \rightarrow x = 1 \quad (2)$$

$$r(x) \rightarrow x = 2 \quad (3)$$

$$q(x) \wedge r(x) \rightarrow l(x) \quad (4)$$

$$l(x) \rightarrow q(x) \quad (5)$$

Which of the following statements are true?

- A. The clauses are recursive.
- B. The clauses are not recursive.
- C. The clauses are satisfiable.
- D. The clauses are not satisfiable.

### 10.1 Theory

**Definition 10.1.** Horn Clause

A Horn clause is a special type of logical formula that has at most one positive literal. In propositional logic, a Horn

clause can be expressed in the form

$$\underbrace{p(x_1, x_2) \wedge q(x_1, x_2, x_3) \wedge \dots \wedge \phi}_{\text{body}} \rightarrow \underbrace{H}_{\text{head}} \quad (\text{Horn Clause})$$

where

- **HEAD** - The head  $H$  is either a single positive literal (query) or the constant false (denoting a contradiction).
- **BODY** - The body consists of a conjunction of zero or more positive literals (queries) and an optional constraint  $\phi$  which is a formula in some underlying theory  $T$ .

**Definition 10.2.** Set of Horn Clauses

We define a set  $\mathbb{C}$  of Horn clauses

$$\mathbb{C} \triangleq \{C_1, C_2, \dots, C_m\} \quad (\text{Horn Clauses})$$

with two important properties:

- **DEPENDENCY** - We say a query  $p$  *depends* on a query  $q$  if there exists a clause in  $\mathbb{C}$  where  $p$  appears in the head and  $q$  appears in the body.
- **RECURSION** - A set of Horn clauses  $\mathbb{C}$  is said to be *recursive* if at least one predicate directly or indirectly depends on itself. Formally, there exists a sequence of clauses  $C_1, \dots, C_k$  such that the head of  $C_i$  depends on the body of  $C_{i+1}$  for  $1 \leq i < k$ , and the head of  $C_k$  depends on the body of  $C_1$ . If no such sequence exists, the set is said to be *non-recursive*. So a simple example is something like

$$q(x) \wedge x' = x + 1 \rightarrow q(x')$$

here  $q$  depends on itself directly.

**Definition 10.3.** Solution  $\Sigma$

We define a solution  $\Sigma$  as a function which maps queries to a formula in the theory  $T$  over the same variables. Denoted as an inference rule

$$\frac{\Sigma(p) \wedge \Sigma(q) \wedge \dots \wedge \phi \rightarrow \Sigma(r)}{\Sigma \models p(x_1, x_2) \wedge q(x_1, x_2, x_3) \wedge \dots \wedge \phi \rightarrow r(x_1)} \text{ SOLUTION}$$

We say that a set of Horn clauses  $\mathbb{C}$  is *satisfiable* if there exists a solution  $\Sigma$  such that all clauses in  $\mathbb{C}$  are satisfied under  $\Sigma$ . If no such solution exists, the set is said to be *unsatisfiable*. Formally

$$\mathbb{C} \text{ is satisfiable} \iff \exists \Sigma. \forall C \in \mathbb{C}. \Sigma \models C$$

## 10.2 Solution

To determine the truth of the statements regarding the set of Horn clauses provided, we need to analyze both the recursion and satisfiability of the clauses.

**Analyzing Recursion:** We examine the dependencies among the predicates in the clauses:

- From clause (1),  $r(x)$  depends on the constraint  $x = 2$ .
- From clause (2),  $q(x)$  depends on the constraint  $x = 1$ .
- From clause (3),  $x = 2$  depends on  $r(x)$ .
- From clause (4),  $l(x)$  depends on both  $q(x)$  and  $r(x)$ .
- From clause (5),  $q(x)$  depends on  $l(x)$ .

We can see that  $q(x)$  depends on  $l(x)$  (clause 5), and  $l(x)$  depends on  $q(x)$  (clause 4), forming a cycle. Therefore, the set of Horn clauses is recursive. Thus, statement A is true, and statement B is false.

**Analyzing Satisfiability:** We attempt to find a solution  $\Sigma$  that satisfies all clauses:

- From clause (1), if  $x = 2$ , then  $r(2)$  must be true.
- From clause (2), if  $q(x)$  is true, then  $x = 1$ .
- From clause (3), if  $r(x)$  is true, then  $x = 2$ .
- From clause (4), if both  $q(x)$  and  $r(x)$  are true, then  $l(x)$  must be true.
- From clause (5), if  $l(x)$  is true, then  $q(x)$  must be true.

Now, let's analyze the implications:

- From clauses (1) and (3), we see that  $r(x)$  can only be true if  $x = 2$ .
- From clause (2),  $q(x)$  can only be true if  $x = 1$ .
- From clause (4), for  $l(x)$  to be true, both  $q(x)$  and  $r(x)$  must be true. However, this is impossible since  $q(x)$  requires  $x = 1$  and  $r(x)$  requires  $x = 2$ .

Since there is no assignment of  $x$  that can satisfy all clauses simultaneously, the set of Horn clauses is not satisfiable. Thus, statement D is true, and statement C is false.

### Final Conclusion:

Based on the analysis, we find that:

- The clauses are recursive (Statement A is true).
- The clauses are not satisfiable (Statement D is true).

## 11 Question - Proving Correctness

We use our weakest precondition method to calculate the set of horn clauses  $\mathbb{C}$  for a program  $s$ . We now use our abstract Horn-solving algorithm with a fixed set of predicates  $P$ . We want to check if the program  $s$  is correct, that is, whether all assertions in  $s$  always hold.

After running our solving algorithm, we receive a solution  $\Sigma$ . We find that  $\Sigma$  doesn't satisfy the clauses, i.e.,  $\Sigma \not\models \mathbb{C}$ . Which of the following statements is true?

- A. We know that the program is wrong, that is, that there is an assertion in the program that doesn't hold for some execution.
- B. We know that the program is correct, that is, all assertions always hold.
- C. We don't know if the program is correct. It could be that we can find a solution that satisfies  $\mathbb{C}$  by adding more predicates to  $P$ .
- D. We don't know if the program is correct. It could be that we can find a solution that satisfies  $\mathbb{C}$  by removing some predicates from  $P$ .

### Definition 11.1. Predicate abstraction soundness

The most important property of predicate abstraction is that it is *sound* but *not complete*. Which is to say that if some abstract reachable state violates an assertion (or satisfies some error condition) then because of *false positives* introduced by the abstraction, we *cannot* conclude that the concrete program is incorrect. The only thing we can prove is *safety* - if no abstract reachable state violates an assertion, then the concrete program is correct.

### Definition 11.2. Over-approximation

Related to the property of soundness is the notion of *over-approximation*. Predicate abstraction creates an over-approximation of the set of reachable states of the program. This means that the abstract model may include states that are not reachable in the concrete program. In other words, the abstraction amounts to a *weakening* or super-set of the actual program behavior. This is why we can have false positives - the abstract model may show a violation of an assertion that cannot actually occur in the concrete program.

Therefore, if our abstract Horn-solving algorithm produces a solution  $\Sigma$  that does not satisfy the clauses  $\mathbb{C}$ , we cannot conclude that the program is incorrect. It may be the case that the abstraction introduced by the chosen predicates  $P$  is too coarse, leading to false positives. By refining the set of predicates (either by adding more predicates to capture more precise properties), we may be able to find a solution that satisfies  $\mathbb{C}$  and thus prove the correctness of the program.

## 11.1 Solution

Based on the properties of predicate abstraction and the implications of the abstract Horn-solving algorithm, we can conclude that statement C is true. We do not know if the program is correct, as it could be that we can find a solution that satisfies  $\mathbb{C}$  by adding more predicates to  $P$ . The failure of the solution  $\Sigma$  to satisfy the clauses  $\mathbb{C}$  does not necessarily imply that the program is incorrect, due to the possibility of false positives introduced by the abstraction. Therefore, we cannot conclude that the program is wrong (statement A is false) or that it is correct (statement B is false). Additionally, removing predicates from  $P$  would not help in finding a satisfying solution, as it would likely lead to a coarser abstraction, which is not beneficial in this context (statement D is false).

## 12 Question - Solving Horn Clauses

Consider the following set of Horn clauses

$$x = 1 \wedge y = 1 \rightarrow q(x, y) \quad (1)$$

$$q(x, y) \wedge y' = y + 1 \rightarrow q(x, y') \quad (2)$$

We are given the following predicate set

$$P \triangleq \{x = y, x = 1, y = 2\}$$

Use the abstract Horn-solving algorithm to solve the Horn clauses. Upon reaching a fixed-point, which solution for  $q$  does the algorithm compute?

### 12.1 Theory

**Definition 12.1.** Strongest Postcondition

We define an operator  $\text{post}$  which computes the strongest postcondition of a formula  $\varphi$  w.r.t a set of variables  $\bar{x}$ . If we define  $\bar{y}$  as the set of variables in  $\varphi$  that are not in  $\bar{x}$ , i.e.  $\bar{y} = \text{FV}(\varphi) \setminus \bar{x}$  then we can define the strongest postcondition as

$$\text{post}(\varphi, \bar{x}) \triangleq \exists \bar{y}. \varphi \quad (\text{Strongest Postcondition})$$

where the following property holds for all variables  $x$  and formulas  $\varphi$

$$\text{post}(\varphi_1 \wedge \varphi_2, \dots, \bar{x}) \equiv \text{post}(\varphi_1, \bar{x}) \wedge \text{post}(\varphi_2, \bar{x}) \wedge \dots \quad (\text{SP Conjunction})$$

**Definition 12.2.** Abstraction function  $\alpha$

We define the abstraction function  $\alpha$  as a function which maps concrete program states to abstract states based on a set of predicates  $P$ . Formally

$$\alpha(\varphi, P) \triangleq \bigvee_{p \in P} \varphi \rightarrow p$$

In practical terms this means that for a given formula  $\varphi$ , we check which predicates in  $P$  are implied by  $\varphi$  and construct a new formula that is a conjunction of those predicates.

**Definition 12.3.** Abstract strongest postcondition

We define the abstract strongest postcondition operator  $\text{post}_{\text{abs}}$  using the abstraction function  $\alpha$  and the strongest postcondition operator  $\text{post}$  over a set of predicates  $P$ . Formally

$$\text{post}_{\text{abs}}(\varphi, \bar{x}, P) \triangleq \alpha(\text{post}(\varphi, \bar{x}), P) \quad (\text{Abstract SP})$$

We can describe the general approach to solving horn clauses with predicate abstraction as follows:

1. *Initialize* - We start with a solution  $\Sigma$  that maps each query  $p$  to  $\perp$ , formally

$$\Sigma(p) = \perp \quad \forall p$$

2. *Iterate* - We pick any clause whos head is a query, i.e. a clause in the form

$$p(\bar{x}) \wedge \phi \rightarrow q(\bar{y})$$

then compute

$$\psi \triangleq \text{post}_{\text{abs}}(\Sigma(p(\bar{x})) \wedge \phi, \bar{y}, P) \equiv \alpha(\text{post}(\Sigma(p(\bar{x})) \wedge \phi, \bar{y}), P)$$

and if  $\Sigma(q) \equiv \psi'$ , then update

$$\Sigma(q) \triangleq \psi' \vee \psi$$

## 12.2 Solution

We start with the initial solution

$$\Sigma(q) = \perp$$

We have two clauses to consider. We will process them in order until we reach a fixed point.

1. *Initialize* - Start with

$$\Sigma(q) = \perp$$

2. *Iterate* (Clause 1) - We compute

$$\psi_1 \triangleq \text{post}_{\text{abs}}(\Sigma(\perp) \wedge (x = 1 \wedge y = 1), (x, y), P) \equiv \alpha(\text{post}((x = 1 \wedge y = 1), (x, y)), P)$$

Here we have  $\bar{x} = \{x, n\}$  and  $\bar{y} = \emptyset$  so

$$\text{post}((x = 1 \wedge y = 1), (x, y)) \equiv (x = 1 \wedge y = 1)$$

Applying the abstraction function  $\alpha$  we get

$$\begin{aligned} \alpha((x = 1 \wedge y = 1), P) &\equiv \bigwedge_{p \in P} ((x = 1 \wedge y = 1) \rightarrow p) \\ &\equiv (x = 1 \wedge y = 1) \rightarrow (x = y) \\ &\quad \wedge (x = 1 \wedge y = 1) \rightarrow (x = 1) \\ &\quad \wedge (x = 1 \wedge y = 1) \rightarrow (y = 2) \\ &\equiv (x = y) \wedge (x = 1) \end{aligned}$$

Thus we update

$$\Sigma(q) \equiv \perp \vee ((x = y) \wedge (x = 1)) \equiv (x = y) \wedge (x = 1)$$

3. *Iterate* (Clause 2) - We compute

$$\psi_2 \triangleq \text{post}_{\text{abs}}(\Sigma(q(x, y)) \wedge (y' = y + 1), (x, y'), P) \equiv \alpha(\text{post}(((x = y) \wedge (x = 1)) \wedge (y' = y + 1), (x, y')), P)$$

Here we have  $\bar{x} = \{x, y'\}$  and  $\bar{y} = \{y\}$  so

$$\begin{aligned} \text{post}(((x = y) \wedge (x = 1)) \wedge (y' = y + 1), (x, y')) &\equiv \exists y. ((x = y) \wedge (x = 1) \wedge (y' = y + 1)) \\ &\equiv (x = 1) \wedge \exists y. (x = y \wedge y' = y + 1) \\ &\equiv (x = 1) \wedge (y' = x + 1) \\ &\equiv (x = 1) \wedge (y' = 2) \end{aligned}$$

Applying the abstraction function  $\alpha$  we get

$$\begin{aligned} \alpha((x = 1) \wedge (y' = 2), P) &\equiv \bigwedge_{p \in P} ((x = 1) \wedge (y' = 2) \rightarrow p) \\ &\equiv (x = 1) \wedge (y' = 2) \rightarrow (x = y') \\ &\quad \wedge (x = 1) \wedge (y' = 2) \rightarrow (x = 1) \\ &\quad \wedge (x = 1) \wedge (y' = 2) \rightarrow (y' = 2) \\ &\equiv (x = 1) \wedge (y' = 2) \end{aligned}$$

Thus we update

$$\Sigma(q) \equiv ((x = y) \wedge (x = 1)) \vee ((x = 1) \wedge (y' = 2))$$

4. *Fixed Point* - Repeating the process will not yield any new information, as the abstraction will continue to yield the same predicates. Thus we have reached a fixed point. The final solution is

$$\Sigma(q) \equiv ((x = y) \wedge (x = 1)) \vee ((x = 1) \wedge (y = 2))$$

## 13 Question - IFC Types

We are given an environment  $\Gamma$ , such that

$$\Gamma(x) = \text{obs} \quad \text{and} \quad \Gamma(y) = \text{sec}$$

We are also given the following program  $s$

$$\text{if } (y \geq 0) \text{ then } x := 2$$

Does program  $s$  type check under the IFC type system with the initial pc label  $\text{obs}$ ? That is, does  $\Gamma, \text{obs} \vdash s$  hold? Explain your reasoning.

### 13.1 Theory

**Definition 13.1.** Information Flow Control (IFC) Type System

An Information Flow Control (IFC) type system is a static analysis framework designed to enforce security policies by controlling how information flows through a program. The primary goal of an IFC type system is to prevent unauthorized information leaks from high-security levels (e.g., confidential data) to low-security levels (e.g., public data).

In an IFC type system, each variable and expression is assigned a security label that indicates its sensitivity level. Common labels include:

- **CONFIDENTIAL (sec)** - Indicates that the data is sensitive and should not be exposed to lower security levels.
- **OBSERVABLE (obs)** - Indicates that the data can be safely observed or accessed by lower security levels.

The type system enforces rules that govern how information can flow between variables and expressions based on their security labels. For example, a variable labeled as **sec** should not influence the value of a variable labeled as **obs**, as this would constitute an unauthorized information flow.

The typing judgment in an IFC type system is typically denoted as

$$\Gamma, \text{pc} \vdash s$$

where

- *Context* ( $\Gamma$ ) - A mapping from variables to their security labels.

$$\Gamma : \text{Var} \rightarrow \{\text{sec}, \text{obs}\}$$

- *Program Counter Label* ( $\text{pc}$ ) - The current security level of the program counter, which reflects the sensitivity of the control flow context.
- *Statement* ( $s$ ) - The program statement being analyzed.

**Definition 13.2.** IFC Typing rules

The IFC type system includes several typing rules that dictate how statements can be typed based on the security labels of the variables involved. Some common rules include:

$$\begin{array}{c}
\text{IFC-PC} \quad \frac{}{\Gamma \vdash \text{pc} : s} \quad \text{IFC-SKIP} \quad \frac{}{\Gamma, \text{pc} \vdash \text{skip}} \quad \text{IFC-ASSIGN} \quad \frac{\Gamma \vdash x : \tau_1 \quad \Gamma \vdash e : \tau_2 \quad \tau_2 \sqsubseteq \tau_1 \quad \text{pc} \sqsubseteq \tau_1}{\Gamma, \text{pc} \vdash x := e} \quad \text{IFC-SEQ} \quad \frac{\Gamma, \text{pc} \vdash s_1 \quad \Gamma, \text{pc} \vdash s_2}{\Gamma, \text{pc} \vdash s_1; s_2} \\
\\
\text{IFC-IF} \quad \frac{\Gamma \vdash b : \tau \quad \Gamma, \text{pc} \sqcup \tau \vdash s_1 \quad \Gamma, \text{pc} \sqcup \tau \vdash s_2}{\Gamma, \text{pc} \vdash \text{if } b \text{ then } s_1 \text{ else } s_2} \quad \text{IFC-WHILE} \quad \frac{\Gamma \vdash b : \tau \quad \Gamma, \text{pc} \sqcup \tau \vdash s}{\Gamma, \text{pc} \vdash \text{while } b \text{ do } s} \\
\\
\text{IFC-APPASN} \quad \frac{\Gamma \vdash f : \tau_1 \rightarrow \tau_2 \quad \Gamma \vdash e : \tau_1 \quad \text{pc} \sqsubseteq \tau_2}{\Gamma, \text{pc} \vdash f e} \quad \text{IFC-ARRASN} \quad \frac{\Gamma \vdash a : \tau \quad \Gamma \vdash e_1 : \text{obs} \quad \Gamma \vdash e_2 : \tau' \quad \tau' \sqsubseteq \tau \quad \text{pc} \sqsubseteq \tau}{\Gamma, \text{pc} \vdash a[e_1] := e_2}
\end{array}$$

**Definition 13.3.** Security Lattice

The security labels in an IFC type system form a lattice structure, which defines the ordering and relationships between different security levels. In our case, we have two labels: *sec* (confidential) and *obs* (observable). The lattice can be visualized as follows:

$$\begin{array}{c}
\text{sec} \\
\uparrow \\
\text{obs}
\end{array}$$

In this lattice, *sec* is considered to be a higher security level than *obs*. The ordering relation  $\sqsubseteq$  is defined such that:

$$\text{obs} \sqsubseteq \text{sec}$$

The join operation  $\sqcup$  is defined as the least upper bound of two security labels. For our labels, we have:

$$\begin{aligned}
\text{obs} \sqcup \text{obs} &= \text{obs} \\
\text{obs} \sqcup \text{sec} &= \text{sec} \\
\text{sec} \sqcup \text{sec} &= \text{sec}
\end{aligned}$$

## 13.2 Solution

To determine whether the program *s* type checks under the IFC type system with the initial program counter label *obs*, we need to analyze the program using the IFC typing rules. The program *s* is given as:

$$\text{if } (y \geq 0) \text{ then } x := 2$$

We start with the typing context  $\Gamma$  where:

$$\Gamma(x) = \text{obs} \quad \text{and} \quad \Gamma(y) = \text{sec}$$

We need to check if the typing judgment  $\Gamma, \text{obs} \vdash s$  holds.

$$\text{IFC-IF} \quad \frac{\Gamma \vdash (y \geq 0) : \text{sec} \quad \Gamma, \text{obs} \sqcup \text{sec} \vdash x := 2}{\Gamma, \text{obs} \vdash \text{if } (y \geq 0) \text{ then } x := 2}$$

Here we need to verify two conditions:

- *Condition 1:* We need to check if  $\Gamma \vdash (y \geq 0) : \text{sec}$ . Since *y* is labeled as *sec* in  $\Gamma$ , the expression  $(y \geq 0)$  is indeed of type *sec*.
- *Condition 2:* We need to check if  $\Gamma, \text{obs} \sqcup \text{sec} \vdash x := 2$ . The join  $\text{obs} \sqcup \text{sec}$  results in *sec*. Now we apply the IFC-Assign rule:

$$\text{IFC-ASSIGN} \quad \frac{\Gamma \vdash x : \text{obs} \quad \Gamma \vdash 2 : \text{obs} \quad \text{obs} \sqsubseteq \text{obs} \quad \text{sec} \sqsubseteq \text{obs}}{\Gamma, \text{sec} \vdash x := 2}$$

Here, all the conditions of the IFC-Assign rule are satisfied:

- $\Gamma \vdash x : \text{obs}$  holds since  $\Gamma(x) = \text{obs}$ .
- $\Gamma \vdash 2 : \text{obs}$  holds since constants are typically considered to have the lowest security level.
- $\text{obs} \sqsubseteq \text{obs}$  holds trivially.
- $\text{sec} \sqsubseteq \text{obs}$  does not hold, which is a violation of the IFC-Assign rule.

Since the last condition  $\text{sec} \sqsubseteq \text{obs}$  does not hold, we conclude that the program  $s$  does not type check under the IFC type system with the initial program counter label  $\text{obs}$ . Therefore, the typing judgment  $\Gamma, \text{obs} \vdash s$  does not hold.

## 14 Question - Security Attacker Model

Say, we've proved non-interference for a program  $s$ . Can we conclude that the program is now secure? That is, have we proved that there can be no more leaks in the program? Explain your reasoning.

### 14.1 Theory

#### Definition 14.1. Non-interference

We define a restriction of the domain of the state  $\sigma$  to the observable variables as  $\sigma \upharpoonright_{\text{obs}}$ . We consider two states  $\sigma_1$  and  $\sigma_2$  to be *observationally equivalent* if they agree on all observable variables, i.e., (but not necessarily on secret variables):

$$\sigma_1 \equiv_{\text{obs}} \sigma_2 \iff \forall x \in \text{Var}. \Gamma(x) = \text{obs} \rightarrow \sigma_1(x) = \sigma_2(x)$$

A program  $s$  satisfies *non-interference* if for any two initial states  $\sigma_1$  and  $\sigma_2$  that are observationally equivalent, the final states after executing  $s$  from these initial states are also observationally equivalent. Formally,

$$\forall \sigma_1, \sigma_2. (\sigma_1 \upharpoonright_{\text{obs}} \equiv \sigma_2 \upharpoonright_{\text{obs}}) \wedge ((s, \sigma_1) \Downarrow \sigma'_1) \wedge ((s, \sigma_2) \Downarrow \sigma'_2) \rightarrow ((\sigma'_1 \upharpoonright_{\text{obs}} \equiv \sigma'_2 \upharpoonright_{\text{obs}}))$$

#### Definition 14.2. Attacker models

To resolve the seeming contradiction of non-interference saying something is secure and cache/timing attacks suggesting otherwise we introduce the notion of *attacker models*. The idea being that non-interference provides security guarantees against a certain class of attackers, while cache/timing attacks consider a more powerful attacker model. In other words the notion of security is always defined relative to an attacker model, i.e. you first consider the capabilities of an attacker and then prove security against that model.

### 14.2 Solution

No, we cannot conclude that the program is secure solely based on the proof of non-interference. While non-interference is a strong security property that ensures that secret information does not influence observable outputs, it does not account for all possible attack vectors.

Non-interference guarantees that if two initial states are observationally equivalent (i.e., they agree on all observable variables), then the final states after executing the program will also be observationally equivalent. This means that an attacker who can only observe the outputs of the program cannot distinguish between different secret inputs based on those outputs.

However, non-interference does not consider side-channel attacks, such as timing attacks or cache attacks, where an attacker may gain information about secret data through indirect means. For example, an attacker might measure the time it takes for a program to execute or observe changes in cache behavior to infer information about secret variables. These types of attacks exploit implementation details rather than the logical flow of information in the program.