Mathematical Logic and its Application to Computer Science

Lecture Notes

Eitan Farchi  Yochai Ben-Chaim
farchi@il.ibm.com  yochaib@il.ibm.com

March 3, 2010
Contents

1 Introduction 9
  1.1 The Knaves and the Knights 9
  1.2 Inductive Definitions of Sets 10
    1.2.1 Initial Definition 10
    1.2.2 MI/MU Example 10
    1.2.3 A Formal Definition 13
    1.2.4 Bottom-Up Definition 14
    1.2.5 Induction Proof Principle 16
  1.3 Propositional Calculus 17
    1.3.1 Syntax 17
    1.3.2 Axiom 19
  1.4 Ruling Function from the Talmud 20

I Lecture Transcripts 23

2 Lecture - Inductive Sets 23
  2.1 Definition 23
    2.1.1 Generic Example 1 24
    2.1.2 Generic Example 2 25
    2.1.3 Vertex Example 25
    2.1.4 Using Inductive Sets in Defining Program Behavior 27
    2.1.5 MI/MU Example 28
    2.1.6 Operative and Declarative Definitions 35
    2.1.7 Inductive Sets in Computer Science 36
3 Lecture - Inductive Sets Continued

3.1 Propositional Calculus ........................................... 43
3.2 Induction on the Structure ...................................... 44
3.3 Tautology ............................................................ 44
   3.3.1 Syntax ......................................................... 45
   3.3.2 Meaning of True Sentences ................................. 51
   3.3.3 Tautologies .................................................. 54

4 Lecture - Hierarchical Glossaries Example

4.1 Main Example ...................................................... 60
4.2 Hierarchy of Dictionaries ........................................ 65

5 Lecture - Hierarchical Glossaries Example, Continued

5.1 Hierarchy of Rules for Glossary Design ....................... 69
5.2 Introducing the Z Specification Language .................... 71
5.3 Hierarchical Dictionaries Example, Continued ............... 74
5.4 Defining Adding a Word and its Definition to a Dictionary .. 75
5.5 Dictionary Example ................................................. 77
5.6 Referred Sets ........................................................ 77
5.7 Z Specification Language Basic Signs ......................... 78

6 Lecture - Formal Specification Languages

6.1 Formal Specification Language - Definition .................. 82
6.2 Ruling Function from the Talmud Revisited ................... 86

7 Lecture - Communication Group Example

3
19.1.1 Purpose ...................................................... 179
19.1.2 Specification of Well-Formed Pairs of Words ........... 179
19.1.3 Operations .................................................. 180
19.1.4 Invariants ................................................... 181
19.1.5 Error Handling ............................................. 181
19.1.6 'Total Operations' ......................................... 182

20 Exercise - Defining a Specification 183

21 Exercise - Communication Group Example 185
   21.1 Communication Group Example, Revisited ................. 185
   21.2 Functional Coverage ....................................... 190

22 Exercise - Functional Coverage Models 192

23 Exercise - FoCuS Models 196
   23.1 Defining Attributes ....................................... 196
   23.2 Defining Constraints ...................................... 196

24 Exercise - Home Exercise #2 200

25 Exercise - Home Exercise #3 202

26 Exercise - Defining a Library Example 206
   26.1 Functions .................................................. 206
   26.2 Library Definition From Lecture #12 ..................... 209

III Home Exercises 211
27 Home Exercise #1 - Induction and Propositional Calculus -
   Version 1
  27.1 Induction Set ............................................ 211
  27.2 Propositional Calculus ................................. 213
  27.3 Simple Program .......................................... 214

28 Home Exercise #1 - Induction and Propositional Calculus -
   Version 2
  28.1 Induction Set ............................................ 216
  28.2 Propositional Calculus ................................. 218
  28.3 Simple Programs and Inductive Sets ................. 219

29 Home Exercise #2 - Functional Coverage in FoCuS - Version 1

30 Home Exercise #2 - Functional Coverage in FoCuS - Version 2

31 Home Exercise #3 - Formal Specification - Version 1

32 Home Exercise #3 - Formal Specification - Version 2

IV References 227
Abstract

The objective of the course is to introduce mathematical logic and explore its applications in computer science, with an emphasis on formal specifications and software testing.

During the course we cover these topics:

- Propositional Calculus
- Structured Induction
- Partial Orders
- First Order Logic
- Formal Specification using the $\mathbb{Z}$ specification language
- Some background in Set Theory, relations, functions and schemes
- Applications of Mathematical Logic to Formal Verification and program analysis

Part I contains transcripts of the lectures, while Part II provides the exercises that were covered. Part III contains the home exercises performed by the students of the course.
1 Introduction

1.1 The Knaves and the Knights

An island is inhibited by two types of people: knaves, who always lie; and knights, who always tell the truth.

Problem

A and B are from that island.

A says, ”At least one of us is a knave.”

Question

What are A and B: knaves or knights?

Solution

If A is a knave, then there are either one or two knaves. As a result, the statement that ”at least one of us is a knave” is true. However, this statement contradicts the fact that A is a knave because knaves always lie.

If A is a knight, then the statement ”At least one of us is a knave” is true if and only if B is a knave. As A is a knight, he is telling the truth, and for the statement to be true, B has to be a knave. Therefore, A must be a knight and B must be a knave.

The problem of the knaves and the knights has many variations. It exemplifies the use of mathematical logics for solving riddles. In this book you will learn how to use mathematical logics in the software development process.
1.2 Inductive Definitions of Sets

1.2.1 Initial Definition

$I(A, P)$ is the inductive set created from the atoms, $A$, using operations $P$. The inductive set is the set obtained from the set of atoms $A$ by repeatedly applying operations from the set $P$.

1.2.2 MI/MU Example

To define an inductive set, we define the atoms ($A$) and the operations ($P$). In this example, we define the set of atoms $A$ to be the set $\{MI\}$. That is, a single member MI is a string containing two characters. We define the operations $P$ as the following operations:

1. $XI \rightarrow XIU$ – which means you add a U at the end of any string ending with I.
2. $MX \rightarrow MXX$ – which means you duplicate everything that comes after an M.
3. $III \rightarrow U$ – which means you can take any sequence of three consecutive I’s and change them into a single U.
4. $UU \rightarrow \text{‘nothing’}$ – which means you omit two consecutive U’s.

We attempt to define the inductive set $I(A, P)$. According to the definition of the inductive set, the atom MI is in the inductive set.

Next, we repeatedly activate the operations on the members of the inductive
We activate operation #2 on MI to get MII.

We denote activating operation #2 on MI and the result MII as $\text{MI} \xrightarrow{\text{Operation 2}} \text{MII}$.

MII is therefore added to the inductive set. Similarly, $\text{MI} \xrightarrow{\text{Operation 2}} \text{MII}$, $\text{MII} \xrightarrow{\text{Operation 1}} \text{MIIIU}$, $\text{MIIIU} \xrightarrow{\text{Operation 3}} \text{MIUU}$ or $\text{MUIU}$. We therefore add MIIII, MIIIIU, MIUU, and MUIU to the inductive set $I(A, P)$.

**Problem**

Prove that $\text{MU} \notin I(A, P)$ - or in other words, that MU is not in the inductive set defined by the atom MI and the operations defined above. Another interpretation of the question is that if we start with the atom MI, and repeatedly activate the operations defined by P, in any possible order, we will never reach MU.

**Proof**

We prove this claim using induction.

We prove that $\text{MU} \notin I(A, P)$ using the claim that any member $w \in I(A, P)$ has a number of I’s that is not a multiple of three. If we can prove this claim, it will prove that $\text{MU}$ is not in $I(A, P)$, since the number of I’s in MU (zero I’s) is a multiple of three.

To prove using induction:

1. We prove for the atoms.

2. We assume the claim is true for any member $w$ in the inductive set, and show that the claim stays true after applying any of the operations on $w$. We term this part of the proof as the *inductive step*. 
We begin by proving for the atoms. We have a single atom in our example—MI—which has one I. One is not a multiple of three and therefore our claim holds.

Next, we assume that for any \( w \in I(A, P) \), the number of I’s in \( w \) is not a multiple of three. We need to show that the claim remains true after activating any of the operations. To prove the claim, we iterate all the possible operations, and show that the claim is true for each possible operation:

- For the first rule, \( XI \rightarrow XIU \): adding a U at the end of a word does not change the number of I’s in the word. Since we assume that any word \( w \) does not have a number of I’s that is a multiple of three before activating the operation, the number of I’s after the operation remains not a multiple of three.

- For the second rule, \( MX \rightarrow MXX \): the number of I’s is multiplied by two. Since the number of I’s in any word \( w \) before the operation was not a multiple of three, we can mark the number of I’s in \( w \)—before the operation—as either \( 3k + 1 \) or \( 3k + 2 \) (for a natural number \( k \)).

  - If the number of I’s in \( w \) was \( 3k + 1 \), then the number of I’s after applying the operation is multiplied by two: \( 2(3k + 1) = 6k + 2 \), which is not a multiple of three.

  - If the number of I’s in \( w \) was \( 3k + 2 \), then the number of I’s after applying the operation is multiplied by two: \( 2(3k + 2) = 6k + 4 = 3(2k + 1) + 1 \). Since \( k \) is a natural number, \( 2k + 1 \) is a natural number, and \( 3(2k + 1) + 1 \) is not a multiple of three.
• For the third rule: III → U, the number of I’s was reduced by three (three I’s were replaced by a U). If the number of I’s in w before the operation was not a multiple of three, after applying the operation, the number of I’s is reduced by three, and remains not a multiple of three.

• For the fourth rule, UU → 'nothing': omitting two U’s does not change the number of I’s in the word. Since we assumed that any word w does not have a number of I’s which is a multiple of three before activating the operation, the number of I’s after the operation remains unchanged and is still not a multiple of three.

This completes the induction step and the proof.

1.2.3 A Formal Definition

A formal definition of the Inductive Set I(A, P):

Given a set of atoms A, and a set of operations P, an inductive set of A and P is a set that is defined by these rules:

1. Contains A.

2. Is closed under operations in P. In other words, if we know that X_1, X_2, ..., and X_k are all in the inductive set, and Z is obtained from X_1, X_2, ..., X_k by one of the operations in P, then Z is also necessarily in the inductive set.

Example

Consider the following example: A = \{1\} and P = +1.

The set \{1, 2, 3, ...\} preserves the rules we defined, and therefore is the
inductive set. But, we also note that the set of real numbers: \{ \ldots, -3, -2, -1, 0, 1, 2, 3, \ldots \} and the set of complex numbers also preserve the rules; that is, all three sets contain the atoms and preserve closure over the operation of adding one.

This example shows that the two rules provided do not define a unique set. To correct this situation, we refine the formal definition of the inductive set, adding a third rule:

Given a set of atoms A and a set of operations P, an inductive set of A and P is a set that

1. Contains A.

2. Is closed under operations in P.

3. Is the minimal (under inclusion) set that meets the previous two rules.

In the example above, the set of natural numbers: \{1, 2, 3, \ldots\} is contained in \{ \ldots, -3, -2, -1, 0, 1, 2, 3, \ldots \} which are the set of real and the set of complex numbers. Therefore, we assume that \( I(\{1\}, +1) = \mathbb{N} = \{1, 2, 3, \ldots\} \) as the set of natural numbers.

To prove the claim that the set of natural numbers is the minimal set that contains the atom and is closed under operations in P, we need a bottom-up characterization of the inductive set I(A, P).

### 1.2.4 Bottom-Up Definition

We define the set \( I_1 \) as the set obtained from applying any of the operations in P once to any atom in A.
We define the set \( I_2 \) as the set obtained from applying any of the operations in \( P \) once to any member of \( I_1 \).

As a general definition, we define the set \( I_k \), for any natural number \( k \in \mathbb{N} \), as the set obtained from applying any of the operations in \( P \), \( k \) times, in any possible order, on any atom in \( A \).

We take the union of all the elements we obtained: \( \bigcup_{k=1}^{\infty} I_k \) and term it \( I'(A, P) \).

Using the MI/MU example from Section 1.2.2, \( I_1 \) is the set obtained from applying any of the operations in \( P \) once on any atom in \( A \) (in this case, just MI).

\[
\begin{align*}
\text{MI} & \xrightarrow{\text{Operation} \#1} \text{MIU} \\
\text{MI} & \xrightarrow{\text{Operation} \#2} \text{MII}
\end{align*}
\]

Applying operations \#3 and \#4 does not change MI.

Therefore, \( I_1 \) is the set \( \{ \text{MI, MIU, MII} \} \).

Similarly, \( I_2 \) is the set \( \{ \text{MI, MIU, MII, MIII, MIUIU, MIIU} \} \).

Returning to the example of \( I(\{1\}, +1) \), we notice that \( I_1 = \{1, 2\} \), \( I_2 = \{1, 2, 3\} \) and \( I_k = \{1, 2, \ldots, k + 1\} \). We therefore conclude that \( I'(A, P) \), which was defined as \( \bigcup_{k=1}^{\infty} I_k \), is the set of natural numbers \( \{1, 2, 3, \ldots\} \).

It remains to be proven that \( I'(A, P) = I(A, P) \).

Claim

Prove that \( I'(A, P) = I(A, P) \).

Proof

We prove the claim using bidirectional inclusion.

To prove that \( I'(A, P) \) is the inductive set, we must show it preserves the three rules that define the inductive set:
• It contains the atoms in the set A.

• It is closed under the operations in the set P.

• It is the minimal set that preserves the first two rules.

$\Gamma(A, P)$ contains the atoms in A by definition.

According to the definition of $I_n$, any $X_1, X_2, ..., X_k$ that are in $I_1$ are also in $I_n$.

By definition, the application of any operation in P on $X_1, X_2, ..., X_k$ results in members that are all in $I_{n+1}$. Therefore, $\Gamma(A, P)$ is an inductive set that contains the atoms and is closed under operations.

Since we defined $I(A, P)$ as the inductive set defined by $I(\{1\}, +1)$, it is the minimal set that contains the atoms and is closed under operations. Therefore $I(A, P)$ is contained in $\Gamma(A, P)$.

On the other hand, any element $y \in \Gamma(A, P)$ was obtained by applying operations from P a finite number of times, starting with the atom A, which means that $y$ is also $\in I(A, P)$.

Since we showed that every element in $\Gamma(A, P)$ is also in $I(A, P)$, so $\Gamma(A, P)$ is contained in $I(A, P)$. Since we also showed that $I(A, P)$ is contained in $\Gamma(A, P)$, the two sets must be equal to each other: $I(A, P) = \Gamma(A, P)$.

### 1.2.5 Induction Proof Principle

In the MI/MU example in Section 1.2.2, we proved that MU is not in the inductive set $I(A, P)$, created from the atom $A = \{MI\}$ by applying the operations. We proved it using the principle of induction. The induction method of proof:
1. First, show that a claim $T$ is true for the atom $(A)$.

2. Second, if we assume the claim $T$ is true for any member of the set before applying an operation from $P$, it remains true after applying that operation.

We deduce that $T$ is true for $I(A, P)$. But we must ask the question: Why are we allowed to deduce using the inductive principle?

The fact is that the set of elements for which $T$ is true is an inductive set. By abuse of notation, we denote the set of elements for which the claim $T$ is true by $T$. Thus, $T \supseteq I(A, P)$ (since $I(A, P)$ is defined as the minimal inductive set for the atoms $A$ and operations $P$). This means that for every element of $I(A, P)$, the claim $T$ holds.

### 1.3 Propositional Calculus

#### 1.3.1 Syntax

We define the syntax of the propositional calculus as follows: atoms are letters or indexed letters, (e.g., $b \in A$, $b_i \in A$, and so on). In addition, if $Q$ and $R$ are $\in I(A, P)$, then so are these:

- $(\neg Q)$
- $(Q \lor R)$
- $(Q \land R)$
- $(Q \rightarrow R)$
We refer to $I(A, P)$ as the set of sentences of propositional logic. The intended meanings of $\neg$, $\lor$, $\land$, and $\rightarrow$ are "not", "or", "and", and "imply", respectively. However, these are just the intended meanings at this point of the discussion. Compare our current situation to that of a programming language without a compiler.

Thus, in our hypo-statical programming language, we might have intended that: "if $x = 5; t = 3; (g(x, x++) > 0); g(x, f(t));" means some sort of calculation. However, if we do not define the calculation, it is just a string of characters devoid of meaning. As a string of characters, it could possibly be a legal string. Thus, $I(A, P)$ defines the legal strings of propositional calculus (or in the analogy, the "programs" that will compile and have correct syntax, though that does not mean they will actually perform something that makes sense).

**Claim 1**

We continue to investigate the syntax. We claim that for any sentence, if it is not the atom, it begins with an opening parenthesis "."

**Proof 1**

$Q$ and $R$ begin with ",", then clearly, so do $(\neg Q)$, $(Q \lor R)$, $(Q \land R)$, and $(Q \rightarrow R)$. Thus, $\neg(q \lor r)$ is not a sentence since it is not the atom, and it does not begin with ",".

**Claim 2**

The number of opening parentheses "," and the number of closing parentheses "" in a sentence is equal.

**Proof 2**

To prove, we use the inductive proof method. According to the inductive
proof method:

1. We show that the claim holds for the atoms.

2. We assume the claim holds for any member of the set and show that the claim still holds after applying any of the operations.

The claim holds for the atoms, since the number of opening and closing parentheses is zero. Next, assuming that the number of opening and closing parentheses in $Q$ and $R$ is equal, we need to show that it is still equal after activating the operations. We note that each operation ($\neg Q$), ($Q \lor R$), ($Q \land R$), and ($Q \rightarrow R$) adds a single opening and a single closing parenthesis. Therefore, equality is kept.

1.3.2 Axiom

We next define the sentences that are ”always correct” (at least, this is the intended meaning) inductively. The atoms $A$ are sentences of this form:

- $(B \rightarrow (C \rightarrow B))$
- $((B \rightarrow (C \rightarrow D)) \rightarrow ((B \rightarrow C) \rightarrow (B \rightarrow D)))$
- $(((\neg B) \rightarrow (\neg C)) \rightarrow (C \rightarrow B))$

where $B$, $C$, and $D$ are any legal sentences in the propositional language. We further term the above atoms as ”axioms”.

Note that for any true values of $B$, $C$, and $D$, the sentences above are intuitively true.

A more precise definition and explanation of the sentences becomes clear
once we define an interpretation of propositional language sentences. P is defined as a single operation: if \((B \rightarrow C)\) and \(B\) are already in the set of "always correct", then so is \(C\) (this operation is referred to as "separation"). Therefore, the inductive set \(I(A, P)\) is intended to define the sentences that are "always true". Note that at this stage, this is only our intention and there is no bearing on the formalism.

One last element of notation at this stage: if \(a \in I(A, P)\), we say that \(\vdash a\). In this case, there is a sequence by which \(a\) is obtained from the atoms of \(I(A, P)\). We refer to this sequence as a proof for \(a\) - \(\vdash a\).

Next, let’s formally prove that \((a \rightarrow a) \rightarrow (a \rightarrow a)\) is indeed always true:

- \(a \rightarrow (a \rightarrow a)\) (axiom)

- \((a \rightarrow (a \rightarrow a)) \rightarrow ((a \rightarrow a) \rightarrow (a \rightarrow a))\)

- \((a \rightarrow a) \rightarrow (a \rightarrow a)\) (separation)

### 1.4 Ruling Function from the Talmud

Assume the following rules: If a bull never attacked before, yet attacks and kills another bull, then the owner of the first bull pays the owner of the second bull compensation in the amount of half the value of the second bull, but only up to the value of the first bull.

Example

The first bull is worth 500 coins and the second is worth 2000 coins. The first bull kills the second bull. The owner of the first bull pays the owner of

\[\text{1The Talmud is a central text of mainstream Judaism, in the form of a record of rabbinic discussions pertaining to Jewish law, ethics, customs, and history}\]
the second bull 500 coins, since the value of the first bull is less than half the value of the second bull.

If the first bull is worth 1000 coins, then 1000 coins are paid. If the first bull is worth 1500 coins, then 1000 coins are paid, since half the value of the second bull is less than the value of the first bull.

Now assume that one of two bulls, owned by one person, killed a third bull owned by another person, but we don’t know which of the bulls killed the third bull. The third bull is worth 2000 coins while the first and second bulls are worth 500 coins and 1000 coins respectively.

If the first bull killed the third bull, then the owner of the first two bulls should pay the owner of the third bull 500 coins. If the second bull killed the third bull, then the owner of the first two bulls should pay the owner of the third bull 1000 coins.

But we don’t know which of the two bulls killed the third bull!

We simultaneously ask both owners of the bulls which bull killed the third bull. Each owner can answer one of the following answers:

- I don’t know (ignorant).
- The first bull did it (first).
- The second bull did it (second).

We model this by Answer = \{ignorant, first, second\}.

The set of possible responses of answers is modeled by the Cartesian Product Answer X Answer.

We are looking for a ruling function such that ruling: A \times A \rightarrow \{0 \text{ coins,}
One ruling function that appears in the Talmud is that if there is an agreement, or at least no disagreement, on who killed the third bull, we go by the undisputed claim and apply the rule that we describe above.

Thus, the ruling function is defined as follows:

\{ (\text{ignorant, ignorant, 0 coins}), (\text{ignorant, first, 500 coins}), (\text{ignorant, second, 1000 coins}), (\text{first, ignorant, 500 coins}), (\text{first, first, 500 coins}), (\text{first, second, 0 coins}), (\text{second, ignorant, 1000 coins}), (\text{second, first, 0 coins}), (\text{second, second, 1000 coins}) \}\}

For those familiar with Game Theory, note that this function defines a Matrix Game.

Does it have a Nash-Equilibrium point in pure strategies?

<table>
<thead>
<tr>
<th></th>
<th>II</th>
<th>Don’t Know</th>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>I</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Don’t Know</td>
<td>0 coins</td>
<td>500 coins</td>
<td>1000 coins</td>
<td></td>
</tr>
<tr>
<td>A</td>
<td>500 coins</td>
<td>500 coins</td>
<td>0 coins</td>
<td></td>
</tr>
<tr>
<td>B</td>
<td>1000 coins</td>
<td>0 coins</td>
<td>1000 coins</td>
<td></td>
</tr>
</tbody>
</table>

Table 1 clearly describes why there is no Nash-Equilibrium in this case. For each entry in the table, at least one of the players has a reason to change his answer to improve the result in his favor.
Part I

Lecture Transcripts

This course covers the uses of Mathematical Logic in Computer Science. Specifically we examine formal specifications (what a computer program is supposed to do), improving compiler optimization (in brief), and program semantics. The purpose is to show how they connect to mathematical logics.

2 Lecture - Inductive Sets

In this lecture we cover these topics:

1. The concept of the inductive set.

2. How to use the inductive set.

3. Operative and declarative definitions of inductive set.

4. Why it is interesting to try to characterize the inductive set.

5. How to use the inductive set in the context of computer programs.

2.1 Definition

Given a set of "atoms" and a set of operations, an Inductive Set is obtained from its atoms by repeatedly applying the operations.

We denote the inductive set obtained from the set of atoms A using the operations in P by $I(A, P)$.
2.1.1 Generic Example 1

We consider the inductive set $I(\{0\}, +1)$ where 0 is the only atom, and the operation is adding 1 (denoted by $+1$ above). What is the inductive set?

We begin with the atoms, and repeatedly apply the operations. Adding 1 to the atom 0 we obtain 1. Adding 1 to 1 we obtain 2, and so on. This process can be presented graphically as:

\[
\begin{array}{c}
0 & \xrightarrow{+1} & 1 & \xrightarrow{+1} & 2 & \xrightarrow{+1} & 3 & \ldots \\
\end{array}
\]

We quickly realize that the result is the set of natural numbers $\mathbb{N} = \{0, 1, 2, 3, \ldots\}$.

Note that some numbers are not in the above inductive set: for example, -1. This can be denoted by $-1 \not\in I(\{0\}, +1)$ or $-1 \not\in \mathbb{N}$.

To receive all $\mathbb{Z}$ (the set of all integers), we add another operation: -1.

How—if at all—can we write $I(A, P)$ to receive all the rational and irrational numbers?

Trivial answer 1: define $A$ as all the rational and irrational numbers, and no operations are needed. Trivial answer 2: $A = \text{all the rational and irrational numbers between 0 and 1, and operations } +1, \text{ and } -1$.

In these trivial cases, we gave an infinite set of atoms. Can we start with a finite number of atoms, or an ordered list of atoms? The answer is no. You cannot reach all the numbers, whether from a finite or ordered set of atoms.

\(^2\{\} \) denotes a set
2.1.2 Generic Example 2

Another example (not numeric):
A is a set of signs: $A = \{a, b, c, d, ..., z\}$ (all the small caps letters of the English language).

$P = \text{concatenation}$.

If $abc$ is in $I(A, P)$, and if $cd$ is in $I(A, P)$ then $abcd$ is in $I(A, P)$.

$I(A, P) = \text{all the strings that use English small caps letters}$.

Note that $I(A, P)$ is an infinite set, since $n$ times $a$ is a member of $I(A, P)$ denoted $aaaa...a$ ($n$ times) $\in I(A, P)$ since we can show the activation series ($n$ times the concatenation operation on the atom $a$). Additionally, we note that $\{a, aa, aaa, aaaa, ... \} \subseteq I(A, P)$. Since $\{a, aa, aaa, aaaa, ... \}$ is infinite, and $I(A, P)$ is larger, then $I(A, P)$ is infinite.

We can describe the inductive set in two ways: one is bottom-up, and the second is based on closure and minimalism. Closure means that activating the operation creates a member that is already in the set.

2.1.3 Vertex Example

For instance, if we look at the inductive set that is all the points in a vertex, the line representing $y = x$ and the operation of adding a vector is still a point on the line $y = x$.

But all the plain is closed under addition. If we take the point $(1, 1)$ and add (only add)—or multiply by a scalar—then all the plain is closed, but not minimal.
I(A, P): A = \{(0, 0), (1, 1)\}

P = multiplying by a scalar, and adding/subtracting vectors. Subtraction is induced by first activating multiplication by a scalar -1, and then adding.

We can remove (0, 0) from the atoms, since we can receive it by multiplying (1, 1) by -1, and then adding (1, 1) and (-1, -1).

The outcome of I(A, P) is the line y=x.

Inductive claim: I(A, P) = the line y=x. Let’s prove it using induction

Proof

(1, 1) is the atom and is therefore it is in I(A, P), and is on the line y=x as 1=1.

THIS PARAGRAPH REQUIRES REWRITING:

Inductive step (closure - activating the operations generates members which are in the set) - assume the claim is true I(A, P) = the line y=x, and show that after activation it is still the line. Assume (a, a) is in I(A, P). Since it is in I(A, P), and according to our assumption, it is on the line y=x, therefore we have (a, a). If we activate the first operation (multiplication by scalar c) we receive the point (ca, ca), and since ca=ca, it is on the line y=x. If we activate the second operation, on (a, a) and (b, b) (which are on the line y=x) and then add, we receive (a+b, a+b), which is also on the line y=x. Therefore, I(A, P) ⊆ the line y = x.

We can identify two types of sets that are closed under the operations: the line y=x and the whole plain (\(R^2\)). I(A, P) ⊆ \(R^2\) and I(A, P) ⊆ the line y = x. But \(R^2\) is not the inductive set, since it contains y = x, and so \(R^2\) does not preserve that it is minimal.
Next, we need to prove that the line $y = x$ is minimal (and therefore the inductive set).

### 2.1.4 Using Inductive Sets in Defining Program Behavior

The question we address here is, "What is the relation between inductive sets and software engineering?"

An important part of the software engineering process of a system is defining its requirements (what the system should do, as opposed to "how" it should do it). Logic tools in general, and inductive sets in particular, help us define what the system should do or actually does.

In the following example, we define what a small program snippet does using the inductive set $I\{\{0\}, +1\}$:

```c
x = 0;

while (true) {
    x++;
    print(x);
}
```

If this program snippet is left to run indefinitely, it prints the natural numbers $\mathbb{N}$.

**Claims**

1. We claim that $\mathbb{N} = I\{\{0\}, +1\}$.

   The claim is that the set deduced from the operation $+1$, starting from atom $0$, is the set of natural numbers $\mathbb{N}$.  
   
27
2. The program snippet above will print the set of numbers \( \mathbb{N} \).

We show that both claims are the same claim.

Showing that both claims are the same claim is part of the connection between the logical tool (the inductive set) and the claim about what the program does (a computer science requirement).

During the course lectures, we use logics to define what artifacts are supposed to do.

In this case, we perform a "logical jump" from a program snippet to the definition of the inductive set \( I(\{0\}, +1) \) without proof.

Next, we attempt to prove and explain this inductive set, using a different definition for the inductive set \( I(A, P) \), where \( A \) represents the "atoms" and \( P \) represents the operations.

The inductive set \( I(A, P) \) can be defined as the set that preserves that

- All the atoms are in the set (that is, all the members of \( A \) are in the set).

- If \( x_1, x_2, \ldots, x_y \) are in \( I(A, P) \) and if operation \( p_1 \) is in \( P \), then \( p_1(x_1, x_2, \ldots, x_y) \) is also in \( I(A, P) \).

In other words, the result of activating the operation \( p_1 \) on the members of the set is also in the set.

### 2.1.5 MI/MU Example

Given the atom \( A = \{\text{MI}\} \) and given the operations (P):

- O1: \( \text{XI} \rightarrow \text{XIU} \)
• O2: MX → MXX

• O3: III → U

• O4: UU → 'nothing'

Explaining the operations:

• O1 means that we add a U to any string ending with I.

• O2 means that we duplicate anything that appears after an initial M.

• O3 means that we replace three I’s with a single U.

• O4 means that we omit two consecutive U’s.

Let’s define what the inductive set I(A, P) is in this case, starting with the atom {MI}. MI is the only member of A (a single atom).

1. Activating operation O2 on MI results in MII. We can write this as O2(MI) = MII or MI \( \overset{O2}{\rightarrow} \) MII. MII is added to the inductive set I(A, P).

2. Activating operation O1 on MII results in MIIU. We can write this as O1(MII) = MIIU or MII \( \overset{O1}{\rightarrow} \) MIIU. MIIU is added to the inductive set I(A, P).

3. Activating operation O2 on MIIU results in MIIUIIU. We can write this as O2(MIIU) = MIIUIIU or MIIU \( \overset{O2}{\rightarrow} \) MIIUIIU. MIIUIIU is also added to the inductive set I(A, P).
Question 1 about the MI/MU Example
How can we reach a member on which we can activate operation O3?

Answer
By activating O2 twice on MI, we receive

- O2(MI) = MII
- O2(MII) = MIIII

Now, we can activate O3 on MIIII: O3(MIIII) = MUI or O3(MIIII) = MIU.

Question 2 about the MI/MU Example
Find a way to reach a member on which we can activate the operation O4.

Answer
We leave this question for the reader to determine.

Returning to the defining of the inductive set
We define "closure" as activating an operation on members of the set, which results in members that are also in the set.

However, the definitions that we gave (all the atoms are in the set and closure) are insufficient because many sets preserve these conditions. To resolve this issue we add an additional condition to the definition.

Example
Let’s re-examine our example inductive set from Section 2.1.1 where we start with the number zero (our atom), and the operation adds 1: I(\{0\}, +1).

According to our definition of the inductive set, we are looking for a set where the atoms are contained (zero is in the set), and the set is closed under the operations (adding 1 in this example).

Intuitively, we would like our definition to denote that the inductive set is the
set of natural numbers \( \mathbb{N} \) (with zero). But the set of all numbers \( \mathbb{Z} \), which includes the negative numbers, also preserves these two conditions because zero is in the set, and for every member in the set, the member plus 1 is also in the set.

Another set that preserves these two conditions is the set of all real numbers \( \mathbb{R} \), which includes fractions, and also includes zero and every number plus 1. We deduce that our definition is insufficient, and we require an additional (third) condition. If we look at the example, we see that \( \mathbb{N} \) is contained in \( \mathbb{Z} \), which is contained in \( \mathbb{R} \), so we mark it as \( \mathbb{N} \subseteq \mathbb{Z} \subseteq \mathbb{R} \).

Questions

What is the additional condition that we need to add?

How do we characterize the desired set?

Answers

We want the \textit{smallest set} that preserves the first two conditions (the \textit{smallest set} under the subset \( \subseteq \) relation). This set is contained in every other set that preserves the first two conditions. In other words, \( I(A, P) \) is the set contained in every other set \( B \) that preserves the first two conditions (all the atoms are in the set and closure).

We can therefore conclude a second definition for the inductive set:

\( I(A, P) \) where

- \( A \) is in the set \( A = A_0 \).

- \( A_1 \) is the set of all members that result from members of \( A_0 \) after activating all the operations in any legal way.

- \( A_2 \) is the set of all members that result from members of \( A_1 \) after
activating all the operations.

• And so on.

Then, \( I(A, P) \) is the unity \( \cup \) from 1 through infinity \( \infty \) of \( A_i \)’s such that
\( I(A, P) = \bigcup_{i=1}^{\infty} A_i \)

If we return to the previous MI/MU example, the atom was \( A = \{MI\} \) and the operations \( (P) \) were

• \( O_1: XI \rightarrow XIU \)
• \( O_2: MX \rightarrow MXX \)
• \( O_3: III \rightarrow U \)
• \( O_4: UU \rightarrow 'nothing' \)

Let’s test our new definition. We begin with the set of just the atoms:
\( A_0 = \{MI\} \)

We activate all the possible operations on all the members of \( A_0 \) to obtain all the members of \( A_1 \).

• \( O1(MI) = MIU \)

• \( O2(MI) = MII \)

We cannot activate \( O3 \) and \( O4 \) on \( MI \), so \( A_1 \) is \( \{MI, MIU, MII\} \).

To obtain \( A_2 \) we need to activate all the four possible operations on all the members of \( A_1 \) (MI, MIU and MII) in any way possible. The legal operations include \( O2(MIU) = MIUIU \).

We cannot activate the three other operations \( (O1, O3, \text{and } O4) \) on \( MIU \).

As for activating the operations on \( MII \):
• O1(MII) = MIIU

• O2(MII) = MIIII

We cannot activate O3 and O4 on MII.

Another way of describing each member in the set is to use the list of operations that resulted in the member. For instance, MIUIU was created from MI by first activating O1 and then O2.

We call the series of operations that begin with an atom and result in the member the "creation series" of the member. The creation series is also sometimes referred to as the derivation or proof.

Claim

1. We mark $I^\ast(A, P)$ as the set: $\bigcup_{i=1}^{\infty} A_i$.

2. We claim $I^\ast(A, P) = I(A, P)$.

Proof of Claim 1

To show that this is true, we use bi-directional inclusion to show that the left side is contained in the right and the right side contained in the left, and therefore they must be equal.

Let’s consider intuitively why this must be true. The proof shows that the set created by activating the operations is the set that preserves both conditions—that the atoms are contained and closure over the operations.

To prove that $I^\ast(A, P)$ is the inductive set $I(A, P)$, it is not enough to show that it contains the atoms and preserves closure over the operations. We also need to show that this is the minimal set that preserves the two conditions.

Why does $I^\ast(A, P)$ contain the atoms? Because $I^\ast(A, P)$ was initially derived
from them, since according to the definition, \( A_0 = A \). Since \( I_\star(A, P) = \bigcup_{i=1}^{\infty} A_i \), it includes \( A_0 \).

**Proof of Claim 2**

Let’s show that \( I_\star(A, P) \) is closed over the operations. To prove this, we need to select a member from the set, activate the operations, and show that the resulting member is also in the set.

We do this by showing that any member in the set has a creation series from the atoms, and thus must be in the set. We take the members \( x, y \) from \( I_\star(A, P) \) and activate an operation \( o \) from \( P \) on them.

We need to show that \( o(x, y) \) is already a member of \( I_\star(A, P) \). This is true because \( x \) and \( w \) were both derived from a creation series of activating operations. Therefore, when we activate the operation \( o \) on \( x \) and \( y \), we have a creation series that results in \( x \) and a creation series that results in \( y \), and so we have a creation series to reach \( o(x, y) \), and therefore \( o(x, y) \) is in \( I_\star(A, P) \).

So far, we have shown that the atoms are in \( I_\star(A, P) \), and that \( I_\star(A, P) \) preserves that it is closed under the operations in \( P \). Or, in other words, we show that \( I(A, P) \subseteq I_\star(A, P) \) because \( I_\star(A, P) \) contains the atoms and is closed over the operations, and we know that \( I(A, P) \) is the minimal set that preserves these two conditions.

To prove that the sets are equal; i.e., \( I(A, P) = I_\star(A, P) \), we need to show that \( I_\star(A, P) \subseteq I(A, P) \).

**Proof**

Each member \( x \) in \( I_\star(A, P) \) has this creation series:

\[ x_1 \rightarrow x_2 \rightarrow ... \rightarrow x_k = x. \]
But this x must also be in $I(A, P)$. So, according to closure, $I_*(A, P) \subseteq I(A, P)$.

Therefore, we have proved that the two sets are each contained in the other, and therefore must be equal: $I(A, P) = I_*(A, P)$.

### 2.1.6 Operative and Declarative Definitions

We note that there are operative and declarative definitions:

- $I$ - is a declarative definition.
- $I_*$ - is an operative definition.

**Example**

$I(\{(1, 0), (0, 1)\}, \{v,w \rightarrow av + bw\})$

where $v$ and $w$ are points in the plane, and $a$ and $b$ are numbers in the set of real numbers $\mathbb{R}$.

For instance:

- $v = (0, 1)$
- $w = (1, 0)$
- $a = -1$
- $b = 3$

We receive $(-1 * (0, 1)) + (3 * (1, 0)) = (3, -1)$.

**Question**

What is the inductive set derived by this definition?

**Answer**
The inductive set is the whole plain $\mathbb{R}^2$. I is the whole plain, and the whole plain is the minimal set that defines the operations in I.

Sub-example

We are looking for the minimal sub-vector space that is closed under the linear combinations and is minimal under the subset relation.

If we only look at $I(\{(1,0), \{v \rightarrow av\})$, we receive only the $x$ axis.

Remembering the concepts of "basis" and "span" from linear algebra, according to the linear algebra language, if we take a vector space and a set of vectors and look at the span of the vectors, $\text{SPAN}(v_1, v_2, ... v_k)$, which is all the possible linear combinations, then a different definition of the span of the vectors is the minimal sub-vector space closed under the linear combinations, and thus contains the vectors.

If we translate this into the language of induction sets, our atoms $A$ are the vectors, and the operations $P$ are the linear combinations. In other words, we multiply each vector with a scalar and add the resulting vectors.

### 2.1.7 Inductive Sets in Computer Science

Consider the following code snippet:

```c
int x;

rand(x);

while (x > 0) {
    print (x);
    x -= ;
```

36
How can we define what this code snippet does using an inductive set? The operation P is subtraction, as long as the member is larger than zero. The atom A is the initial random number x.
The inductive set is all the natural integer numbers from \( \text{rand}(x) \) down to 1.

**Exercise**

How can we use an inductive set to define all the possible values of x?

**Example Program**

```java
if (a < b) {
    x = b-a;
    y = a-b;
} else {
    z = b-a;
    t = a-b;
}
```

The question is: Can we make this program more efficient using inductive sets?

Intuitively, we notice that a more efficient activation is achieved by performing the subtraction operations once only and then setting their values:

\[
\begin{align*}
    r &= b-a; \\
    f &= a-b;
\end{align*}
\]
if (a < b) {
    x = r;
    y = f;
} else {
    z = r;
    t = f;
}

We would like to write an automatic computer program that finds and makes this change.

Question
Write a computer program that automatically finds and suggests this change. In fact, computer compilers actually create efficiency operations. So how do inductive sets help with the process of creating effective compilers? For instance, optimization options can be set when activating the compiler so that the compiler actually changes the assembly code.

Answer
Let’s try and perform an abstraction of the inductive set to characterize the change in the program:

1. if(a < b) {
2.     x = b - a;
3.     y = a - b;
\[ \text{else} \{ \]

4. \( z = b - a; \)

5. \( t = a - b; \)

\}

First, we number the rows to mark them as operations.

We start with the set of atoms:

\[ A = \{1, \emptyset\}, \{2, \emptyset\}, \{3, \emptyset\}, \{4, \emptyset\}, \{5, \emptyset\} \]

Our goal is to define the inductive set \( I(A, P) \) so that it contains \( \{1, \{b - a, a - b\}\} \). That is, to associate the expressions \( a - b, b - a \) with operation 1.

What are the operations?

Each operation \( i \) means that when entering the command in line \( \{i, X\} \):

\( I \) \( \{2, Y\}, \{4, Z\}, \{1, X\} \rightarrow \{1, Y \cup Z \cup X\} \)

\( II \) \( \{3, Y\}, \{2, X\} \rightarrow \{2, X \cup Y \cup \{b - a\}\} \)

\( III \) \( \{3, X\} \rightarrow \{3, X \cup \{a - b\}\} \)

\( IV \) \( \{5, X\}, \{4, Y\} \rightarrow \{4, X \cup Y \cup \{b - a\}\} \)

\( V \) \( \{5, X\} \rightarrow \{5, X \cup \{a - b\}\} \)

The possible flows of the program are from command line 1 to either 2 and then 3 and then end, or 4 and then 5 and then end.

To spell out rule \( I \): with regard to the expressions used in the past (1, 2, and 4) don’t forget them and don’t add any new knowledge to them.

Rule \( II \) says that to the knowledge received from the past from 2 and 3, don’t forget them, and add \( b - a \), because this is the expression used in 2.
When activating rule $II$ on $\{2, \emptyset\}, \{3, \emptyset\}$ we receive:

$$\{2, \emptyset\}, \{3, \emptyset\} \xrightarrow{\text{ruleb}} \{2, \emptyset \cup \emptyset \cup \{b - a\}\} = \{2, \{b - a\}\}$$

Our goal is to deduce what $I(A, P)$ is from the set of rules.

We start with the set of atoms:

$$A = \{1, \emptyset\}, \{2, \emptyset\}, \{3, \emptyset\}, \{4, \emptyset\}, \{5, \emptyset\}$$

We quickly see that $\{5, a - b\}$ and $\{3, a - b\}$ are also in the set.

Reminder: We prove things for inductive sets by proving that if something is true for the atoms, then it is also true for the operations.

**Claim**

If $\{5, X\}$ is in the inductive set, then it is either $\{5, \emptyset\}$ or $\{5, \{a - b\}\}$.

**Proof**

For the atoms this holds true, since $\{5, \emptyset\}$ is in the atoms.

Now, let’s assume that $\{5, X\}$ preserves that $X$ is either $\emptyset$ or $a - b$. We need to show that activating the operations preserves this condition; that is, that what we receive is either $\emptyset$ or $a - b$.

It is obvious that we can only activate operations that start with $\{5$ and then something. The options are:

$$\{5, \{b - a\}\} \rightarrow \{5, \{b - a\} \cup \{b - a\}\} = \{5, \{b - a\}\}$$

$$\{5, \emptyset\} \rightarrow \{5, \emptyset \cup \{b - a\}\} = \{5, \{b - a\}\}$$

We show that activating the operations preserves the claim; therefore the claim is true.

Next, if we start with $\{4, \emptyset\}$ and activate the rules that include $\{4, \ldots\}$, we can receive:

$$\{4, \{b - a\}\} \text{ and } \{4, \{a - b, b - a\}\}$$

From all the options that we receive from all the possible activations, the
most interesting one is this:

\{1, \{a - b, b - a\}\}

From the calculation of \(I(A, P)\), we show that preserving the results of the expressions, \(a - b\) and \(b - a\), is efficient for saving time.

How did we deduce that this is the interesting option?

By selecting the "largest" members:

\{1, \{a - b, b - a\}\} contains \{1, \{a - b\}\} and \{1, \{b - a\}\}

It’s possible to deduce the largest members automatically.

Intuitively, we see that the most interesting place for us is the entrance to the if statement (covered in the next lecture).
3 Lecture - Inductive Sets Continued

In this lecture we will cover these topics:

- Propositional calculus
- Propositional calculus from the inductive set point of view
- Propositional calculus from the point of view of the relationship between programming languages and their meaning

Inductive Sets - Continued

In the previous lecture we discussed the definition of the inductive set: \( I(A, P) \). We discussed the set created by the atoms \( A \) and operations \( P \), where \( A \) denotes atoms and \( P \) denotes operations.

We provided two definitions:

1. An *operational* definition ("we say how to create the set"): Start with atoms \( A \) and activate operations \( P \) repeatedly. This type of definition is a "bottom-up" definition.

2. A *declarative* definition: ("without saying how to create the set, we define it"): The minimal set (in terms of inclusion) that maintains two conditions (this is a "top-down" type definition):
   
   (a) All the atoms \( A \) are in the set.

   (b) Closure over the operations: If there are \( k \) members in the set, \( x_1, x_2, ..., x_k \) are all in the set, and you activate an operation \( p \in P \) on them, then \( p(x_1, x_2, ..., x_k) \) is also in the set.
We show that both definitions are equivalent.

This equivalence will allow us to apply the uses of operational and declarative definitions when defining a programming language.

Another aspect of programming languages refers to denotation semantics (with regard to declarative definitions). Given the definitions, we want to try and use denotation semantics to define the first type of logic that we will use: propositional calculus. First-order logic is briefly covered in the next sections.

### 3.1 Propositional Calculus

Propositional Calculus covers the following items:

**Leads To:** if \( x \) then \( y \)

**Or:** \( x \) or \( y \)

**And:** \( x \) and \( y \)

**Not:** not \( x \)

Where \( x \) and \( y \) are types of sentences. For instance, \( x \) is "It is raining now". The result can either be true or false according to the condition if \( x \) and \( y \) are true.

We distinguish between syntax and semantics and between the rules that create the sentences and the meaning of the sentences.

**Syntax**

\[ ((\neg p \rightarrow q) \lor r) \]

In terms of semantics, the syntax has this meaning: Not \( p \) leads to \( q \) or \( r \),
where $p$, $q$, and $r$ are sentences in the language, and each sentence is either true or false.

So where do we stand?

### 3.2 Induction on the Structure

Assuming atomic assumptions: $p$, $q$, $r$, ...

Example: $p$ is the claim that all the tall boys want to drink beer.

In this case $p$ is a complex sentence.

We marked $p$, $q$, $r$, ... as our atoms $A$, and added the operations $\{\land, \lor, \neg, \to\} = P$. Hence, if $r$ and $t$ are claims, then $(t \to r)$, $(\neg t)$, $(t \lor r)$, and $(t \land r)$ are also claims. $I(A, P)$ is the set of all claims.

### 3.3 Tautology

Tautology is where a sentence is always true. For instance, $((p \to q) \to (\neg q \to \neg p))$

We can use a truth table as shown in Table 2.

<table>
<thead>
<tr>
<th>$p$</th>
<th>$q$</th>
<th>$\neg p$</th>
<th>$\neg q$</th>
<th>$p \to q$</th>
<th>$\neg q \to \neg p$</th>
<th>$((p \to q) \to (\neg q \to \neg p))$</th>
</tr>
</thead>
<tbody>
<tr>
<td>T</td>
<td>T</td>
<td>F</td>
<td>F</td>
<td>T</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>T</td>
<td>F</td>
<td>F</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>F</td>
<td>T</td>
<td>T</td>
<td>F</td>
<td>T</td>
<td>F</td>
<td>T</td>
</tr>
<tr>
<td>F</td>
<td>F</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
</tr>
</tbody>
</table>

**Table 2: Truth Table**

Question

How can we define all the tautologies using an inductive set?

The answer to the question begins with the understanding of the world of
Let’s start with the world of syntax.

For now, we put aside the "meaning", which we will later refer to as semantics.

Our purpose is to create a language that uses "Or", "And", and so on; a language that also allows us to characterize tautologies such that the sentences are always true.

For instance: \( \neg p \lor p \)

We start with the purpose of how to define all the sentences in the language. After defining all the sentences, we will define all the true sentences. See Figure 1:

Figure 1: Sentences that are always true as a subset of all sentences
the definition of time. The idea is that the same tools that we use today are the same tools that we use for all logic types.

**Exercise**

How can we define all the sentences in the language?

An analogy from the computer world would be a program that we can compile, meaning that the syntax is correct and not necessarily that the program does anything logical or effective.

**Answer**

We use the technique of the inductive set:

What are the atoms? The letters in the English alphabet, along with the option to use indexes. For instance:

- \( p, q, r \)
- \( p_1, r_7 \)

If \( X \) and \( Y \) are sentences then these are the operations:

- \( (\neg X) \) is a sentence
- \( (X \land Y) \) is a sentence
- \( (X \lor Y) \) is a sentence
- \( (X \rightarrow Y) \) is a sentence

**Example**

Sentences:

- \( (((\neg p) \rightarrow q) \lor r) \) is a sentence and therefore is in \( I(A, P) \). This is true because \( p, q, \) and \( r \) are atoms.
Therefore, \((-p), ((-p) \rightarrow q), \text{ and } (((-p) \rightarrow q) \lor r)\) are all sentences according to the base definitions.

**Question**

Are the following sentences?

1. \(p \land q\)
2. \((-p \rightarrow q)\)
3. \(-\neg p\)
4. \(- (q \land p)\)

**Answers**

1. is not a sentence since there are no parentheses.

2. is not a sentence since there are no parentheses around the \(-p\).

3. is not a sentence since there are no parentheses around the \(-p\).

4. is not a sentence since there are no closing parentheses.

**Question**

Can we claim that the number of opening and closing parentheses in every valid sentence is the same?

**Claim**

The number of opening “(” and closing parentheses “)” in every valid sentence in I(A, P) is the same.

**Proof**

Using induction:
1. Check for the atoms. For the atoms: p, q, r, p₁ there are no parentheses; we have zero opening and zero closing parentheses, which are equal.

2. Check for the operations. Assume that the number of opening and closing parentheses are equal in X and Y and are equal to n. How many opening and closing parentheses do we have after activating the operations?

   (a) In (¬X), we have n+1 opening and closing parentheses.

   (b) We show the same for the rest of the operations: (X ∧ Y), (X ∨ Y),
   (X → Y) - n+1 opening and n+1 closing parentheses.

Q. Why is use of the technique of proof using induction correct?
A. This type of induction is called: ”induction on the structure”.

Q. How do we use this proof technique?
A. We prove that it is true on the atoms, and that if it is true before activating the operations, it is also true after activating the operations. Therefore, the claim is true.

Q. Why is the use of this technique correct?
A. Later, we may use axiom techniques to prove things, but in this case, we can actually prove our claim.

Assume we have a claim T. Let’s look at the set over which the claim T is true. This set T maintains that all the atoms are in T, A ⊆ T, and it maintains closure over the operations. If X, Y ∈ T and you activate an operation p and the result p(X, Y) is also in T; therefore, T is an inductive set over the atoms A and the operations P.

Q. What can we say about the relationship between T and I(A, P)?
A. I(A, P) is the minimal set so \( I(A, P) \subseteq T \), meaning that for every member in I(A, P), the claim is true.

This proves why the use of the induction technique is valid.

So far, the main purpose of this lecture has been to define sentences in terms of syntax. Now we define the syntax of all valid sentences.

We begin with the definition of the Atoms.

Assume that \( X \), \( Y \) and \( Z \) are sentences. Then, the following are also sentences:

- \((X \rightarrow (Y \rightarrow X))\)
- \(((X \rightarrow (Y \rightarrow Z)) \rightarrow ((X \rightarrow Y) \rightarrow (X \rightarrow Z)))\)
- \(((\neg X) \rightarrow (\neg Y)) \rightarrow (Y \rightarrow X))\)

We term these sentences as axioms. For instance, if \( X \) is \((p \rightarrow q)\) and \( Y \) is \(((\neg p) \vee q)\), then our first atom axiom is \(((p \rightarrow q) \rightarrow (((\neg p) \vee q) \rightarrow (p \rightarrow q)))\).

Actually, we wrote an infinite number of atoms, because each sentence \( X \), \( Y \), and \( Z \) represents an infinite number of sentences/axioms/atoms.

Out of the scope of this discussion, there is also an implementation that uses only the \( \rightarrow \) and \( \neg \), to represent all the sentences.

As for operations - there is only one:

If \( X \) is always true, and \((X \rightarrow Y)\) is always true, then \( Y \) is always true.

This is the only operation, and we will later term it *separation*.

Example

In the example, we want to see an axiom and a list of operations, to visualize the formal system.

\((p \rightarrow ((p \rightarrow p) \rightarrow p))\) is an axiom because
\begin{itemize}
  \item $X$ is $p$
  \item $Y$ is $(p \rightarrow p)$
  \item $X$ and $Y$ are sentences
\end{itemize}

Therefore, this is an axiom of the sort $(X \rightarrow (Y \rightarrow X))$.

$$((p \rightarrow ((p \rightarrow p) \rightarrow p)) \rightarrow ((p \rightarrow (p \rightarrow p)) \rightarrow (p \rightarrow p)))$$

is also an axiom according to the second axiom $(((X \rightarrow (Y \rightarrow Z)) \rightarrow ((X \rightarrow Y) \rightarrow (X \rightarrow Z))))$:

\begin{itemize}
  \item $X$ is $p$
  \item $Y$ is $(p \rightarrow p)$
  \item $Z$ is $p$
\end{itemize}

Now, let’s activate the operation:

If something is true and leads to something leads to something is true, then something leads to something is true.

By activating the operation on the previous two lines:

$$((p \rightarrow (p \rightarrow p)) \rightarrow (p \rightarrow p))$$

The following is also an axiom:

$(p \rightarrow (p \rightarrow p))$ is an axiom according to the first axiom.

$X$ is $p$.

$Y$ is $p$.

In this case, we can activate the separation operation, meaning that we deactivate the axiom again.

From $(p \rightarrow p)$, we defined the claim that $(p \rightarrow p)$ as $\vdash (p \rightarrow p)$
So, why did we “play this weird game?” What were we trying to prove using the separation operation?

3.3.2 Meaning of True Sentences

We use induction to prove the true values of every sentence. We want to define a function. The function takes input sentences, and returns an output of either true or false.

At this stage, we can’t do this since we need more infrastructure.

Instead, let’s define the truth values for sentences as shown in Tables 3 - 6.

Table 3: Or Truth Table

<table>
<thead>
<tr>
<th>X</th>
<th>Y</th>
<th>(X \lor Y)</th>
</tr>
</thead>
<tbody>
<tr>
<td>T</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>T</td>
<td>F</td>
<td>T</td>
</tr>
<tr>
<td>F</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>F</td>
<td>F</td>
<td>F</td>
</tr>
</tbody>
</table>

Table 4: And Truth Table

<table>
<thead>
<tr>
<th>X</th>
<th>Y</th>
<th>(X \land Y)</th>
</tr>
</thead>
<tbody>
<tr>
<td>T</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>T</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>F</td>
<td>T</td>
<td>F</td>
</tr>
<tr>
<td>F</td>
<td>F</td>
<td>F</td>
</tr>
</tbody>
</table>

Table 5: Leads To Truth Table

<table>
<thead>
<tr>
<th>X</th>
<th>Y</th>
<th>(X \rightarrow Y)</th>
</tr>
</thead>
<tbody>
<tr>
<td>T</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>T</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>F</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>F</td>
<td>F</td>
<td>T</td>
</tr>
</tbody>
</table>
Table 6: Not Truth Table

<table>
<thead>
<tr>
<th>X</th>
<th>(¬X)</th>
</tr>
</thead>
<tbody>
<tr>
<td>T</td>
<td>F</td>
</tr>
<tr>
<td>F</td>
<td>T</td>
</tr>
</tbody>
</table>

Next, note that we have things here that we don’t need.

For instance, look at Table 7.

Table 7: New Relation Truth Table

<table>
<thead>
<tr>
<th>X</th>
<th>Y</th>
<th>(¬X)</th>
<th>((¬X) ∨ Y)</th>
</tr>
</thead>
<tbody>
<tr>
<td>T</td>
<td>T</td>
<td>F</td>
<td>T</td>
</tr>
<tr>
<td>T</td>
<td>F</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>F</td>
<td>T</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>F</td>
<td>F</td>
<td>T</td>
<td>T</td>
</tr>
</tbody>
</table>

The table values are equivalent to those of the (X → Y) operation.

((¬X) ∨ Y) ≡ (X → Y)

We can "live" without the →, which we can replace with ∨, ∧, and ¬.

Similarly, we can look for a minimal list of signs that will be enough for all the truth tables.

Claim

Using ∨ (or), ∧ (and), and ¬ (not), we can express any truth table.

For instance, let’s look for the meaning of ⊕ using ∨, ∧, and ¬ (see Table 8).

Table 8: XOR Truth Table

<table>
<thead>
<tr>
<th>X</th>
<th>Y</th>
<th>(X ⊕ Y)</th>
</tr>
</thead>
<tbody>
<tr>
<td>T</td>
<td>T</td>
<td>F</td>
</tr>
<tr>
<td>T</td>
<td>F</td>
<td>T</td>
</tr>
<tr>
<td>F</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>F</td>
<td>F</td>
<td>F</td>
</tr>
</tbody>
</table>
Claim

Using "leads to" → and "Not" ¬, we can denote any other operation.

For now, we do not prove this claim.

Claim

Anything we can deduce from the axiomatic system is always true. In other words, the axiomatic system is a tautology.

We will prove this claim using induction:

For the atoms—the axioms—we need to show that this is true:

\((X \to (Y \to X))\)

For this to be false, \(X\) needs to be true while \((Y \to X)\) is false.

But, if \(X\) is true, we conclude that \(Y \to X\) is true (according to the truth table), and therefore, according to the truth table (see Table 9), \((X \to (Y \to X))\) is true.

<table>
<thead>
<tr>
<th>(A)</th>
<th>(B)</th>
<th>((A \to B))</th>
</tr>
</thead>
<tbody>
<tr>
<td>T</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>T</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>F</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>F</td>
<td>F</td>
<td>T</td>
</tr>
</tbody>
</table>

As for the second axiom:

\(((X \to (Y \to Z)) \to ((X \to Y) \to (X \to Z)))\)

So, \((X \to Y)\) must be true and \((X \to Z)\) must be false.

\((X \to Z)\) must be false denotes that \(X\) is true and \(Z\) is false.

From \(((X \to (Y \to Z))\) is true denotes that \((Y \to Z)\) is true, and therefore,
Y is true.

According to the truth table (Table 9), $(Y \rightarrow Z)$ is false, in contradiction to our assumption that $(Y \rightarrow Z)$ is true.

Another option for proof:

$(Y \rightarrow Z)$ is false denotes that $(X \rightarrow (Y \rightarrow Z))$ is false in contradiction.

Now, let’s look at $((\neg X \rightarrow \neg Y) \rightarrow (Y \rightarrow X))$

Convince yourself that this is always true.

End of proof (for the operation):
If $X$ is always true and $(X \rightarrow Y)$ is always true, denotes that $Y$ is always true.

The only situation where $X$ is true and $(X \rightarrow Y)$ is true in the truth table is where $Y$ is also true.

We introduce a new symbol, $|\equiv X$ which means that $X$ is always true.

So, what did we prove?
$|\equiv X \iff |\equiv X$

3.3.3 Tautologies

So how can we define all the tautologies using an inductive set?

We use these axioms:

- $(\beta \rightarrow (\alpha \rightarrow \beta))$
- $((\beta \rightarrow (\alpha \rightarrow \delta)) \rightarrow ((\beta \rightarrow \alpha) \rightarrow (\beta \rightarrow \delta)))$
- $(((\neg \alpha) \rightarrow (\neg \beta)) \rightarrow (\beta \rightarrow \alpha))$

Where $\alpha, \beta, \delta$ are claims.

There are an infinite number of axioms described here, because each is a
claim that can be something similar to: $\alpha$ can be $\neg p \rightarrow (\neg q \rightarrow \neg p)$.

If $\alpha \rightarrow \beta$ is in the set, and $\alpha$ is in the set, then $\beta$ is also in the set. We term this separation.

We can use the inductive set $A$ of all the axioms and the operation of separation. We mark $\alpha$ is in the inductive set using: $\vdash \alpha$.

Example

$\beta = (p \rightarrow q)$

$\alpha = (\neg p \rightarrow q)$

$\delta = (p \rightarrow q)$

According to axiom 1:

$((p \rightarrow q) \rightarrow ((\neg p \rightarrow q) \rightarrow (p \rightarrow q)))$

According to axiom 2:

$(((p \rightarrow q) \rightarrow ((\neg p \rightarrow q) \rightarrow (p \rightarrow q))) \rightarrow (((p \rightarrow q) \rightarrow (\neg p \rightarrow q)) \rightarrow ((p \rightarrow q) \rightarrow (p \rightarrow q))))$

When we activate separation we get this:

$((p \rightarrow q) \rightarrow (\neg p \rightarrow q)) \rightarrow ((p \rightarrow q) \rightarrow (p \rightarrow q))$

Claim

Assuming $T$ is the set of all tautologies, we claim that $I(A, P) \subseteq T$. An additional claim that we will not prove is that $T = I(A, P)$.

So, we need to prove (using induction, of course) that $I(A, P) \subseteq T$. We begin with the atoms; each atoms is a tautology. We prove it using truth tables. For instance, $(\beta \rightarrow (\alpha \rightarrow \beta))$ as shown in table 10.
Next, we assume that $\alpha \to \beta$ is a tautology, and that $\alpha$ is a tautology. We want to activate separation. We need to show that $\beta$ is also a tautology.

As $\alpha$ is a tautology, therefore $\alpha$ is always T, so we only look at those rows in the truth table where $\alpha$ is always T. Also, we assume that $\alpha \to \beta$ is a tautology and therefore we only look at the row in the truth table where $\beta$ is T. So, $\beta$ is T, which is what we wanted to prove.
4 Lecture - Hierarchical Glossaries Example

So far, we covered the inductive set $I(A, P)$ and characterized it using two options:

- **Bottom-Up** (also known as operative) - from atoms you activate the operations.
- **Top-Down** (also known as declarative) - you verify that the set includes the atoms and is closed over the operations, and then you take the minimal set.

We discussed propositional calculus.

We defined a sentence using induction - We took the letters $p, q, r, \ldots$ - and determined that if $\alpha, \beta, \ldots$ are sentences, then $(\alpha \land \beta)$ is also a sentence, $(\neg \beta)$ is also a sentence, and so on.

We defined a proof system.

We showed that if we can prove a sentence denoted by $\vdash \alpha$, then $\models \alpha$ which means that the sentence is always true.

We used the axioms:

$$(\beta \rightarrow (\alpha \rightarrow \beta))$$

$$(\beta \rightarrow (\alpha \rightarrow \delta)) \rightarrow ((\beta \rightarrow \delta) \rightarrow (\beta \rightarrow \delta)))$$

$$(\neg \alpha \rightarrow (\neg \beta)) \rightarrow (\beta \rightarrow \alpha))$$

Note that all of the axioms are tautologic. That is, they have the same values in truth tables. An example is presented in Table 11.
Table 11: Example of a Tautologic Axiom

<table>
<thead>
<tr>
<th>α</th>
<th>β</th>
<th>(α → β)</th>
<th>β → (α → β)</th>
</tr>
</thead>
<tbody>
<tr>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>T</td>
<td>F</td>
<td>F</td>
<td>T</td>
</tr>
<tr>
<td>F</td>
<td>T</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>F</td>
<td>F</td>
<td>T</td>
<td>T</td>
</tr>
</tbody>
</table>

Operation:

\[ \vdash \alpha \rightarrow \beta \]

\[ \vdash \alpha \]

Next, we activate Separation:

\[ \vdash \beta \]

**Example**

\[ \alpha = p \rightarrow q \]

\[ \beta = \neg p \rightarrow q \]

Using Axiom 1:

\[ ((p \rightarrow q) \rightarrow (((\neg p) \rightarrow q) \rightarrow (p \rightarrow q))) \]

Using Axiom 2:

\[ (((p \rightarrow q) \rightarrow (((\neg p) \rightarrow q) \rightarrow (p \rightarrow q))) \rightarrow (((p \rightarrow q)) \rightarrow ((\neg p) \rightarrow q)) \rightarrow ((p \rightarrow q) \rightarrow ((\neg p) \rightarrow q))) \]

Next, we activate Separation:

\[ (((p \rightarrow q) \rightarrow ((\neg p) \rightarrow q)) \rightarrow (((p \rightarrow q) \rightarrow ((\neg p) \rightarrow q))) \]

The second characteristic is complementary in this sense: if something is true, it can be proved.

\[ \models \alpha \Rightarrow \vdash \alpha \]

Semantics and syntax are the same in propositional calculus (see Table 12).
Table 12: Semantics and Syntax in Propositional Calculus

<table>
<thead>
<tr>
<th>JAVA</th>
<th>The meaning of the program</th>
</tr>
</thead>
<tbody>
<tr>
<td>Syntax</td>
<td>Semantics</td>
</tr>
<tr>
<td>Sentence</td>
<td>Sentences</td>
</tr>
<tr>
<td>Axioms</td>
<td>True-False</td>
</tr>
<tr>
<td>Proof</td>
<td>Tautology</td>
</tr>
<tr>
<td>⊢ (\alpha)</td>
<td>(\models) (\alpha)</td>
</tr>
</tbody>
</table>

\(\vdash \alpha \leftrightarrow \models \alpha\)

Example

One of the main purposes of this course is learning how to define what a software system does.

This is known as specification or defining a software requirement, e.g., "The system will perform the action quickly."

Is this a satisfactory requirement? It is an undefined requirement and can cause the project to fail.

Bad requirements cause projects by contract to fail.

Another problematic requirement: "The system should always be available."

What is "always" and what is "available"?

These are also problematic requirements:

- "The system should respond to user requirements within two thousandths of a second."
- "The system should never lose data."

Our purpose is to create a very distinct definition of system requirements. To do so, we use the Z specification language.

The Z language uses set theory + relational calculus + types of sets (for
example, universe, needed for checking consistency).

The Z also requires we only use members of the same type for the sake of checking consistency. For example, we won’t have a group consisting of \{Subaru, Volvo, University of Haifa\}.

### 4.1 Main Example

In this example we implement a glossary. We would like to enhance the editor with which we write documents during the software development process, so it will support a glossary containing terms that we define while writing design and requirements documents. This includes giving new definitions for existing terms. For instance, "thread" is originally (according to the dictionary) a thin line, but in software engineering, it is a system process with a single stack. Another example is adding acronyms such as CSP (Constraints Satisfaction Programming), and dealing with acronyms that have multiple definitions.

We would like to define a hierarchical glossary, with terms which are relevant for all documents, with an option for overriding and an option for defining document-specific glossary terms.

Terms can be potentially defined as forbidden for use at each abstraction level. For instance, we can forbid the use of the specific term "byte" while defining a network protocol.

We would create a system of glossaries (dictionaries) with specific relationships between the glossaries while preserving consistency between hierarchies of glossaries.
Why do we need this system? And why do we need consistency? Many times, when you look at a series of written documents, each document is preserved at a certain level. At that level, you can use the dictionary of the language in which it was written. But then, when writing a more technically specific level of documentation, the dictionary of the language is acceptable only as a starting point. In addition, we need a glossary of specific terms, containing terms that are used differently than the way they are defined in the "language". New terms may also be added to the glossary. For instance, "CSP" is not in the English language, but it will appear in the specific glossary with the definition "Constraint Satisfaction Problem.”
The term "thread" in the glossary will be defined as a type of execution process; something that has a control flow, but with no heap memory (like a "light-process"), as opposed to the dictionary meaning of "thread" which is a string.
We need to use the most relevant glossary for the term. When defining terms, we have a problem. We don’t want to use words that are too specific—words that will limit the implementation. For instance, while defining a communication protocol, we don’t want to use the term: "byte". Why? Because we want to leave an option to implement the protocol using bytes or words (two bytes) or any other implementation.
Next, we attempt to define the hierarchy of the glossaries in the Z language. This is not instead of the definition in words; it is complementary.
We first need to define some infrastructure:
\[
\text{AlphaBet} = \{a, b, ..., z, A, B, ..., Z\} \quad (\text{the set of letters})
\]
We need the AlphaBet set and punctuation signs to define strings and dic-
tionary/glossary entries.

Punctuation = \{" ", ",", "!", ",?", ...\}

We define a sign:

\text{sign} = \text{AlephBet} \cup \text{Punctuation}

Next, we define a word (term) and a sentence (which will be used later as the definition of the term).

A word in a dictionary (we allow words without a logical meaning):

\text{Words} = \bigcup_{i=1}^{\infty} A_i

Where \( A_i = X_{j=1}^{i} \text{AlphaBet} \)

In this case, \( i \) is the length of the word and \( X \) is the Cartesian Product.

A \textit{Cartesian Product} of order \( i \) is all possible permutations of a string containing \( i \) letters from the \text{AlphaBet} set.

\[
= \{(a_1, a_2, ..., a_i) \mid a_j \in \text{AlphaBet} \leq j \leq i \}
\]

Example 1

For \( i = 3, \bigcup_{i=1}^{3} A_i = \text{AlphaBet} \times \text{AlphaBet} \times \text{AlephBet} = \{aaa, aab, aac, abc, zaa, zab, zac, ..., ZZZ\} \)

\( a_1 a_2 a_3 \in \bigcup_{i=1}^{3} \text{AlphaBet} \) if and only if \( \Leftrightarrow a_1 \in \text{AlphaBet} \land a_2 \in \text{AlphaBet} \land a_3 \in \text{AlphaBet} \)

We further define sentences:

\text{Sentences} = \bigcup_{i=1}^{\infty} P_i

Where \( P_i = X_{j=1}^{i} \text{sign} \)

We define the length of the strings \( \# \):

\( \#W = i \Leftrightarrow W \in A_i \)

\( \#S = i \Leftrightarrow S \in P_i \)

Example 2
#(aaa) = 3
#(abc) = 3

A glossary/dictionary is a set of couples: terms (words) and sentences that make up the definition. For instance: \{("dog", "an animal with four legs"),
("cat", "an animal with four legs that does not bark")\}

We continue by defining the substring function:

\(\text{substring}(W, S) : \text{Words} \times \text{Sentences} \rightarrow \{\text{True}, \text{False}\}\)

\(\forall W \in \text{Words}, \forall S \in \text{Sentence} \)

\(\text{substring}(W, S) = \text{True} \iff \exists i \in \mathbb{N}, \forall j \text{ such that } (#W + i) > j \geq i, W(j-i) = S(j)\)

Meaning:

\(\text{substring}(W, S) : \text{Words} \times \text{Sentences} \rightarrow \{\text{True}, \text{False}\}\)

Which means a function from the Cartesian Product of Words and Sentences to \{True, False\}.

A function is something that creates a connection between each member of the source to a single member in the destination. Our source is the Cartesian Product of words and sentences. Our destination is \{True, False\}.

We define our function as:

\(\forall W \in \text{Words}, \forall S \in \text{Sentence} \)

\(\text{substring}(W, S) = \text{True} \iff \exists i \in \mathbb{N}, \forall j \text{ such that } (#W + i) > j \geq i, W(j-i) = S(j)\)

Where \(\iff\) means if and only if and \(W(j)\) means the \(j^{th}\) character in the word.

Example 3

\(\text{substring}(abc, atzbc) = \text{False}\)
\(\text{substring}(abc, ztabcr) = \text{True}\)
\#W = \#(abc) = 3

We look at j, which is the indexes between i=3 to \#W+i = 3+3 = 6, not including 6.

W(j-i) = S(j)

For j=3, W(0) = S(3) = ”a”
For j=4, W(1) = S(4) = ”b”
For j=5, W(2) = S(5) = ”c”

Example 4:

S = ”I’m going home”
W = ”home”

Starting with index zero

S(0) = ”I”
i = 10
\#W = 4

\forall i + \# W > j \geq i

So j is between 10 (inclusive) and smaller than 14 (up to 13 inclusive).

S(10) = W(0) = ”h”
S(11) = W(1) = ”o”
S(12) = W(2) = ”m”

and S(13) = W(3) = ”e”

Therefore, the word ”home” is a substring of the sentence ”I’m going home”.

Observation\Motivation

One interesting aspect in the process of software development, is that if you are able to choose a correct and consistent set of terms, your requirements and definitions are much more accurate and productive.
4.2 Hierarchy of Dictionaries

Now we define the hierarchy of dictionaries. k Dictionaries (Glossaries):

\[ D_1, \ldots, D_k \in \mathcal{P}(\text{Words \times Sentences}) \]

Each \( D_i \), \( i = 1, 2, \ldots, k \) is a dictionary (Glossary) where X is the Cartesian Product, and \( \mathcal{P} \) is the Power Set; the whole set of subsets.

Each glossary, \( D_i \), in the hierarchy preserves a relationship between its words and their meanings (as a pair \((W, S)\) of words and their meanings).

Example 1

\[ D_1 = \{ ("dog", "something that barks") , ("cat", "something my wife keeps at home") \} \]

First rule:

\[ \forall i, j \in \{1, \ldots, k\} \text{ and } \forall (W_1, S_1) \in D_i \text{ and } \forall (W_2, S_2) \in D_j \text{ such that } j > i \]

it preserves that \( W_2 \neq W_1 \land \neg \text{substring}(W_2, S_1) \) (where k is the same k as above).

Explanation in words: in glossary i, you can’t use a word either as a Word or as a Sentence.

This means that in the sentence in \( S_1 \), we can’t use words that we define later in \( D_j \), \( j > i \).

Note that the following is a tautology:

\[ \models (\neg a \land \neg b) \leftrightarrow \neg (a \lor b) \]

You can convince yourself using the truth table (Table 13).
Table 13: If And Only If Truth Table

<table>
<thead>
<tr>
<th>p</th>
<th>q</th>
<th>( p \leftrightarrow q )</th>
</tr>
</thead>
<tbody>
<tr>
<td>t</td>
<td>t</td>
<td>t</td>
</tr>
<tr>
<td>t</td>
<td>f</td>
<td>f</td>
</tr>
<tr>
<td>f</td>
<td>t</td>
<td>f</td>
</tr>
<tr>
<td>f</td>
<td>f</td>
<td>t</td>
</tr>
</tbody>
</table>

In the next lecture we will add rules to the Hierarchy of Glossaries.
5  Lecture - Hierarchical Glossaries Example, Continued

Lecture Outline

• Hierarchy of Rules for Glossary Design
• Sets
• Type
• Cartesian Product Sets
• Functions
• Bijective Function
• Equivalence Classes
• Fixed Point Theorem

In the previous section, we discussed Inductive Sets, and showed some uses, for instance, proving using the Inductive Sets.

As the purpose of the course is to exemplify the connection between mathematical logics and software testing, we turn to the phases of software development and show how mathematical logics can be used to improve them.

On such phase is the requirements phase. When defining system requirements, we define what a system is supposed to do—not how. During the design phase, we define how the system is supposed to do it.

In this lecture, we focus on the requirements phase.
So, how do we define the problem we are trying to solve?

People either do not write software correctly, or do not write software that meets the actual needs. Most software projects are thrown away, and never used. Why is that? Because of misunderstandings during the whole process. The issue is the price. If we implement something different from the requirements, the price of finding that it does not fit the customers’ needs is very high. Finding that it does not fit during the requirements stage would cost much less.

In summary, it is hard to write requirements correctly, and writing incorrect requirements causes very expensive problems (as opposed to problems with design or implementation). The reason is that if only the implementation is incorrect, but the design was correct, you don’t have to go back more than a single phase. Therefore, in this course, we provide tools for defining what a system is supposed to do.

Examples of bad requirements:

"The system should respond fast.” What is fast? What defines fast?

"The system should always be available.” What is always? What is available? Available from where?

"The system should never lose data” causes a cost for redundancy which is unacceptable to the customer, so instead, you negotiate the trade-off.

We try to introduce you to the use of a new language, called Z, to define functional software requirements.

The functional requirements are those that are not the RAS (reliability, availability and serviceability) requirements.

The Z specification language uses first-order logics combined with the addi-
tional relationships: "for-each" and "exists”. Relationships include functions.

5.1 Hierarchy of Rules for Glossary Design

Another rule:
\[ \forall (W_1, S_1) \in D_i \]
\[ \forall (W_2, S_2) \in D_i \]
\[ W_1 \neq W_2 \land S_1 \neq S_2 \]

We define:
result = \{ OK, OKDup, FailedAbstract, FailedDup \}
i, j, r = 1, ..., k

We further define:
\[ \text{complete}_{D_i} = \{ (W, S) - \text{such that } \exists 1 \leq j \leq i \text{ such that } (W, S) \in D_j \land \neg (\exists r > j | (W, S) \in D_r) \} \]

Where \((W, S) \in \mathcal{P}(\text{Words X Sentences})\)

These are the terms that are allowed for use in the i’th level.

Operation

Adding a word and a definition: \((W?, S?) \in \mathcal{P}(\text{Words X Sentences})\) to \(D_i\)

\(D_i\) is termed **Schema** (in Z language).

The ”?” in Z Language denotes ”input”.

\(i?: \mathbb{N}, (W?, S?): \text{Words X Sentences}, \text{res!}: \text{Result} \)

We want to return a result from the Result type that we defined before.

(! is the output of the operation in Z)

\[ (\exists k \geq j > i?, \land \exists (W1, S1) \in D_j \text{ such that } W1 = W? \lor \text{substring}(W1, S1)) \land \]
which means that the result is:
res! = FailedAbstract
\forall
(\exists (W1, S1) \in D_i \text{ such that } W1 = W?)
\land
res! = FailedDup
\forall
(\exists 1 =< j =< i (W1, S1) \in D_j \text{ and } W1 = W?)
\land
D'_i - the situation of the dictionary after the operation
D'_i = D_i \cup \{(W?, S?)\}
res! = OKDup
OKDup = we can define again a term that we defined in a previous level
\forall
D'_i = D_i \cup \{(W?, S?)\}
\land
res! = OK

Next, we prove by induction on the operation that if we start with the dictionary \( D_i = \emptyset \) (the atom) and define the operation as the schema, then we can use the inductive set proof method to prove that all resulting dictionaries preserve the same rules. Intuitively, this is clear, since our only operation adds—without breaking the rule.

Hierarchical Glossaries Example, Continued
We denote the hierarchy of dictionaries as \( D_1, D_2, \ldots, D_k \). Each Dictionary \( D_i \) is of type \( P(\text{Words} \times \text{Sentences}) \) where \( 1 \leq i \leq k \). In words, each dictio-
nary is a set of pairs of words (dictionary terms) and sentences (the term’s meaning). Another way of saying it is that \( D_i : \mathcal{P}(\text{Words} \times \text{Sentences}) \). \( X \rightarrow Y \) denotes that \( X \) is “of type” \( Y \).

Let’s see how it works:

A sample of a set in \( D_i \) (a semantic example):
\[
\{ (\text{cat}, ”an animal which I have at home”), (\text{dog}, ”an animal that I don’t have at home”), (\text{snake}, ”an animal I found yesterday in my yard”) \}
\]

Another example: \( \{ (”abc”, ”e”), (”abc”, ”abc”), (”abc”, ”d”) \} \) is a dictionary since the syntax is legal (even though the semantics might not make much sense).

Another example: \( \{ (” abc”, ” abc def ”), (”abc”, ”d”) \} \) is not a dictionary since the first term includes a space, which is illegal in the syntax we defined.

### 5.2 Introducing the Z Specification Language

The Z language includes:

- First order logic (propositional calculus and ordinal calculus): \( \land, \lor, \rightarrow, \neg \)
- Relations and functions: \( \forall, \exists \)
- Each set has to be of a type; we do not allow all the possible sets. For instance, we do not want to have sets of the type: \( \{\text{Tomato, Volvo, Roof}\} \) since they are members of different types.

The type is a tool that enables testing the consistency of the definition of a specification.

\(^{3}\)which is also saying that \( X \) is a member of \( Y \).
∀ i : N, 1 ≤ i, ∧ i ≤ k, ∀ (W, S) ∈ D_i, W \neq S

This means that the term and definition will be different.

The error in the definition is that we cannot compare things of different types: W \neq S, Words and Sentences are not of the same type. So—an open question—how do we overcome this obstacle? It’s not easy! It requires more knowledge of the Z language.

This definition is actually an ”invariant”; that is, it is true throughout the whole lifecycle of the system.

∀ i : N, 1 ≤ i, ∧ i ≤ k, ∀ (W1, S1) ∈ D_i, ∀ (W2, S2) ∈ D_i, W1 = W2 → S1 = S2

This means that each term has only a single definition.

That is, for every two pairs of words and sentences in the dictionary, if the word is the same word, the definition has to be the same too. This is also an invariant.

∀ w : Words, ∀ s : Sentences

substring : Words X Sentences → {T, F}

substring(W, S) = T ⇔ ∃ i ∈ N, ∀ j such that \#W+i > j ≥ i, W(j - i) = S(j)

substring is a function from pairs of words and sentences to a set of True, False.

What is a function? A function associates each value from the source (in our case, Words X Sentences) with a single member in the target (True or False).

\#W is the length of the string W.

Example

Prove that ’ab’ is a substring of ’drabf’.
#’drabf’ = 5

We take i to be 2. j is the values from i = 2, and to the length of ‘ab’ #’ab’ = 2, 2+2=4; thus j is the values from 2 to 4.

Future Enhancements

In the next lecture (see section 6) we will attempt to define operations such as addition and removal. For that, we need to better improve the definition of our world and notations.

Our world consists of a series of documents. Each document is specified with a specific dictionary:

\[ D1 \leftrightarrow SP1 \text{ (Specification)} \]
\[ D2 \leftrightarrow SP2 \text{ (Specification)} \]
\[ D3 \leftrightarrow SP3 \text{ (Specification)} \]

and so on...

Now, we’d like to define a new document, SP5, with a dictionary D5. The question is, what terms am I allowed to use and not allowed to use in SP5?

We decide that we are not allowed to use any terms defined after 5, and we are allowed to use the latest added definition of any term defined before 5.

We define CD_1, CD_2, ... CD_k : P(\text{Words X Sentences}) where each CD_i (current dictionary) is the words allowed for use in the specification SP_i.

\[ \forall i, 1 \leq i \leq k, (w, s) \in CD_i, \text{max}(w) \leq i \]

This means that (w, s)—a word and its definition—is in the current dictionary, if the maximum appearance of the term is before i, where max(w) is the maximum place in which w appears.

\[ \text{max : Words} \rightarrow \mathbb{N} \]

\[ \text{max}(w) = i \iff \]
(∃ s ∈ Sentences ∧ j ≤ i, (w, s) ∈ D_j) ∧ (¬ ∃ s ∈ Sentences ∧ j > i, (w, s) ∈ D_j)

5.3 Hierarchical Dictionaries Example, Continued

We define the term "thread". In D_1 we define "thread" as a string. In D_4 we define "thread" as a light computer process. In any specification using dictionaries greater than or equal to level 4, "thread" means the "light computer process" definition.

CD_i—where i = 1, 2, ..., k—is the set of words that you can use at level i.

Note that to reach a specific definition, we can use examples, drawings, and diagrams, and write "in words" the definition before we write it formally.

Let’s try and define CD_i formally. CD_i is a set of Words and their Definition.

The use of the set usually indicates the use of P. Words and their definitions are indicated by Words X Sentences.

So CD_i is of type: P(Words X Sentences), which means a set of pairs, and each pair is an instance of Words and a defining instance of Sentences.

Example

CD_i = {("thread", "light weight computer process"), ("CSP", "Constraint Satisfaction Problem")}

We defined that at each level we’d like to use only terms that appeared before the level (and not just after the level), and in that case, we’d like to use the most up-to-date relevant definition.

(w, s) ∈ CD_i ⇔ ∃ j ≤ i, (w, s) ∈ D_j. In words, this means that a pair of word and sentence, is legal in dictionary of level i CD_i if and only if there
exists a \( j \), smaller or equal to \( i \), where the pair of word and sentence is in the dictionary of level \( j \).

Next, we would like to add the definition of using the latest, most currently available version: \( (w, s) \in CD_i \iff j \leq i, (w, s) \in D_j \land \forall l, k : \mathbb{N}, (l \leq i) \land (k \leq i) \land l < k \land (w, s_1) \in D_l \land (w, s_2) \in D_k \rightarrow (w, s_1) \not\in CD_i \)

In words, if \( k \) and \( l \) are dictionaries before \( i \), and \( l \) is before \( k \), and \( (w, s_1) \) is in dictionary \( l \), and \( (w, s_2) \) is in dictionary \( k \), then \( (w, s_1) \) can't be in the list of allowed words in level \( i \).

Note, the \( \mathcal{Z} \) specification language is a declarative and not operative language, so we define formal definitions without defining how we can operatively reach it.

Example

We would like to reach a simpler definition, possibly using the "max" function.

5.4 Defining Adding a Word and its Definition to a Dictionary

We formally define the addition of a term and its definition to a dictionary \( D_i \). In the \( \mathcal{Z} \) specification language, we use "?" to denote input. We would like to add \( (w?, s?) : \) Words X Sentences to dictionary \( D_i \). We define an output, or result, which is the updated dictionary (a side-effect change), and a response, which is the result of the operation. We mark the resultant output using "!".

We begin with defining when we would like to allow the addition of a pair
(w?, s?) to dictionary D_i.

1. If (w?, s_1) is already in Dictionary D_i, what do we want to do? We would like to tell the user that we are overriding the definition (adding the new definition to replace the old one). In the case S_1 is different than s?, so we mark the response okDup.

2. Same as in #1 but for the case that s_1 = s?, we mark the response as okCompleteDup.

We gradually define the Result set with all the possible responses. Result = okDup | okCompleteDup.

We gradually define the operation formally, defining a schema in Z as follows:

\[
\begin{align*}
&\exists \ s_1 \in \text{Sentences}, \\
&(w?, s_1) \in D_i \Rightarrow \\
&D_i' = (D_i \cup \{(w?, s_1)\}) \setminus \{(w?, s_1)\} \\
&s_1 \neq s? \Rightarrow \text{res!} = \text{okDup} \\
&s_1 = s? \Rightarrow \text{res!} = \text{okCompleteDup}
\end{align*}
\]

Above the line, we define the input and output. Below the line, we define the different cases.

The first case says that if a sentence for the word we would like to add already exists in the dictionary, the dictionary after the operation is the dictionary with the new sentence, and we remove the previous definition.

Exercise

Prove using induction that each word that appears in dictionary D_i appears
only once.

Hint: The atoms $\emptyset = D_1 = D_2 = ... = D_k$, and the operations P add a word and a sentence $(w?, s?)$ to dictionary $D_i$.

## 5.5 Dictionary Example

Assume we have dictionaries $D_1$, $D_2$, ..., $D_k$ as before, and the term byte only appears in level $D_6$, but in $D_2$ ”byte” is used in the definition of term $w$. Is this acceptable? Do we want to allow defining terms using disallowed terms? We would like to prevent the option of adding a defining sentence for a term that uses terms that are disallowed in the dictionary level to which the term is to be added.

We add another case (denoted by $\lor$) to our previous definition: $\exists w1 : \text{Words, substring}(w1, s?) \land w1 \notin CD_i \Rightarrow \text{res!} = \text{Fail}$

which means a word $w1$ exists, in the defining sentence $s?$ which we would like to add to dictionary $D_i$, but $w1$ is disallowed in $CD_i$, so the operation should fail. In the case of a failed result, we do not change the dictionary.

We update the Result set to be:

Result = okDup $|$ okCompleteDup $|$ Fail

## 5.6 Referred Sets

When defining a formal specification, to which sets are we allowed not allowed to refer?

We will use induction for defining the sets we are allowed to use, with operations Cartesian Product and Power Set. We define these as the basic types.
We define the Atoms: \([A_1], [A_2], \ldots, [A_k]\)

For instance, \([\text{Result}] = [\text{okDup} | \text{okCompleteDup} | \text{Fail}]\)

and we define the operations: \(P\) and \(X\).

Therefore, we conclude that \(P(A_2 X A_3)\) is also a type.

Note that all the sets to which we are allowed to refer must be subsets of allowed types.

Exercise

Assuming the atoms: \([\{1, 2, 3\}], \{[4, 5]\}\) Are the following allowed types or not?

1. \(\{3, 4\}\) is disallowed since we can’t reach this set from activating Power Set or Cartesian Product on the atoms. We can’t refer to this set, since it is not a subset of any existing type.

2. \(\{\{3\}, \{4\}\}\) is not a type because we won’t reach this set from the basic types (atoms) and the operations. If it is a set we are allowed to refer to, we need to show a derivation order of operations that results in a set containing this set. We can’t do that without an additional parenthesis: \(\{(\{3\}, \{4\})\}\).

3. \(\{1, 2\}\) is not a type, but we are allowed to refer to it since it is a subset of a type.

5.7 **Z Specification Language Basic Signs**

We started talking about some of the basic signs of the Z specification language. Z is a very rich language, with many additional signs. The use of
many signs has not been proven to improve readability or quality. 
We discussed the restrictions of the \( \mathcal{Z} \) language that cause us to be more spe-
cific, and find problems in the specification and software. We use \( \mathcal{Z} \) to define 
what a software program is supposed to do (and not how). We discussed the 
use of "types". 
Why aren’t requirements a part of software development? Because require-
ments aren’t just for software development. Defining functions is also needed 
in other domains. 
The \( \mathcal{Z} \) specification language includes first order logics, group theory, and 
types. 
First order logics includes, for instance, \( \lor, \land, \neg, \rightarrow, \forall, \exists \), signs for functions 
and relations, functions \( (f, g) \), relationships \( (R, S, T) \), and so on. 
Group theory includes \( \in, \subseteq, \cup, \cap, P, X \) (Cartesian Product), and so on. 
The term "Types" includes only the certain types that we are allowed to 
refer to (all the members need to be of the same "type"). 
Our purpose in this lecture for each specification is to define the types to 
which we are allowed to refer. 
We define the types we are allowed to refer to using induction. Types are 
defined as the groups that we are allowed to refer to in each specification. 
We are allowed to refer to any group that is a type or that is a subset of a 
type. 
Note: The requirement makes a distinction on the operations that you can 
perform. For instance, if you take a \( \cup \) on two groups that you are allowed 
to discuss, you are not necessarily allowed to refer to the union of the two 
groups.
To define all the types we are allowed to refer to for a specific specification, we use an inductive set.

We use a set of "basic types" that we are allowed to refer to—even though we cannot define them—as the atoms of the inductive set. We mark them as \([A_1], [A_2], ..., [A_k]\). In each system, there will be basic types.

**Exercise**

What are the basic types in the definition of the glossaries?

Union (\(\cup\)) between types cannot be used to define a new type. So we redefine the union operation:

\[
A \cup B = \{ x \mid x \in A \lor x \in B \} \iff \exists R \in I(A, P), A \subseteq R \land \exists S \in I(A, P), B \subseteq S \land R = S
\]

That is, \(A\) and \(B\) are subsets of the same type.

Next, we define a relation:

\(R\) is a relation if \(R \in P(A \times B)\), where \(A\) and \(B\) are types. In other words, \(R\) is a relation if it is a set of couples: \(a\) from \(A\), and \(b\) from \(B\), and \(A\) and \(B\) are types.

**Example**

Our basic types are \([C = \{a, b, c\}], [D = \{0, 1\}]\).

\(C \times D = \{(a, 0), (a, 1), (b, 0), (b, 1), (c, 0), (c, 1)\}\) is a type.

The following is a relation: \(\{(a, 0), (a, 0), (a, 0), 1)\}\) because it is a set which is a member of \((C \times D) \times D\).

Function is a specific example of the relation \(f: P(A \times B)\) where \(f\) is a relation and for each member of the source \((A)\) there is a single member in \((B)\), which means there is a single couple in \(f\).

80
6 Lecture - Formal Specification Languages

Lecture Outline

- Rules for Design of Glossaries Hierarchy, continued
- Sets
- Type
- Cartesian Product Sets
- Functions
- Bijective Function
- Equivalence Classes
- Fixed Point Theorem

In the previous lecture we discussed specifications as definitions of what a system is *supposed to do*. We use mathematical logic and set theory. We ended the lecture with a sample specification and found that it was incorrect (to be revisited in a later lecture).

In this lecture, our aim is to define the formal language used for specifications. While defining, we try to provide examples of snippets of specifications. Our discussion is limited to sets whose members are only of the same type. For example: \{ Subaru, USA \} is not a set we refer to, as Subaru is a type of car and USA is a country, so they are not of the same type.

We present, in parallel, the formal language syntax and its associated interpretation (semantics).
6.1 Formal Specification Language - Definition

Syntax

Our purpose is to define the sets that we are allowed to refer to. We define the following symbols by induction:

1. Assuming k symbols: \([A_1], ..., [A_k]\).

2. If Y and Z are allowed symbols, then Y \(\times\) Z and \(\mathcal{P}(Y)\) are also allowed symbols.

3. We define \(I(A, P)\) as the set of all possible allowed symbols.

The intended meaning of these symbols is the possible set types.

Interpretation

To create a specification, we map \([A_1], ..., [A_k]\) to k sets \(A_1, ..., A_k\), and define the inductive set \(I(A, P)\) as the set of possible types where \(\times\) is interpreted as the Cartesian Product operation and \(\mathcal{P}\) as the Power Set operation.

Example

\(A_1 = \{\text{Subaru, Volvo, Fiat}\}\)

\(A_2 = \{\text{Slow, Fast, Medium}\}\)

\(A_1 \times A_2 = \{(\text{Subaru, Fast}), (\text{Subaru, Slow}), (\text{Subaru Medium}), ..., (\text{Fiat, Medium})\}\)

\(\mathcal{P}(A_1 \times A_2) = \emptyset, \{(\text{Subaru, Fast}), (\text{Fiat, Fast})\}... \}\)

| \(A_1 \times A_2\) | = 9
| \(\mathcal{P}(A_1 \times A_2)\) | = \(2^9\)

Members of \(\mathcal{P}(A_1) \times \mathcal{P}(A_2)\) are pairs of subsets of \(A_1\) and subsets of \(A_2\).

A sample member: \((\{\text{Subaru, Volvo}\}, \{\text{Fast, Slow}\}) \in \mathcal{P}(A_1) \times \mathcal{P}(A_2)\).
We limit our discussion to (the specification can only refer to) subsets of sets that are types, or members of sets that are types.

Syntax

- \( \in \) Interpreted as a member of
- \( \neg, \wedge, \vee, \rightarrow \) Interpreted as in propositional logic
- \( \subseteq \) Intended interpretation is subset

\[
F \subseteq G \iff (x \in F \Rightarrow x \in G) \land \exists H \in I(A, P) : G \in H
\]

The intended meaning is that \( F \) is a subset of \( G \) if and only if for every \( x \) which is a member of \( F \), \( x \) is also a member of \( G \). In "regular" set theory, this definition is sufficient. In our scope we add to this definition an additional requirement that \( G \) be a member of some set \( H \in I(A, P) \). \( H \) must be a type we are allowed to refer to.

Semantics

Same \( A_1 \) and \( A_2 \) as before.

Now we check if one set is a subset of the other:

Is the following true?

\[
\{\text{Subaru, Fast}\} \subseteq \{\text{Subaru, Volvo, Fast}\}
\]

No! Why? Because the members of the set do not belong to a single type. In "regular" set theory, this would have been True, since \( x \in A \Rightarrow x \in B \), but in our language, this is not True since both sets do not belong to any single type.

\[
\{\text{Subaru, Volvo, Fast}\} \notin I(A, P)
\]

Claim
Any member of $I(A, P)$ that is not $A_1$ or $A_2$ has a size of at least three members. Note that each member of $I(A, P)$ is a set.

Proof by induction

For the Atoms, the statement is true. Assuming $Y$ and $Z$ are members of $I(A, P)$ and $|Y| > 3$ and $|Z| > 3$, then $|P(Y)| > |Y| > 3$, $|P(Z)| > |Z| > 3$.

**What is missing in the definition?**

Example of building $I(A, P)$ on different sets:

$A_1 = \{a, b\}$

$A_2 = \{c, d\}$

$P(A_1) = \{\emptyset, \{a\}, \{b\}, \{a, b\}\}$

$P(A_2) = \{\emptyset, \{c\}, \{d\}, \{c, d\}\}$

$A_1 \times A_2 = \{(a, c), (a, d), (b, c), (b, d)\}$ $A_1 \times P(A_1) = \{(a, \emptyset), (a, \{a\}), (a, \{b\}), (a, \{a, b\}), (b, \emptyset), (b, \{a\}), (b, \{b\}), (b, \{a, b\})\}$

$P(\emptyset) = \{\emptyset\}$, $P(\{\emptyset\}) = \{\emptyset\}$...

We continue with the syntax. We defined a member, types, basic operations, and so on. What’s next? Let’s define equality:

$A = B \iff A \subseteq B \land B \subseteq A \land \exists C : I(A, P), (B \in C) \land (A \in C)]$

$\exists C :$ we will need to denote of which type is $C$.

Whenever we refer to $\exists x$ and $\forall x$, we need to define the set over which $x$ is known as a valid set.

$\forall x \in A$ and $\exists x \in A$ are valid $\iff A \subseteq C$

We add the definition of a relation, $R$:

$R$ is defined as $R \subseteq Y \times Z$ where $Y$ and $Z$ are types.

In other words, $R$ is a set of pairs.
Summary
So far, in our syntax, we allow the use of:

- Sets: \([A_1], ..., [A_k]\)
- \(I(A, P)\) - types
- \(\in\)
- \(\subseteq\)
- \(=\)
- \(\exists\)
- \(\forall\)
- \(R\) - relation

Relative definition
\([A_1], ..., [A_k]\)
\(I(A, P)\) - types

And for
\(\in\)
\(\subseteq\)
\(=\)
\(\exists\)
\(\forall\)
\(R\) - relation

We add the restriction of types.
6.2 Ruling Function from the Talmud Revisited

The ruling function from the Talmud is described in section 1.4.

We continue the definition with a justification for referring to the types:

\[ P \subseteq (A \times A) \times \text{Payment} \]

\[ \text{Payment} \in P((A \times A) \times \text{Payment}) \]

So our discussion about this relation is valid. If, for instance, we define a function \( f \) from \( A \times A \) to Outcomes:

\[ f : A \times A \rightarrow \text{Outcomes} \]

\[ f = \{((\text{don’t know}, \text{don’t know}), 0), ((\text{don’t know}, A), 500), \ldots, ((B, B), 1000)\} \]

Let’s verify if \( f \) is a type.

\[ \text{Outcomes} = \{0, 500, 1000\} \]

is a type, since it is a basic type.

We had \[ \text{Costs} = \{500, 1500, 2000\} \], which we do not use, so we can remove it. On the other hand, Costs describes the cost of the bulls, so we might need it later.

We notice that \( \text{Costs} \cap \text{Outcomes} \neq \emptyset \), which isn’t ”elegant” since it is rare for intersections between basic types to be not empty.

We can solve this, for instance, by referring to \( \mathbb{N} \) as the basic type of the natural numbers, which makes Outcomes and Costs subsets of \( \mathbb{N} \), so we are allowed to discuss Outcomes and Costs:

\[ \text{Outcomes} \subseteq \mathbb{N}, \text{Costs} \subseteq \mathbb{N}. \]

If we decide that Outcomes is a basic type, what is the type of function \( f \)?

Since \( f \subseteq (A \times A) \times \text{Outcomes} \),

\( f \) is of type
f : \( P((A \times A) \times \text{Outcomes}) \)
Since \( A \) and \( \text{Outcomes} \) are basic types, and we are allowed to use the Cartesian Product and Power Set operations, \( (A \times A) \) is a type, \( (A \times A) \times \text{Outcomes} \) is a type, and \( P((A \times A) \times \text{Outcomes}) \) is a type.

We use the fact that if \( C \in D \) and \( D \) is a type, then \( C : D \).

In other words, if \( C \subseteq E \Rightarrow C \in P(E) \).

If we decide that \( \text{Outcomes} \) is not a basic type, and we use \( \mathbb{N} \) as a basic type:

\[ f : P((A \times A) \times \mathbb{N}) \]

**Exercise**
What is the type of function \( f \) if we were to define both \( \text{Costs} \) and \( \text{Outcomes} \) as basic types?

We leave this as an exercise for the reader.

**Summary**

- Set Theory
- Logics with quantifiers
- Logics with Types, defining a language that helps define what software is supposed to do
- Example from the Talmud (with Bulls), modeled using a function
7 Lecture - Communication Group Example

We want to model which computers / end-points are in touch with which others, and can transfer data. We term the example: "Group Communication".

We aim to define a group of computers that all share the same set of messages, although each computer can be either in an up or down state, and the communication between them can fail.

Potentially, because we do not restrict the network, each computer can communicate with all other computers.

We begin by describing the state of each computer. We term the computers Nodes. Each Node can be in one of two states: \{up, down\}.

A node in the down state cannot communicate with other nodes, and other nodes cannot communicate with that node. A node in the up state can potentially communicate with other nodes that are up.

We define the Nodes as a basic type: \([\text{Nodes}]\). We start with two groups: \([\text{Nodes}], [\text{States} = \{\text{up, down}\}]\). The symbol "/" marks Basic Types.

We allow referring to the basic types and the inductive set induced by the basic types and the operations: Cartesian Product, marked by \(X\), and Power Set, marked by \(P\).

We define \([\text{Nodes}]\) as the set of computers in the network. We define \([\text{States}]\) as an attribute state of the computer: \(up\), i.e., which is up and running; or \(down\), i.e., not running.

The purpose of the formal definition is to define a situation where only machines that are up can be connected in a Communication Group, and ma-
chines that are down cannot be in the Communication Group (which also
makes sense intuitively).

We aim to model changes in states in the Communication Group (machines
changing status from up to down, and vice versa).

We mark the [Nodes] basic type as N, and the [States] basic type as S.
The next thing we need is a name for the computers. We agree that there is
a unique number indicating each computer. So, we use the N basic type to
indicate the nodes: [Nodes : N].

We reduce the number of Nodes to a finite number of nodes, so Nodes ⊆ N
⇒ Nodes ∈ P(N). P(N) is a type because N is a basic type, and so N is a
type and therefore P(N) is a type.

We claim that: f : Nodes → States. What does this mean? f is the function
that maps a state to each node. f ⊆ Nodes X States, States is a type, Nodes
⊆ N and N is a type. A general function f: A → B is a ⊆ A X B.

Therefore, f ∈ P(N X States), and P(N X States) is a type.

Next, we define Connected as a set of all the computers (Nodes) connected
to one another.

\[
\begin{align*}
\text{Connected} & : P((NXS)X(NXS)) \\
\forall (n_1, s_1), (n_2, s_2) : NXS \\
((n_1, s_1), (n_2, s_2)) & \in \text{Connected} \rightarrow \\
s_1 = s_2 & = \text{up}
\end{align*}
\]

Example

\{((1, up), (4, up)), ((4, up), (3, down))\} is not Connected because node 3
is down.
Next, we define a function from Nodes to their States:

\[
\text{state} : \text{Nodes} \rightarrow \text{States}.
\]

In this case the function is a \textit{complete function} because for each member \(n\) of the source (Nodes in our case) there is a single value in the target (States), which is \(\text{state}(n)\).

\[
\begin{align*}
\text{state} & : \mathcal{P}(\text{Nodes} \times \text{States}) \\
(\forall \, n \in \text{Nodes}, \exists \, s \in \text{States} \\
(n, s) \in \text{state}) & \land \\
(\forall \, n : \mathbb{N}, \forall \, s_1 : S, \forall \, s_2 : S, \\
(n, s_1) \in \text{state} \land (n, s_2) \in \text{state} & \rightarrow s_1 = s_2)
\end{align*}
\]

To verify that \(\text{state}\), which is of type \(\mathcal{P}(\text{Nodes} \times \text{States})\), is something to which we can refer, we need to verify that \(\mathcal{P}(\text{Nodes} \times \text{States})\) is a \textit{type}.

[Nodes] is a basic type and therefore a type. [States] is also a basic type and therefore a type. Nodes \(\times\) States is also a type since the Cartesian Product is an allowed operation between types. \(\mathcal{P}(\text{Nodes} \times \text{States})\) is also a type since the Power Set is an allowed operation on types.

We define Connected as a requirement that is an invariant.

We attempt to define the operations in the system, starting with the "easier" action. But first we need to define the initial state of the system.

In the initial state, we would like the invariant to be correct and to define the state of each node.

For instance, if the Connected set is empty (\(\emptyset\)), it preserves the invariant. Another option is that all the states of the nodes are up (no matter what that set Connected contains).
Initial State

Next, we aim to define the Initial State:

\[
\begin{align*}
\text{state} : N &\rightarrow S \\
\text{connected} &= \emptyset \land \\
\forall (n, s) \in \text{state}, s &= \text{down}
\end{align*}
\]

This schema means that the initial state is that all computers (nodes) are down.

We next aim to model the operations and changes that can occur in the system, and we start with a fail.

\[
\begin{align*}
\text{Fail} : ((n_1, s_1), (n_2, s_2)) \rightarrow P((\text{NXS}) \times (\text{NXS})) \\
((n_1, s_1), (n_2, s_2)) &\in \text{connected} \land \\
\text{connected}' &= \text{connected} - \{(n_1, s_1), (n_2, s_2)\}
\end{align*}
\]

Example

We assume the set Connected is \{((1, up), (2, up))\}.

Our input is: ((1, up), (2, up)).

The predicate calculation is ((1, up), (2, up)) \in Connected is true and

\[
\text{Connected'} = \text{Connected} - \{((1, up), (2, up))\} = \{((1, up), (2, up))\} - \{((1, up), (2, up))\} = \emptyset.
\]

If the input was illegal; for example, our input is: ((1, up), (2, down)), then

\[
(((1, up), (2, down)) \in \text{Connected}) \text{ is false, and therefore Connected'} \text{ remains unchanged}
\]
\[ \text{Fail Node} \]
\[ n? \in \text{Nodes} \]
\[ \begin{align*}
\text{State}(n?) &= \text{up} \land \\
\text{State}'(n?) &= \text{down} \land \\
\forall n^\sim \in \text{Nodes}, s^\sim \in \text{States}, \\
((n?, s), (n^\sim, s^\sim)) &\in \text{Connected} \rightarrow \\
\text{Connected}' &= \text{Connected} - \{((n?, s), (n^\sim, s^\sim))\} \land \\
((n^\sim, s^\sim), (n?, s)) &\in \text{Connected} \rightarrow \\
\text{Connected}' &= \text{Connected} - \{((n^\sim, s^\sim), (n?, s))\}
\end{align*} \]

Note: To activate the operations: \( \cup \) and \( \cap \) on sets, both arguments (sets) need to be of the same type and therefore when we write \( \text{Connected}' = \text{Connected} - \text{something} \), that something has to be a set.

Example

State = \{ (1, up), (2, up), (3, down), (4, up) \}

Connected = \{ ((4, up), (1, up)), ((1, up), (2, up)) \}

We activate \text{Fail Node} on node \( n? = 1 \):

\[ \text{State}(n?) = \text{State}(1) = \text{up} = \text{True} \]

\[ \text{State}' = \{ (1, \text{down}), (2, \text{up}), (3, \text{down}), (4, \text{up}) \} \]

\[ \text{Connected}' = \text{Connected} - \{ ((4, \text{up}), (1, \text{up})) \} - \{ ((1, \text{up}), (2, \text{up})) \} = \emptyset \]

We use sets \( A \) and \( B \) to help us with the operation

\[ A = \{ ((n?, s), (n^\sim, s^\sim)) : \mathcal{P}((N \times S) \times (N \times S)) \land ((n?, s), (n^\sim, s^\sim)) \in \text{Connected} \} \land \]

\[ B = \{ ((n^\sim, s^\sim), (n?, s)) : \mathcal{P}((N \times S) \times (N \times S)) \land ((n^\sim, s^\sim), (n?, s)) \in \text{Connected} \} \land \]

\[ \text{Connected}' = \text{Connected} - (A \cup B) \]
Node\_Back

\[ n? : \text{Nodes} \]
\[ \text{State}^\prime(n?) = \text{up} \]

Connection\_Back

\[ ((n, s), (n^\sim, s^\sim)) ? : \mathcal{P}([\text{NXS}]\_\times [\text{NXS}]) \]
\[ \text{State}(n) = \text{up} \wedge \]
\[ s = \text{up} \wedge \]
\[ \text{State}(n^\sim) = \text{up} \wedge \]
\[ s^\sim = \text{up} \wedge \]
\[ \text{Connected}^\prime = \text{Connected} \cup \{((n, s), (n^\sim, s^\sim))\} \]

or, in a simpler yet equivalent way:

Connection\_Back

\[ n : \text{Nodes}, n^\sim : \text{Nodes} \]
\[ \text{State}(n) = \text{up} \wedge \]
\[ \text{State}(n^\sim) = \text{up} \wedge \]
\[ \text{Connected}^\prime = \text{Connected} \cup \{((n, \text{up}), (n^\sim, \text{up}))\} \]
8 Lecture - Communication Group Example, Continued

In this lecture we continue with the Communication Group example. The modeling process we are discussing, as a whole, is part of the larger review process, which is a process intended to find problems and issues of concern.

We use the methods to find problems:

1. Exact writing
2. Simulation according to an example
3. Writing proof

Bugs are much more costly when found in the customers’ sites, as opposed to finding them during the design phase.

So why don’t organizations spend more time on finding design problems? Nearly 80% of projects fail. So why aren’t these design methods implemented to find problems? The answer is that people don’t know how to perform them or use them effectively.

Usually, a simple model is only generated the third time a model is written. One of the best practices in software development is to rewrite your code; i.e., throw away the original and write it again from scratch.

Communication Group Example, Continued

We start with two groups (basic types): [Nodes], [States = \{up, down\}].

In our world we are allowed to refer to basic types and recursive activations
of the operations; the Power Set and Cartesian Product. We can mark them like this: \(I\left(\{\text{[Nodes]}, \text{[States]}\}, (X, \mathcal{P})\)\)

\[
\begin{align*}
\text{nodeState} : \mathcal{P}(\text{Nodes} \times \text{States}) \\
\text{nodeState} : \text{Nodes} \rightarrow \text{States}
\end{align*}
\]

The \(\rightarrow\) indicates that this is a function; that is, \(\text{nodeState}\) is a function (in this case, a full function).

\[
\begin{align*}
\text{Connection} : \mathcal{P}(\text{nodeState}) \\
x \in \text{Connection} \iff \vert x \vert = 2
\end{align*}
\]

The idea is to look at the arcs, which are connections between nodes of size two.

For example, \(\text{Connection} = \{(\text{A, up}), (\text{B, up})\}, \{(\text{A, up}), (\text{E, up})\}, \{(\text{B, up}), (\text{E, up})\}\).

\[
\begin{align*}
\text{Node\_Down} \\
n? : \text{Nodes} \\
\text{nodeState}'(n?) = \text{down} \land \\
\text{Connection}' = \text{Connection} \setminus \\
\{(n1, s1), (n2, s2) \in \text{Connection} \mid n1 = n?\}
\end{align*}
\]

\[
\begin{align*}
\text{Node\_Up} \\
n? : \text{Nodes} \\
\text{nodeState}'(n?) = \text{up}
\end{align*}
\]
\[ \text{Add\_Connection} \]
\[
(n_1, s_1), (n_2, s_2) : (\text{NodesXStates})
\]
\[
s_1 = s_2 = \text{up} \land
\text{Connection}’ = \text{Connection} \cup \{(n_1, s_1), (n_2, s_2)\}
\]

\[ \text{Remove\_Connection} \]
\[
\{(n_1, s_1), (n_2, s_2)\} : \text{Connection}
\]
\[
\text{Connection}’ = \text{Connection} - \{(n_1, s_1), (n_2, s_2)\}
\]

Example

We focus on the consistency of the group, which ensures that the model is correct. There are other possible focus areas that we do not deal with.

Our purpose is to characterize a set of nodes that are possibly communicating with one another. Each node can either be in a state of up or down. It is not possible for two nodes that are not both up to be connected.

We define an initial state in which all the nodes are ”down” and the Communication Group is empty (there is no communication between any of the pairs).

Nodes can change state to up or down, and communication couples can be added or removed. There are options to add/remove nodes.

We began with the definition of two basic types: [Nodes] and [States = \{Up, Down\}].

nodeState is a relation (a function) from Nodes to States.

\( f : \text{Nodes} \rightarrow \text{States} \)

\( f : \mathcal{P}(\text{Nodes X States}) \)
Next, we check the type of the Communication Group (Connection) set to verify it is a type we can refer to.

Reminder: Connection is a set of subsets of Nodes where the size of each subset is 2. We denote this in formal writing as:

\[
\text{Connection} : \mathbb{P}(\mathbb{P}(\text{Nodes}) | x \in \text{Connection} \Rightarrow |x| = 2
\]

This definition and specification differs from our previous definition of the Connection set in that in this definition the connection between two nodes is *not directed*.

Note: There is a hidden decision taken here, not specifically stated in the requirements. At this phase of the software development process, it is ”cheap” to change the design between a Connection set which is directed, and one which is not. Changes performed during the design phase are cheap, because they are performed before a few hundred lines of code are written.

We adjust our invariant for the indirected connection between nodes, or in formal writing:

\[
\{n_1, n_2\} = x \in \text{Connection} \Rightarrow (f(n_1) = \text{Up} \land f(n_2) = \text{Up})
\]

Another way to write the invariant, is that for any node that is down, the node cannot be in the Communication Group.

We continue by writing the operations explicitly in formal writing, and start with the Node_{Up} operation:

\[
\text{Node}_{Up} \\
\quad n? : \text{Nodes} \\
\quad f'(n?) = \text{Up}
\]
Node\textsubscript{Down} is more complicated, since it also needs to check in which connections the node participates, and disconnect those connections. If we don’t disconnect those connections - the invariant will be broken.

\[
\begin{array}{l}
\text{Node}_\text{Down} \\
\hspace{1em} n? : \text{Nodes} \\
\hspace{2em} f'(n?) = \text{Down} \land \\
\hspace{3em} \text{Connection}' = \text{Connection} - \\
\hspace{4em} \{ x = \{n_1, n_2\} \in \text{Connection} \\
\hspace{5em} \mid n_1 = n? \lor n_2 = n? \} \\
\end{array}
\]

The way to come up with the correct formal writing of the operations typically involves a roundabout process in which the requirements are defined in words and then translated to formal writing of the specification. In doing so, we find that there are missing details in the requirements. We correct the requirements, refine the formal specification, and so on in a roundabout process.

The \text{Connection}_\text{Up} and \text{Connection}_\text{Down} operations in formal writing are:

\[
\begin{array}{l}
\text{Connection}_\text{Up} \\
\hspace{1em} \{n_1, n_2\}? \in \mathcal{P}(\text{Nodes}) \\
\hspace{2em} f(n_1) = \text{Up} \land \\
\hspace{3em} f(n_2) = \text{Up} \land \\
\hspace{4em} \text{Connection}' = \text{Connection} \cup \{n_1, n_2\} \\
\end{array}
\]

\[
\begin{array}{l}
\text{Connection}_\text{Down} \\
\hspace{1em} \{n_1, n_2\}? \in \mathcal{P}(\text{Nodes}) \\
\hspace{2em} \text{Connection}' = \text{Connection} \setminus \{n_1, n_2\} \\
\end{array}
\]
Reminder: The Communication Group is the set of nodes such that each node is connected with any other node in the set.

We add an initial state for the Communication Group to be the empty set $\emptyset$.

**Question**

Do the operations Node-Up and Node-Down change the Communication Group?

**Answer**

In the current definition, there can be many Communication Groups, since we did not define it as the maximal group.

Therefore, we would like to add to the definition that the Communication Group must be the maximal group.

To make the Communication Group a single group, it will need to be a set containing the previous definitions, that is, a set of subsets in which each node is connected.

But, in this case, the definition becomes too complicated, and is out of the scope of this lecture.

To summarize what we have so far:

We defined the operations: Node_Up, Node_Down, Connection_Up and Connection_Down, an invariant and an initial state.

**Claim**
Prove that the invariant is preserved in any state of the system.

Proof
We prove using induction.

We begin with the initial state. For the atoms, we need to show that $\forall n :$ Nodes, $f(n) = \text{Down}$.

Since $\text{Connection} = \emptyset$, it is preserved by the empty set of couples that for every couple in $\text{Connection} \{n_1, n_2\} \in \text{Connection}$, $n_1$ and $n_2$ are both in the Up state, because there are no couples.

Next, we assume that the invariant is true; that is:

$\forall \{n_1, n_2\} \in \text{Connection}, f(n_1) = \text{Up} \land f(n_2) = \text{Up}$.

In the inductive step, we activate the different operations and show that the invariant is still preserved. We iterate through the operations, beginning with the Node_Up operation:

There are two options:

- If, before the operation, $f(n?) = \text{Down}$, then according to the inductive assumption, $n?$ was not in $\text{Connection}$. Therefore, after the operation, the invariant is still preserved.

- If, before the operation $f(n?) = \text{Up}$, then according to the inductive assumption, $n?$ might or might not have been in $\text{Connection}$. In this case, $n?$ stays Up, so we did not change $\text{Connection}$.

To avoid needing to check two options in the proof, we could add a test to the Node_Up operation that checks the state of the node before the operation.
Next, let’s prove for the Node\_Down operation. Again, there are two options:

- If, before the operation, $f(n?) = \text{Up}$, then in the Connection Group before the operation, $\text{Connection} = \{ \{n_1, n_2\} : P(\text{Nodes}) \mid (n_1 \neq n? \land n_2 \neq n?) \land \{n_1, n_2\} \in \text{Connection} \} \cup \{ \{n_1, n_2\} : P(\text{Nodes}) \mid (n_1 = n? \lor n_2 = n?) \land \{n_1, n_2\} \in \text{Connection} \}$. In words, we separate the Connection set into the pairs in which $n?$ participates and those in which $n?$ does not participate.

- If, before the operation, $f(n?) = \text{Down}$, then according to the inductive assumption, $n?$ wasn’t in the Connection group.

We leave the reader to similarly iterate through the other operations to complete the proof.
9 Lecture - Communication Group Example, Continued

Revisiting the Communication Group example, we find a different—more effective—interpretation and specification. We begin with the initial state definition:

\[
\begin{align*}
\text{Initial\_State} & \\
\forall n : \text{Nodes} & \\
\text{nodeState}'(n) = \text{Down} \land \\
\text{Connection}' & = \emptyset
\end{align*}
\]

Next, we formally describe the operations:

\[
\begin{align*}
\text{Connection\_Up} & \\
\text{n1?}, \text{n2?} : \text{Nodes} & \\
\text{nodeState}(\text{n1?}) & = \text{Up} \land \\
\text{nodeState}(\text{n2?}) & = \text{Up} \land \\
\text{Connection}' & = \text{Connection} \cup \{\text{n1?}, \text{n2?}\}
\end{align*}
\]

The basic types we use are [Nodes] and [States = \{Up, Down\}]. We additionally define the function: nodeState of type Nodes \to States. We define the Connection set using the following schema:

\[
\begin{align*}
\text{Connection} : \text{PP(Nodes)} & \\
\text{x} \in \text{Connection} & \iff | \text{x} | = 2
\end{align*}
\]

The invariant can then be written as this:
\[ \text{Invariant} \]
\[ x = \{n_1, n_2\} \in \text{Connection} \iff \text{nodeState}(n_1) = \text{Up} \land \text{nodeState}(n_2) = \text{Up} \]

We continue with the revised operations:

\[ \text{Node\_Down} \]
\[ n? : \text{Nodes} \]
\[ \text{nodeState}'(n?) = \text{Down} \land \text{Connection}' = \text{Connection} - \{x \in \text{Connection} \mid n? \in x\} \]

\[ \text{Node\_Up} \]
\[ n? : \text{Nodes} \]
\[ \text{nodeState}'(n?) = \text{Up} \]

\[ \text{Connection\_Down} \]
\[ x? : \text{Connection} \]
\[ \text{Connection}' = \text{Connection} - x \]

Let’s review the process; we started with a model. We then noticed that the model describes a directed graph. That is, the connections between the nodes in the Communication Group were directed, so we decided to recreate the model from scratch as an indirected graph.

We then discovered the model can be written in a simpler manner if we don’t write the state for each node in the graph, but rather only use the nodeState function on the nodes. This is a classic process of making things simpler by
iterations of specifications.
Simpler models preserve that they make are easier to find problems in, and
are easier to use to find problems in the system requirements.
This is a third iteration of the model, and we use it in this lecture to analyze
if it is correct. There is an interesting trade-off that says that as long as you
can continue claiming on the system then the system can be simpler, and
still hold true.

Claim
Our invariant claim is that for each member in Connection, both nodes have
a node state that is Up.

Proof
To prove the claim we need to start from the initial state where there are no
connections and all nodes are down and apply the operations in any order,
showing that the invariant is preserved. In the proof we use the inductive
set induced from the initial state and the defined operations. For the initial
state, the invariant is true according to the definition of the empty set, as
in the empty set there are no connections. In the inductive step, we assume
the invariant is true for any member of the inductive set, and need to prove
that the invariant is still preserved after activating any of the operations on
the members of the inductive set.
Any operation that is not invoked does not change the Connection set, and
therefore the invariant is preserved.
We iterate through the operations. For operation Node_Up, the input is n?.
There are two options:

1. n? is not a member of x, where x ∈ Connection and Node_Up does not
add any new members to Connection, so n? is not a member of any pair in Connection after activating the operation.

2. n? is a member of x, where x ∈ Connection. According to the inductive assumption, nodeState(n?) = Up, and the Connection is left unchanged. According to the inductive assumption, the Connection set preserves the invariant.

For operation Connection_Down, there are again two options:

1. \{n1?, n2?\} is a member of Connection.
   According to the inductive assumption, the invariant claim is valid for the Connection set, and so it is also valid for the set after removing the pair: Connection’ = Connection - \{n1?, n2?\}.

2. \{n1?, n2?\} is not a member of the Connection set, and so Connection is unchanged. According to the inductive assumption, Connection preserved the invariant before the operation, and as Connection was not changed, it still preserves the invariant after the operation.

For the operation Connection_Up, there are two options:

1. Either n1? or n2? or both, are in the Down state, or one of them, or both, are not in the Connection set. According to the inductive assumption, Connection preserves the invariant before and after the operation.

2. nodeState(n1?) = Up \land nodeState(n2?) = Up. In this case, the pair \{n1?, n2?\} is added to the Connection set, denoted in formal
writing by \( \text{Connection}' = \text{Connection} \cup \{\{n1?, n2?\}\} \). According to the inductive assumption, the \text{Connection} set preserved the invariant before the operation. As both \( \text{nodeState}(n1?) = \text{Up} \) and \( \text{nodeState}(n2?) = \text{Up} \), the set \( \text{Connection}' \) also preserves the invariant, as it includes the set \( \text{Connection} \) and a pair of nodes that are both in the Up state.

For operation Node\_Down, there are also two options:

1. \( n? \) is not a member of \( x \in \text{Connection} \). In formal writing, we write this as \( \{x \in \text{Connection} \mid n? \in x\} = \emptyset \). Activating the operation, we receive \( \text{Connection}' = \text{Connection} - \emptyset = \text{Connection} \). According to the inductive assumption, the invariant is preserved for both the \text{Connection} set and \( \text{Connection}' \) (which is equal to \( \text{Connection} \)).

2. \( n? \) is a member of \( x \in \text{Connection} \). Activating the operation, we receive \( \text{Connection}' = \text{Connection} - \{x \in \text{Connection} \mid n? \in x\} \). According to the inductive assumption, for each \( x \in \text{Connection}' \), if \( n? \notin x \), the invariant is preserved. Otherwise, \( n? \in x \) is not possible, since we removed it. We did not change the state of any of the nodes which are not \( n? \), and so the invariant is preserved in \( \text{Connection}' \).

This completes the proof, using induction on the model, that for any connection, both nodes are in the Up state.

The order of operations for checking a model is very similar to the order of operations for checking software:

1. We write the specification.

2. We read what we wrote.
3. We review the model. During the review, someone who did not write
the model acts as the reader, and we try to simplify the model.

4. The first three steps (reading/writing/reviewing the model) are a round-
about process, and have been shown as effective in finding problems.

5. In the exploration step we activate interesting scenarios of the model.

6. We test the software by planning a set of interesting scenarios.

7. Proof $\Leftrightarrow$ Model Checking. In this step, each of the possible scenarios
is explored and tested for validity and correctness.
10 Lecture - Testing and Verification

10.1 Communication Group Example Summary

We use the latest definition of the specification for examples and proofs.

The initial state was defined as:

\[
\begin{align*}
\forall n : Nodes & \rightarrow \text{nodeState}'(n) = \text{Down} \land \\
\text{Connection}' & = \emptyset
\end{align*}
\]

The invariant was formally defined as:

\[
\begin{align*}
\text{Invariant} & \quad x = \{n1, n2\} \in \text{Connection} \iff \\
& (\text{nodeState}(n1) = \text{Up} \land \text{nodeState}(n2) = \text{Up})
\end{align*}
\]

Our basic types were: [Nodes], [States = \{Up, Down\}]. We additionally used
the function nodeState of type Nodes \rightarrow States.

We defined the Connection set according to the following schema:

\[
\begin{align*}
\text{Connection} : \text{PP}(\text{Node}) & \quad x \in \text{Connection} \iff |x| = 2
\end{align*}
\]

The operations are:

\[
\begin{align*}
\text{Node\_Down} & \quad n? : \text{Nodes} \rightarrow \\
& \text{nodeState}'(n?) = \text{Down} \land \\
& \text{Connection}' = \text{Connection} - \{x \in \text{Connection} \mid n? \in x\}
\end{align*}
\]
\[ \text{Node}_\text{Up} \]
\[ n? : \text{Nodes} \]
\[ \text{nodeState}'(n?) = \text{Up} \]

\[ \text{Connection}_\text{Down} \]
\[ x? : \text{Connection} \]
\[ \text{Connection}' = \text{Connection} - x? \]

\[ \text{Connection}_\text{Up} \]
\[ n1?, n2? : \text{Nodes} \]
\[ \text{nodeState}(n1?) = \text{Up} \land \text{nodeState}(n2?) = \text{Up} \land \]
\[ \text{Connection}' = \text{Connection} \cup \{n1?, n2?\} \]

### 10.2 Testing and Verification

We focus on and explain three methods of testing and verification of software:

1. **Modeling Checking.**

2. **Functional Coverage** which is also practiced in the exercise, via a tool named FoCuS (http://www.alphaworks.ibm.com/tech/focus).

3. **Simulation.**

As hardware design these days uses VHDL, which is a software program in itself, software testing spans across the boundary of hardware and software and applies to both. In addition, the term **validation** occasionally replaces the term **testing**.
**Formal Verification** is a form of testing in which a computer program checks whether or not a model meets a certain property (termed invariant). For example, in our Communication Group model, we required that two nodes can be connected only if both nodes are in the Up state. We proved the invariant using induction on the operations (schemes). Sometimes, proofs of properties can be obtained using a computer program. This will be referred to as an instance of **Formal Verification**.

In today’s practice, hardware verification and testing includes formal verification, mostly performed using model checking, simulation, and the use of functional coverage. This practice requires a great deal of investment, but is justified as the price of a hardware defect is very high. To understand the price of a defect, estimate the price of having to collect all of the chips that were already distributed for a given release. On the other hand, the high investment is one of the reasons formal verification, simulation and functional coverage are *not* widely used in software testing. Other reasons include scalability and skills issues.

The typical price of a software defect for a middleware business software product might range between 25K$ and 40K$. In contrast, the price of a bug in hardware is much higher. Apparently, the price of a software bug does not justify the usage of the more stringent testing techniques applied for hardware.

Another category of tools that are used in software testing are **Static Analysis Tools** (for example, the FindBugs [http://findbugs.sourceforge.net/](http://findbugs.sourceforge.net/)) tool). These tools are used to analyze the code and search for possible defects. The example we gave in Section 2.1.4 applies techniques that are
typical to such tools.

Note: Typically, these techniques are equivalent to a definition of a set by induction and are sometimes referred to as **Fixed Point** techniques, as the inductive set can be viewed as a fixed point set, because reactivating the operations does not add any new members to the inductive set. It is interesting to note that these types of techniques are also used in model checking. Thus, induction on a set (as viewed as a fixed point of a set of operations) is an underline technique for both static analysis tools and model checking.

One of the main challenges in software testing is defining a single model that covers the different cases that need to be tested. This challenge is exuberated to a verification problem where the designer explicitly defines a small number of cases that are translated into a much larger number of cases.

For instance, to define all the natural numbers \( \mathbb{N} \), we can use a definition of an inductive set, with the atom 0 (zero) and the operation +1: \( \text{I}(0, +1) \). We used two items to define a very large (infinite in size) set.

There are problems with using an inductive set and proving using the induction principle:

1. You need to define a model.
2. You need to add a claim.
3. You need to be able to prove the claim.
4. The first three items need to be performed manually.

**Example**

Consider the following sample code excerpt:
1. if (a > 0) {
2.   print(a);
3. } else {
4.   print(-a);
5. }
6. if (b > 0) {
7.   print(b);
8. } else {
9.   print(-b);
10. }

After numbering the lines, we notice that there are two paths that result from the first ”if” statement in row #1; either row #1 and then row #2 are performed, or row #1 and then row #4 are performed. There are also two paths that result from the second ”if” statement in row #6: either row #6 and then row #7 are performed, or row #6 and then row #9 are performed. Since the paths are independent of each other, there are a total of four optional paths for the two ”if” statements.

Adding additional ”if” statements, each independent of the ones, exponentially increases the number of paths in the number of decisions (where each code statement is a decision). In our example, for two decisions we have $2^2$
possible paths, and in general (and we leave it for the reader as an exercise to prove), for $k$ decisions, we have $2^k$ possible paths.

In the next section, we explain Model Checking in detail.

### 10.2.1 Model Checking

It is often common that the designer visualizes a small scale of the whole problem that projects the whole problem. There are different approaches for projecting the whole problem in a small scale implementation. One such method is Model Checking. Model checking is a program that receives as INPUT the model, including the operations, the claim (invariant), and the initial state. Using the Communication Group example, you can review the initial state, operations, and invariants, and consider them as input to a computer program (see Section 7). In addition, some constraints are added. In the Communication Group example, [Nodes] is a finite state. Specifically, we might constrain it further to have just four nodes [Nodes = \{1, 2, 3, 4\}]. It is reasonable to assume that most of the interesting problems will also occur with just four nodes, where the number of possible states in the system is finite.

The OUTPUT of the model checker is a response stating that the claim (invariant) defined for the model is either true or false. In case of a false claim, an example is provided in which the invariant fails. The example will be in the form of history or sequence of operations that occur one after the other, initiating from the initial state and resulting in the failure of the invariant.

So, let's initiate the system:
Initial State

nodeState(i) = Down, i=1, 2, 3, 4
Connection = ∅

We define the Nodes basic type with just four nodes: [Nodes = \{1, 2, 3, 4\}].

We define four operations: Node_Up, Node_Down, Connection_Up, and Connection_Down.

What can the model checker do for us? The first thing that the model checker can do is to activate all the possible operations and save all the possible states.

For instance, activating Node_Up(n) we receive:

nodeState(i) = Down, i=2,3,4
nodeState(1) = Up
Connection = ∅

Doing the same for nodes 2, 3, and 4, we receive a model state in which all the nodes are in the Up state.

Another option is to activate the operation (Node_Down) on the initial state.

We then activate the Connection_Up operation, formally written as Connection_Up(i, j) for each i, j = 1, 2, 3, 4. As all the nodes are in the Down state, for every pair of values of i and j, the response from the Connection_Up operation is false and the model state remains unchanged.

Next, we activate the Connection_Down operation. As Connection = ∅, Connection_Down does not change the state of the model.

Table 14 lists all the possible states of the model with four nodes. State #5 is the first state in the list where there are at least two nodes that are in the Up state. In this case, we can start activating operations on the Connection set, and add the state of the Connection set to our model.

We continue to activate all the possible operations, and stop when we have a
Table 14: States Table

<table>
<thead>
<tr>
<th>Node</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initial State</td>
<td>Down</td>
<td>Down</td>
<td>Down</td>
<td>Down</td>
</tr>
<tr>
<td>State 1</td>
<td>Up</td>
<td>Down</td>
<td>Down</td>
<td>Down</td>
</tr>
<tr>
<td>State 2</td>
<td>Down</td>
<td>Up</td>
<td>Down</td>
<td>Down</td>
</tr>
<tr>
<td>State 3</td>
<td>Down</td>
<td>Down</td>
<td>Up</td>
<td>Down</td>
</tr>
<tr>
<td>State 4</td>
<td>Down</td>
<td>Down</td>
<td>Down</td>
<td>Up</td>
</tr>
<tr>
<td>State 5</td>
<td>Up</td>
<td>Up</td>
<td>Down</td>
<td>Down</td>
</tr>
<tr>
<td>State 6</td>
<td>Down</td>
<td>Up</td>
<td>Up</td>
<td>Down</td>
</tr>
<tr>
<td>State 7</td>
<td>Down</td>
<td>Down</td>
<td>Up</td>
<td>Up</td>
</tr>
<tr>
<td>State 8</td>
<td>Up</td>
<td>Down</td>
<td>Down</td>
<td>Up</td>
</tr>
<tr>
<td>State 9</td>
<td>Up</td>
<td>Down</td>
<td>Up</td>
<td>Down</td>
</tr>
<tr>
<td>State 10</td>
<td>Down</td>
<td>Up</td>
<td>Down</td>
<td>Up</td>
</tr>
<tr>
<td>State 11</td>
<td>Up</td>
<td>Up</td>
<td>Up</td>
<td>Down</td>
</tr>
<tr>
<td>State 12</td>
<td>Down</td>
<td>Up</td>
<td>Up</td>
<td>Up</td>
</tr>
<tr>
<td>State 13</td>
<td>Up</td>
<td>Up</td>
<td>Down</td>
<td>Up</td>
</tr>
<tr>
<td>State 14</td>
<td>Up</td>
<td>Down</td>
<td>Up</td>
<td>Up</td>
</tr>
<tr>
<td>State 15</td>
<td>Up</td>
<td>Up</td>
<td>Up</td>
<td>Up</td>
</tr>
</tbody>
</table>

complete list of all the possible states and activating operations does not add a new state. This situation is termed *closed over the operations*. According to the definition, this is an inductive set.

The main problem with model checking (the technological effort) is that there is not enough space to save all the possible states. This is an elementary problem with suggestions for resolution in multiple research works. The additional problem is that the process of activating all the possible operations in any possible model state might take a great deal of time.

10.2.2 Functional Coverage

Functional coverage assists in checking the model states we would like to cover in testing. Naturally, we would like testing to cover all the possible pairs of
node states. The key question functional coverage deals with is: “How do we know if we tested enough in the simulation?” Functional coverage provides a numeric estimate of the percentage of testing performed.

When defining our tests, we define the interesting states we would like to reach in our testing. As a starting point, we select two nodes. We define our requirements from testing:

- To cover all the possible operations: Node_Up, Node_Down, Connection_Up, and Connection_Down.
- To test states where there is no connection (the Connection set is $\emptyset$) and when there is at least one connection.
- To test all the possible states of two nodes: (Up, Up), (Up, Down), and (Down, Down).

These requirements serve as a basis for defining a functional coverage model. We use them to define a functional coverage model using the Z Specification Language:

We define nodeState as all the possible values of pairs of states of two nodes: $\text{nodeState} = \{\text{UpUp, UpDown, DownDown}\}$. We redefine Connection as stating if there is or isn’t at least one connection in the original Connection set: $\text{Connection} = \{\text{Yes, No}\}$.

We keep the original set of operations, defined as $\text{Operation} = \{\text{Node_Up, Node_Down, Connection_Up, Connection_Down}\}$.

The Functional Coverage Model
\[ M : \text{nodeState} \times \text{Connection} \times \text{Operation} \]
\[ \forall x : \text{Operation}, (\text{DownDown}, \text{yes}, x) \notin M \]
\[ \forall x : \text{Operation}, (\text{DownUp}, \text{yes}, x) \notin M \]

Coverage takes time and space constraints into consideration, and allows the definition of what to cover under the constraints. For instance, we could define that we want to cover any possible state of a node at least once.

10.2.3 Simulation

In the simulation process, again and again we activate an arbitrary order of the operations, testing, in each state, the correctness of the invariant. The key question in the simulation process is when to stop; that is, when we’ve tested enough.

The simulation process can be defined as containing the following steps:

1. Define all the states that we’d like to generate in simulation.

2. Defines states that can be seen in simulation (error states).

3. Help point simulations reach these states.

Lecture Summary

In this lecture we described these key terms:

- Modeling checking
- Functional coverage
- Simulation
In model checking, we started with the axioms, applied the operations, and tried to cover all the possible options. We ran into space and time restraint problems. We concluded the importance of the manner in which we explicitly and accurately define the problem.
11 Lecture - Abstraction

In this lecture we show how abstraction is used to eliminate details while still allowing us to model the problem. In abstraction, we disregard certain details in a way that preserves the important relationships between the model parts and participants.

11.1 Generic Example 3

We define the Nodes set as including six nodes: \( \text{Nodes} = \{1, 2, 3, 4, 5, 6\} \).
We would like to look at the pairs of nodes that can potentially communicate with one another. We term the group of pairs of nodes that can communicate as \( \text{pConnect} \). \( \text{pConnect} \) is of type pairs of Nodes, denoted in formal writing as \( \text{pConnect} : \mathcal{P}(\text{Nodes} \times \text{Nodes}) \).

We further define that a path exists between nodes \( a \) and \( b \) \((a, b \text{ in Nodes})\) if there is a series of \( k \) nodes \( a_1, a_2, ... , a_k \) such that \( a_1 = a \) and \( a_k = b \), and each pair of nodes \( (a_i, a_{i+1}) \) is in \( \text{pConnect} \), for every \( i = 1, 2, ... k-1 \).

To understand the idea of abstraction we define sub-cycles of nodes as a group of nodes in which we can reach each node from each of the other nodes in the group. The idea of abstraction, is that if it is possible to reach a certain node in the first sub-cycle from a certain node in the second sub-cycle, it is also possible to reach any node in the first sub-cycle from any node in the second sub-cycle.

We attempt to formalize this requirement by defining a basic type of the nodes \( [V] \) and a relation \( R \) of type pairs of nodes, denoted in formal writing as \( R : \mathcal{P}(V \times V) \).
We define $B$ as a set of subsets. $B = \{B_1, B_2, ..., B_n\}$ where $B_i : \mathbb{P}(V)$. Each $B_i$ is a sub-cycle of nodes; that is, each node in $B_i$ can communicate with any other node in $B_i$. Formally: $\forall B_i \in B, \forall s, t : V, s, t \in B_i$ there is a path between $s$ and $t$, and there is a path between $t$ and $s$. Also, $V$ is the union of all the sets of type $B_i$, formally written as: $V = \bigcup_{i=1}^n B_i$. 

$B$ is the concrete abstract definition of Abstraction.

We add an additional definition:

A Path over $B$ is an induced relation of $R$ over $B$, formally written as: $\tilde{R} : (B_i, B_j) \in \tilde{R} \iff \exists s \in B_i, t \in B_j$ such that $(s, t) \in R$.

**Exercise**

What is the type of $\tilde{pConnect}$?

We leave this as an exercise for the reader.

**Claim**

If a path exists in $\tilde{R}$ from $B_i$ to $B_j$ then there exists a path in $R$ from each node $t \in B_i$ to each node $s \in B_j$.

**Proof**

To explain the proof, we use an example. We assume that $(B_1, B_2) \in pConnect$ and $(B_2, B_3) \in pConnect$, and try to prove that there is a path in $pConnect$ from $B_1$ to $B_3$.

We pick a member $s$ from $B_1$, and a member $t$ from $B_3$. According to our assumptions that $B_1$ is a sub-cycle and that $s \in B_1$, there is a path from $s$ to any other member in $B_1$, and especially to the member $s' \in B_1$ which is the member that is connected to a member in $B_2$ (according to $(B_1, B_2) \in pConnect$). According to the definition of $B_2$, there is a path in $B_2$ between the member that was connected to $B_1$ and the member that is connected to
$B_3$. Also, according to the definition of $B_3$, there is a connection between the member of $B_3$ that was connected to $B_2$, which we mark as $t'$, and the member $t \ (\in B_3)$ that we want to reach.

Formal Proof

We look at the path from $B_i$ to $B_j$ and mark it as: $B_i = A_1, A_2, ..., A_k = B_j$. Since this is a path, $(A_i, A_{i+1}) \in \tilde{R}$, there exists $\exists s_i' \in A_i$ and $s_{i+1} \in A_{i+1}$ such that $s_i', s_{i+1} \in R$. We select $s \in B_i$ and $t \in B_j$. According to the definition of $A_i$, there is a path from $s$ to $s_i' \in A_1 = B_i$. Also, for each $A_i$, there is a path from $s_i$ to $s_i'$. For $A_k = B_j$ there is a path from $s_k$ to $t$, and therefore there is a path from $s$ to $t$, which is what we wanted to prove.

The idea of abstraction is that while you disregard details, you are still able to prove the claim.
12 Lecture - Relations and Functions

From this lecture onwards, we revisit the ideas discussed in the course. The following lectures do not include new material, but rather go in-depth into the material already covered in the course.

Question #1

Prove the following claim

Claim

Given: A ⊆ B ⊆ C, prove that: A ∪ B = B ∩ C.

Proof

We intuitively believe that A ∪ B = B and B ∩ C = B, which together implies that A ∪ B = B ∩ C.

To complete the proof, we use an assisting claim.

Assisting Claim

A ∪ B = B

Proof

We prove using bi-directional subsetting.

First, we would like to prove from left to right A ∪ B ⊆ B, that is, any member x of A ∪ B is also a member of B. By definition of the union operator, x ∈ A ∪ B implies that x ∈ A and x ∈ B.

By definition of subset, x ∈ A ∧ A ⊆ B implies that x ∈ B.

((x ∈ A) → (x ∈ B)) ⇒ ((x ∈ A) ∨ (x ∈ B)) ⇒ (x ∈ B) ⇒ A ∪ B ⊆ B,

which is what we wanted to prove.

Observation

Why do we use this type of theorem proofing? Because if we make it just a
little more formal, a computer could read and analyze it.

We look at p as x ∈ A and q as x ∈ B, and we use [(p ∨ q) ∧ (p → q)] → q ∨ q. To check if this statement is true, we can check the cases of p and q (each can be T or F) and use a truth table.

A second sentence we use is q ∨ q → q. Again, to validate if this statement is true, we can use a truth table.

Back to the proof of the assisting claim, we prove the other direction; that B ⊆ A ∪ B. By definition of ∪, x ∈ B ⇒ x ∈ B ∨ x ∈ A ⇒ x ∈ A ∪ B, which is what we wanted to prove.

In other words, marking q as x ∈ B and p as x ∈ A we proved that q → p ∨ q.

We leave for the reader to complete the proof by proving the second half of the bi-directional subsetting B ∩ C = B.

Question #2
Assume [X] is a type and A ⊆ X.

Prove that ∀ B ⊆ X, A ∩ B = A ⇔ A = Ø.

Proof

Again we use bi-directional subsetting to complete the proof.

We start with the right-to-left direction. Assuming A = Ø, then by definition of the empty set—a set with no members—A = x ∈ Ø ≡ F.

[x ∈ Ø → x ∈ A ∩ B] ≡ T ⇒ A ⊆ A ∩ B.

As for the left-to-right direction, assuming x ∈ A ∩ B, then by definition of ∩, x ∈ A ∧ x ∈ B. This implies that x ∈ A ⇒ A ∩ B ⊆ A ⇒ A ∩ B = A.

Since we started with any B, we proved that ∀ B ⊆ X, A ∩ B = A.

The original claim uses an if-and-only-if type statement, so we also need
to prove the other way around. Assuming \( \forall B \subseteq X, A \cap B = A \), and in particular, if \( B = \emptyset \), \( A \cap B = A \), then \( \emptyset = A \cap \emptyset = A \cap B = A \Rightarrow A = \emptyset \).

It still remains to prove that
\[ A \cap \emptyset = \emptyset. \]

We prove it using bi-directional subsetting:
\[ x \in \emptyset \Rightarrow [x \in \emptyset \rightarrow x \in A \cap \emptyset] \equiv T \Rightarrow \emptyset \subseteq A \cap \emptyset. \]

As for the other direction:
\[ x \in A \cap \emptyset \Rightarrow x \in A \land x \in \emptyset \Rightarrow x \in \emptyset \Rightarrow A \cap \emptyset \subseteq \emptyset \Rightarrow \emptyset = A \cap \emptyset. \]

**Additional Questions**

1. A, B \( \subseteq X \)
   \[ A - B = (X - B) - (X - A) \]
   Definition: \( A - B = \{ x \in A \mid x \not\in B \} \)

2. \( A = A \cap (X - B) \iff A \cap B = \emptyset \)

3. \( n = 1, 2, 3, ... \) \( A_n = \{ x \in R \mid |X| \geq \frac{1}{n} \} \)
   (a) Prove: \( A_1 \subseteq A_2 \subseteq A_3 \subseteq ... \)
   (b) Solve: \( \bigcup_{n=1}^{\infty} A_n = ? \)
   (c) Solve: \( \bigcap_{n=1}^{\infty} A_n = ? \)

**Claim**
\( \forall i \in \mathbb{N}, A_i \subseteq A_{i+1} \).

**Proof**

Given a certain \( i \), we take \( x \in A_i \Rightarrow (x \leq -\frac{1}{i}) \lor (\frac{1}{i} \leq x) \) according to the definition, so:
\[ \frac{1}{i} \leq x \Rightarrow \frac{1}{i+1} \leq x \Rightarrow (\frac{1}{i} \leq x \rightarrow \frac{1}{i+1} \leq x). \]
\[ x \leq \frac{1}{i} \Rightarrow x \leq -\frac{1}{i+1} \Rightarrow ((x \leq -\frac{1}{i}) \rightarrow (x \leq -\frac{1}{i+1})) \Rightarrow (x \leq -\frac{1}{i+1}) \lor (\frac{1}{i+1} \leq x). \]
We therefore need to prove (we leave this as an exercise for the reader) that:

\[ ((q \lor p) \land (p \to r) \land (q \to s)) \to (s \lor r) \]

\[ \cup_{n=1}^{\infty} A_n = R - \{0\} \]

\[ \cap_{n=1}^{\infty} A_n = A_1 \]

**Exercise**

Assuming \( R = \) all the numbers, and \( n = 1, 2, 3, \ldots \) \( A_n = \{x \in R \mid -n < x < \frac{1}{n}\} \)

Find:

1. \( R - \cup_{n=1}^{3} A_n = ? \)
2. \( \cap_{n=1}^{\infty} (R - A_n) = ? \)
3. \( \cup_{n=1}^{\infty} (R - A_n) = ? \)
4. \( \cup_{n=1}^{\infty} \cap_{k=1}^{n} A_k = ? \)

**Relations and Functions**

Assume the basic types \([A]\) and \([B]\) and define \( R \) as \( R : P(A \times B) \).

We can term \( R \) as a relation. Actually, \( R \) is a function, which is a special type of relation (in this case, a partial function):

\[
R : P(AXB)
\]

\[ \forall x \in A, \exists y, y' \in B, (x, y) \in R \land (x, y') \in R \Rightarrow y = y' \]

We define the domain of \( R \) as \( \text{Dom}(R) = \{x \in A \mid \exists y : B, (x, y) \in R\} \).

We further define the range of relation \( R \) as \( \text{Ran}(R) = \{y \in B \mid \exists x : A, (x, y) \in R\} \).
If $R$ is a function, we say that $R$ is a **full function** if $\text{Dom}(R) = A$, and we say that the function is over if $\text{Ran}(R) = B$.

If $R$ is a relation, we say that $R$ is a full relation if $\text{Dom}(R) = A$, and we say that the relation is over if $\text{Ran}(R) = B$.

Another type of relation is a one-to-one relation:

$$R : \mathcal{P}(AXB)$$

$$\forall x, x' \in A, \quad x \neq x' \land$$

$$\exists y, y' \in B, \quad [if (x, y) \in R \land$$

$$(x', y') \in R \Rightarrow$$

$$y \neq y']$$

**Lecture Summary**

In this lecture, we started revisiting the ideas we discussed in the course.

We discussed what does and does not constitute a function. We discussed what 1-to-1 functions and what surjective functions are. The definition of the 1-to-1 function is based on

if $x \neq y \Rightarrow f(x) \neq f(y)$.

We noted that $x \neq y \rightarrow f(x) \neq f(y)$ if and only if $\leftrightarrow ((f(x) = f(y)) \rightarrow (x = y))$ is a tautology.

If we mark $q \equiv ((f(x) = f(y))$ and $p \equiv (x = y)$ we actually show that: $(\neg p \rightarrow \neg q) \leftrightarrow (q \rightarrow p)$.

We can use a truth table to show this is a tautology (see Table 15).
<table>
<thead>
<tr>
<th>p</th>
<th>q</th>
<th>¬p</th>
<th>¬q</th>
<th>¬p → ¬q</th>
<th>q → p</th>
<th>(¬p → ¬q) ↔ (q → p)</th>
</tr>
</thead>
<tbody>
<tr>
<td>T</td>
<td>T</td>
<td>F</td>
<td>F</td>
<td>T</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>T</td>
<td>F</td>
<td>F</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>F</td>
<td>T</td>
<td>T</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>T</td>
</tr>
<tr>
<td>F</td>
<td>F</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
</tr>
</tbody>
</table>

### 13 Lecture - Defining a Library Example

In this lecture we continue to revisit the course material and go into depth using examples.

**Question**

Assume the group \( \mathbb{N} \) represents the natural numbers 0, 1, 2, ...

\( f : \mathbb{N} \rightarrow \mathbb{N} \) implies that \( f \) is a function from \( \mathbb{N} \) to \( \mathbb{N} \).

\( f : \mathcal{P}(\mathbb{N} \times \mathbb{N}) \) implies that \( f \) is a partial function of ordered couples of natural numbers. For instance, we can mark \( f(n) = n+1 \).

Another way of writing the function \( f \) is as a set of pairs: \( f = \{(i, j) \in \mathbb{N} \times \mathbb{N} | j = i+1\} \).

\( f \) is a relation or a partial set of the Cartesian Product \( \mathbb{N} \times \mathbb{N} \); that is, \( f \) is of type \( \mathcal{P}(\mathbb{N} \times \mathbb{N}) \).

Table 16 presents a few questions and answers on types, relations, and functions.
A formal proof that Dom(f) = \( \mathbb{N} \) is performed using bi-directional subsetting.

To prove that \( \mathbb{N} \subseteq \text{Dom}(f) \), we start with \( i \in \mathbb{N} \). According to the definition of \( f \), \( (i, i+1) \in f \), and \( i \in \text{Dom}(f) \), so \( \mathbb{N} \subseteq \text{Dom}(f) \).

The proof that \( \text{Dom}(f) \subseteq \mathbb{N} \) is according to the definition of \( f \) and Dom, as \( \text{Dom}(f) \) is a subset of the source of the function \( f \).

Since we proved bi-directional subsetting, we can conclude that \( \mathbb{N} = \text{Dom}(f) \).

**Question**

Is \( f \) a 1-to-1 function?

**Answer**

Yes.

**Proof**

We prove by negation. Assume that \( n_1, n_2 \in \mathbb{N} \), and \( f(n_1) = f(n_2) \). We will show that \( n_1 \) must equal \( n_2 \).
According to the definition of \( f \), \( f(n_1) = n_1 + 1 = n_2 + 1 = f(n_2) \).

Subtracting 1 from each side of the equality, we derive that \( n_1 = n_2 \), which is what we wanted to prove.

The proof is logically equivalent to our previous finding that \( n_1 \neq n_2 \rightarrow f(n_1) \neq f(n_2) \).

### 13.1 Defining A Function

We attempt to put together a relation or a function:

R : W \times Y. R is of type \( \mathcal{P}(\mathbb{N} \times \mathbb{N}) \).

Given that R is a relation from A to B, and T is a relation from B to C (where A, B, and C are sets of a given type), we can formally write them as

\[
R : \mathcal{P}(A \times B), \quad T : \mathcal{P}(B \times C).
\]

We define a new operator \( \circ \) as

\[
T \circ R = \{(a,c) | \exists b \in B, (a,b) \in R \land (b,c) \in T\}.
\]

The \( \circ \) operator preserves the following rules:

1. \( T \circ (S \circ R) = (T \circ S) \circ R. \)

2. \( T \circ (S \cap R) \subseteq (T \circ S) \cap (T \circ R). \)

3. \( T \circ (S \cup R) = (T \circ S) \cup (T \circ R). \)

4. \( (T \circ S) - (T \circ R) \subseteq T \circ (S - R). \)

We aim to prove these rules, starting with associativity.

**Claim**

\( T \circ (S \circ R) = (T \circ S) \circ R. \)

**Proof**
Assume R is a relation from A to B, S is a relation from B to C, and so on. We denote them in formal writing:

\[ R : \mathcal{P}(A \times B) \]
\[ S : \mathcal{P}(B \times C) \]
\[ T : \mathcal{P}(C \times D) \]

We select a pair \((a, d) \in T \circ (S \circ R)\).

According to the definition of \(\circ\), there is a member \(c\) such that \(\exists c \in C, (a, c) \in (S \circ R) \land (c, d) \in T\). Also, there is a member \(b\) such that \(\exists b \in B, (a, b) \in R \land (b, c) \in S \land (c, d) \in T\).

Since \((a, b) \in R, (b, d) \in T \circ S\), it implies that \((a, d) \in (T \circ S) \circ R \Rightarrow T \circ (S \circ R) \subseteq (T \circ S) \circ R\).

**Graph Example**

A graph is a set of nodes and arcs between the nodes.

**Question**

Build a function that returns the length of the communication between nodes for any two connected nodes (number of nodes in the path between the nodes).

The function can then be used to find the longest distance between any two nodes in the environment.

**Answer**

We begin by adding a hidden requirement from the system, which is to build it supporting distribution; that is, build a system containing many distributed processes that form paths in the graph.

We begin by formally defining the communication graph.

We define the basic types using the Natural Numbers \(\mathbb{N}\), and formally define
the group of nodes: $\text{Nodes} : \mathcal{P}(\mathbb{N})$.

We note that $\text{Nodes}$ is a finite set.

\[
\begin{align*}
\exists \ i : \mathbb{N}, & \ | \text{Nodes} | < i \\
\land & \ | \text{Nodes} | 1
\end{align*}
\]

Next, we define the communication graph:

\[
\begin{align*}
E : \mathcal{P}(\mathcal{P}(\text{Nodes})) \\
& x \in E \Rightarrow |x| = 2
\end{align*}
\]

We define $E'$ as the transitive closure of $E$. $E'$ is the inductive set that is derived from $E$ with the transitive operation $E' = I(E, "\text{Transitivity}" )$.

What is transitive? If \( \{v, w\} \in E' \land \{w, r\} \in E' \rightarrow \{v, r\} \in E' \).

How is $E'$ created? By looking at the arcs from each source to each target, check which arcs start with the target.

What is the type of $E'$? Same as $E$: $\mathcal{P}(\mathcal{P}(\text{Nodes}))$.

Next, we attempt to define a set of processes. These processes will form trails in the graph, but will also "get tired" and stop at some point, unless they return to a node they already visited. Any process needs to retain the information of where it started, how many nodes it passed so far, its name, and which nodes it passed so far.

We define a process for each node. We define the process $\text{PR}$ as a function, or in formal writing, $\text{PR} : \mathcal{P}(\text{Nodes} \rightarrow \text{Nodes} \times \mathbb{N} \times \mathbb{N} \times \text{Nodes})$, but since $\text{Nodes}$ is not a basic type, we need to alter this to be a type we are allowed to refer to: $\mathcal{P}(\mathbb{N} \times \mathbb{N} \times \mathbb{N} \times \mathbb{N})$. 

131
The first Nodes in the type is the node we started at; The second \( N \) in the type is the number of nodes we already passed in the path. The third \( N \) is the maximum number of nodes encountered till we stop the process. We mark this number as a finite natural number \( k > 1 \). The fourth Nodes is the current node.

We define an action of the trails passing from node to node using the following schema.

\[
(i_1, i_2, i_3, i_4)^? : PR, \\
v? : Nodes
\]

\[
(i_4, v^?) \in E \land \\
v^? \neq i_4 \land \\
v^? \neq i_1 \land \\
i_2 \leq i_3 \land \\
PR' = (PR \setminus \{(i_1, i_2, i_3, i_4)^?\}) \\
\cup\{(i_1, i_2 + 1, i_3, v^?)\}
\]

\( v^? \) is the node that we would like to reach.

The first condition means that you continue the trail to a node you can reach.

The second condition means that you reach a node that is not the current node.

The third condition means that you reach a node that is not the node you started with.

The fourth condition means that you have not yet reached the maximum number of nodes you are allowed in a trail.

Exercise

Remove the maximum number of nodes you can pass in the trail, and use only a single process (allow returning to the first node). What is the inductive
set that is a result of a single process, the action, and without the maximum number of nodes per trail? Prove it by induction.
14 Lecture - Defining a Library Example, Continued

14.1 Defining a Library

We begin with defining the basic types to use: [Books] and [Users].

We define an additional basic type: [bookState = \{Taken, notTaken\}].

Our purpose is to be able to represent all the books that a user holds. This will be represented by a function from a user to a subset of the book, denoted in formal writing as \( f : \text{[Users]} \rightarrow \text{P(Books)} \).

All the taken books can be represented using: \( \bigcup_{u \in \text{Users}} f(u) \).

All the books in the library, inLib, can be represented as all the books minus all the taken books, or in formal writing: \( \text{Books} \setminus \bigcup_{u \in \text{Users}} f(u) \).

We define an invariant: \( \emptyset = (\bigcup_{u \in \text{Users}} f(u)) \cap \text{inLib} \land (\bigcup_{u \in \text{Users}} f(u) \cup \text{inLib} = \text{Books}) \).

Instead of using bookState to define if a book is taken or not taken, we can define a relation \( B \) between books and users. \( B \) is not only a relation, but actually a function, because no book can be taken by more than one user.

\( B \) is of type \( \text{P(Books X Users)} \).

\( B : \text{Books} \rightarrow \text{Users} \).

\( B \) is a partial function, or in a formal definition:

\( \forall x \in \text{Books} (((\exists y \in \text{Users} \mid (x, y) \in B) \land (\exists z \in \text{Users} \mid (x, z) \in B)) \rightarrow z = y) \).

**Function Notation**

\( \rightarrow \) denotes a function.
denotes a full and bijective function.

\[ \rightarrow \] denotes a partial and bijective function.

\[ \rightarrow \rightarrow \] denotes a full and surjective function.

\[ \rightarrow \rightarrow \rightarrow \] denotes a partial and surjective function.

\[ \rightarrow \rightarrow \rightarrow \] denotes a partial function.

Each function can be bijective or not, surjective or not, and partial or not, equaling a total of eight different combinations of types of functions.

**Exercise**

Define a specification of all the different types of functions.

Check: Do they make sense?

**Partial Solution**

Instructions for solving the exercise: for each of the three attribute (bijective, surjective, partial) define one of three basic types and continue from there.

Continuing with the library definition example, we define an operation of borrowing a book. Before we do so, we require an additional definition of the total number of books:

All the taken books are defined as

\[ \text{borrowedBooks}: \bigcup_{u \in \text{Users}} \{b \in \text{Books} \mid (b, u) \in B\}. \]

Another form of writing the list of all taken books: \( \{b \in \text{Books} \mid \exists u \in \text{Users}, (b, u) \in B\} \).

\text{inLib} are all the rest of the books, that is, the books that are not borrowed and are in the library.

\text{inLib} is of type: \( \mathcal{P}(\text{Books}) \).

Another way of thinking about it is that \text{inLib} is a subset of books, or in formal writing: \( \text{inLib} \subseteq \text{Books} \), and \text{Books} is a type, and therefore \text{inLib} is
of type $\mathbb{P}(\text{Books})$.

Next, we define an invariant on Books:

\[
\text{borrowedBooks} \cup \text{inLib} = \text{Books}.
\]

\[
\text{borrowedBooks} \cap \text{inLib} = \emptyset.
\]

Now we have the infrastructure required for defining the operations. We begin with the definition of the operation borrowedBook. In the initial state, borrowedBooks = $\emptyset$ and inLib = Books.

| $b?$ : Books |
| $u?$ $\in$ Users |
| $\neg \exists u \in \text{Users},$ |
| $(b?, u) \in B \land$ |
| $B' = B \cup \{(b?, u?)\} \land$ |
| inLib' = inLib - \{b?\} |

**Exercise**

Define the operation of returning a book.

**Exercise**

Define the state of the system after each activation for a series of activating the operations five times.

**Course Summary, so far**

To summarize where we have reached in the course, we went through first order logics, propositional calculus and inductive sets, and induction on the structure, and described uses for analyzing programs, semantics of computer languages, testing and checking specifications of software using building a model, using exploration of the model, using proofs and model checking.
14.2 File System Example

In terms of the OS, a file system is a series of bytes that are organized in blocks and other data models. In the OS, the bytes are stored as a tree, but in terms of the user, they appear to be a series of bytes that are somehow consecutive. We aim to model the abstraction that the user sees as a series of bytes.

Each byte can either contain or not contain data.

An abstraction of a file can appear to be something like 010011end-of-file.

When trying to model the environment, we think of the following possible operations: create, open, close, delete, read, write, change, and so on.

Later, we might also want to model the directories. For instance, we might want to model the dir or ls commands.

We begin with modeling a file as a series of bytes with a specific ending character termed end-of-file.

We define the needed basic types: \{0, 1, end-of-file\}.

We add a restriction that denotes that end-of-file is the last character.

If we describe the file as an abstraction of a1a2a3a4... we can describe it as a function from a numbered index (\(\mathbb{N}\)) to the basic type. Hence, the file is of type function from \(\mathbb{N}\) to \{0, 1, end-of-file\}.

Next, we add restrictions that disallow the appearance of end-of-file twice, or not at the end of the file.
We add a restriction that denotes that once an `end-of-file` character appears, any character appearing after that is an `end-of-file` character.

**Exercise**

Define the operation `read` that takes an input file, an index place, and a number of bits to read, and returns the requested information.

### 14.3 Preparation for Final Exam

In preparation for the final exam, go over the lectures and exercises, and solve the exercises.

The final exam contains four questions, of which you will need to answer three.

Types of questions:

- Define a type of specification from a given description.
- Define a simulation according to a definition of a system (for instance, given an initial state and a definition, what happens after you apply operations).
- Prove a simple characteristic on a given definition.
• Answer technical questions, such as determining the resulting group from a sample of an inductive operation, or proving a characteristic of the resulting type.

• Show equality between sets, relations, functions, and so on.

• Analyze simple code snippets.
Part II

Exercises

15 Exercise - Logics

This exercise covers these topics:

- A reminder on basic of predicate calculus.
- From predicate calculus, how we derive semantics.
- From semantics, how we derive verification using invariants.

15.1 Predicate Calculus

Predicate calculus is, in essence, a proving system. To use it, you choose a language (using basic terms such as ”+” or ”period”) and a collection of axioms, and then you can start deriving formal conclusions from the axioms. Prepositional calculus describes a valid conclusion.

15.2 Semantics

Semantics are used to investigate the meaning associated with the syntax. To derive semantically means that something is true in any model in which the axioms are true.
15.3 What Constitutes a Proof?

A proof must be based on axioms. Otherwise, proving that 1 plus 1 equals 2 takes more than 200 pages.

15.4 Logics

15.4.1 Sets

Different ways to mark sets:

1. Curly parentheses, \{\}, with the group members inside.
   
   (a) An explicit list of members in parentheses; for instance, \{1, 2, 70\}.
   
   (b) Using a list that explains a rule; for instance, \{0, 1, 2, 3, ...\}.
   
   (c) Using a common attribute; for instance, \{i: where i is an even number\}.

2. By giving the set a name using a known letter; for instance, \(\mathbb{N}\equiv\) the natural numbers.

15.4.2 Basic Terms

• **In:** \(x \in A\) means that the member \(x\) belongs to set \(A\).

• **Subset:** \(A \subseteq B\) means that the set \(A\) is a subset of set \(B\) if for every \(x \in A\) it also holds that \(x \in B\).

   - \(A \subseteq B\), then it is said that \(A\) is a subset of \(B\).
   
   - **Note:** The empty set \(\{\}\), also known as \(\emptyset\), is a subset of any other set.
• A true subset: \( A \subset B \), if for every \( x \in A \), it also holds that \( x \in B \) and there exists at least one \( x_0 \in B \), such that \( x_0 \notin A \).

• Equality among sets: Two sets, \( A \) and \( B \), are equal (denoted \( A = B \)), if \( A \) and \( B \) have the same members.
Equality between sets is proved using bi-directional subsetting.

15.4.3 Actions between Sets

• Unity: \( A \cup B = \{ x : x \in A \text{ or } x \in B \} \).

• Intersection: \( A \cap B = \{ x : x \in A \text{ and } x \in B \} \).

• Subtraction: \( A \setminus B = \{ x : x \in A \text{ and } x \notin B \} \).

• Complementary: \( A^c = x : x \notin A \).
Note that in the complementary operator you must also define the world in which you are working.

• Foreign Sets: where \( A \cap B = \emptyset \).

Example #1
Given a set of Scandinavian countries: Scandinavian = \{Denmark, Finland, Norway, Sweden, Iceland\}, and a definition of a set named Benelux: Benelux = \{Belgium, Netherlands, Luxembourg\}, what are:

• Scandinavian \( \setminus \) Benelux = ?

• Scandinavian \( \cap \) Benelux = ?

• Scandinavian \( \cup \) Benelux = ?
• Given that the world is all the countries in the world, find: Scandinavian\(^c\) = ?

**Example #2**

Prove formally that:
\[ A \cap (B \cup C) = (A \cap B) \cup (A \cap C). \]

**Proof**

To prove the equality we use bi-directional subsetting:

1. We begin by proving that: \( A \cap (B \cup C) \subseteq (A \cap B) \cup (A \cap C). \)

2. Next, we prove that: \( (A \cap B) \cup (A \cap C) \subseteq A \cap (B \cup C). \)

**Additional Exercises**

Prove formally that:

1. \( A \cup (B \cap C) = (A \cup B) \cap (A \cup C). \)

2. \( (A \cap B) \cup (A \setminus B) = A. \)

**Example #3**

Figure 2 displays a Venn Diagram.
1. Color the area that is: $A \cup B$.

2. Color the area that is: $(A \cap B) \cup C$.

3. Color the area that is: $(A \setminus C) \cup (B \setminus C)$.

4. Color the area that is: $(B \setminus A) \cup C$.

### 15.5 Truth Tables

We use the following notations:

- **not** relationship: $\neg$.

- **or** relationship: $\lor$.

- **and** relationship: $\land$.

Following are the truth tables for logical relationships. Table 17 describes the ”if then” relationship denoted by $p \rightarrow q$. 
Table 17: If Then Relationship Truth Table

\[
\begin{array}{c|c|c|}
 p & q & p \rightarrow q \\
 t & t & t \\
 t & f & f \\
 f & t & t \\
 f & f & t \\
\end{array}
\]

Table 18 describes the "if and only if" relationship denoted by \( p \leftrightarrow q \).

Table 18: If And Only If Relationship Truth Table

\[
\begin{array}{c|c|c|}
 p & q & p \leftrightarrow q \\
 t & t & t \\
 t & f & f \\
 f & t & f \\
 f & f & t \\
\end{array}
\]

Exercise #1
Build a truth table for the statement: \((p \land q) \lor r\).

Exercise #2
Build a truth table for the statement: \((p \lor q) \land (q \lor r)\).

15.6 Logical Equivalence

**Definition:** Two statements are said to be logically equivalent if they have
the same set of values under the same assignment to their atoms, or, in other
words, they have the same truth tables.

Exercise #3
Prove that \( p \leftrightarrow q \) and \((p \rightarrow q) \land (q \rightarrow p)\) are logically equivalent.
15.7 Logical Quantifiers

We use the following notations:

- There exists $\exists$.
- For all $\forall$.

Example #4
Claim: "Beethoven was a composer of opera". A different way to write the sentence in a logically formal way is: "There exists an opera such that Beethoven was the composer of", or, in a mathematical logic formal writing: $\exists x : \text{opera}$ such that Beethoven was the composer of $x$.

Another claim: "Sao Paolo is bigger than any city in Europe". A different way to write the sentence is that "For every city $c$, if $c$ is in Europe, then Sao Paolo is bigger than $c$". In a mathematical logic equivalent of formal writing:

$\forall c : \text{city such that} c \text{ is in Europe} \Rightarrow \text{Sao Paolo is bigger than } c$.

$\forall p$ is the same in mathematical logic formal writing as $\neg \exists \neg p$. So, we can also write our previous sentence as this:

$\neg \exists c : \text{city such that} c \text{ is in Europe} \Rightarrow (c \text{ is in Europe} \Rightarrow \text{Sao Paolo is bigger than } c)$.

This is an example of translating $\forall$ to $\neg \exists$.

Example #5
"There is a certain country to which Sao Paolo belongs, and Sao Paolo is bigger than any other city in that country.”

We break this up into parts of the sentence, and start to write it formally:

$\exists \text{ co : country, such that Sao Paolo is in co} \land \forall \text{ ci : city, ci is in co} \land \neg \text{ ci is Sao Paolo, then } \Rightarrow \text{ Sao Paolo is bigger than } ci$. 
Exercise

Explain the following sentence: $\exists \ p : \text{person}, \text{such that } \forall \ n : \text{person}, \text{and } n$ is a neighbor of $p \Rightarrow p$ never speaks to $n$.

Exercise

Explain the following sentence: $\forall \ p : \text{person}, \exists \ n : \text{person}$ such that $\forall \ t : \text{person}, \neg n = t \Rightarrow \neg n$ lives with $t \land p$ knows $n$. 
15.8 Inductive Definition

Given that:

- $X$ is the world upon which the set is built.
- $A$ is the core set, and a subset of $X$.
- $P$ is a set of creation operations.

We define the set $I(A, P)$ as the set that preserves a set of requirements.

Example #6

$X$ is the set of real numbers $\mathbb{R}$, $A = \{0\}$, and $P = \{f^+\}$ - $f^+(x) = x + 1$.

The derived inductive set $I(A, P) = \mathbb{N}$. To show that this is true, we need to show that the inductive set preserves the following rules:

1. $A$ is a subset of $I(A, P)$: $A \subseteq I(A, P)$.

2. Closure over the operations in $P$:

   If $f \in P$ (where $f$ is an operation from $P$) with $n$ values, and $x_1, x_2, ..., x_n \in I(A, P)$, then $f(x_1, x_2, ..., x_n) \in I(A, P)$.

3. Minimality: for every set $Z$ that preserves both conditions 1 and 2:

   $I(A, P) \subseteq Z$

Example #7

1. Vector with operation: the inductive set induced by an operation defined as a vector $w$ receives $c$ times $w$ is the straight line (span vector $v$).
2. The inductive set resulting from \{1\} and the operation $x$ to $2x$ is $\langle 2^N \rangle$.

3. The inductive set resulting from the starting point $(0, 0)$ and the operation $(x, y) \rightarrow (x, y + 1)$ or $(x + 1, y)$ or $(x, y - 1)$ or $(x - 1, y)$ is the lattice.

4. We can define an inductive set describing the famous game "Towers of Hanoi", starting with the initial state and defining a set of operations as all the operations that preserve the rule that you can only put smaller pieces on larger pieces.

   Deriving the inductive set then allows us to check if a certain legal possibility exists in I(A, P), and can be used to check if the solution can be achieved.

5. We can define an inductive set of the game of chess. The definition of the initial state is easy, but defining all the possible operations might be a tough task.

### 15.9 Proof Using Induction

Assume the following code snippet:

```c
while(x > 0){
    x = x - 1
}
```

For instance, consider an example in which we initiate the value of $x$ to be 3. By defining the inductive set, we are actually defining a connection between
the activation plan and the inductive set.

We aim to prove that \( x \) is always greater than or equal to zero using induction. To prove using induction means that we need to show that for every \( x \in I(A, P) \), \( x \geq 0 \).

1. According to the inductive proof method, we begin the proof by showing that the atoms preserve the claim. If, for instance, \( n=1 \), then since \( n \geq 0 \), we proved for the atom.

2. In the inductive step, we assume the claim is true for any member of the inductive set; that is, the member is \( \geq 0 \). We need to show that the result of activating any of the operations on any of the members still preserves the claim.

There are two possibilities:

(a) The member we selected is zero, in which case we do not perform the subtraction and it remains zero, and therefore preserves the claim.

(b) The member we selected is greater than zero, so after subtracting 1, it is still \( \geq 0 \).

**Exercise Summary**

In this exercise we discussed the following topics:

- Basics of logics and of key definitions.
- Basics of truth tables and representations of some basic relationships using truth tables.
• Basic ways to prove logical equivalence.

• We reached the basic definition of the inductive set and the inductive form of definition and proof method.
16 Exercise - Inductive Definition

Given that

- $X$ is the world above which the set is built.
- $A$ is the core set, and a subset of $X$.
- $P$ is a set of creation operations.

The inductive set $I(A, P)$ is the set that preserves a set of requirements.

Example

Assume $X$ is the set of real numbers, denoted by $\mathbb{R}$. We define $A = \{0\}$ and $P = \{f^+\}$ such that $f^+(x) = x + 1$.

From the definition of $A$ and $P$ we derive that $I(A, P) = \mathbb{N}$.

We verify that $I(A, P) = \mathbb{N}$ using validation of the following three requirements:

- $A$ is a subset of $I(A, P)$: $A \subseteq I(A, P)$.
- Closure over the operations in $P$. Closure means that for every operation $f$ in $P$ (formally written $f \in P$) with $n$ values, and $x_1, x_2, ... x_n \in I(A, P)$, then $f(x_1, x_2, ... x_n)$ is also $\in I(A, P)$.
- Minimal: for every set $Z$ that preserves both conditions 1 and 2:
  $I(A, P) \subseteq Z$.

Examples

1. A vector with the following operation: Vector $w$ receives $c$ times $w$.
   The inductive set is the straight line (span vector $v$).
2. Defining an inductive set that is a lattice: begin with the initial point 
(0, 0), and an operation: \((x, y) \rightarrow (x, y + 1)\) or \((x + 1, y)\) or 
\((x, y - 1)\) or \((x - 1, y)\).

3. An inductive set defining the ”Towers of Hanoi” example: begin from 
the initial state, and define the operation according to the rule that 
you can only put smaller pieces on larger pieces. 

Defining the ”Towers of Hanoi” problem using an inductive set allows 
us to ask if a certain legal possibility exists in \(I(A, P)\). A state in \(I(A, P)\) is a state we can reach using the allowed operations starting from 
the initial state. This method allows us to check if the solution can be 
achieved.

4. Similarly, but more complicated in the definition of the operations, is 
an inductive set representing a chess game.

16.1 Proof Using Induction Revisited

Consider the following code snippet:

\[
\text{while}(x > 0) \{ \\
\text{ } x = x - 1 \\
\} 
\]

For instance, we look at the example where \(x\) is initialized to the value 3, \(x=3\). 
We aim to describe the connection between the activation plan of the code 
snippet and an inductive set. For instance, we use the inductive set to prove
that values of $x$ are always greater than or equal to zero. That is, for every $x \in I(A, P)$, it implies that $x \geq 0$. How can we prove it using induction? We use the inductive proof method containing the following steps:

1. We show the claim holds for the atoms. We choose a first atom; for instance, $n=1$, and since $n \geq 0$, we show for the atom that $x \geq 0$.

2. The inductive step: assume all members of $I(A, P)$, preserve the claim that they are $\geq 0$. We need to show that members derived from these members by applying the operations are also $\geq 0$. There are two possibilities:

   (a) The member we select is zero, in which case we do not perform the subtraction and it remains zero;

   (b) The member we choose is $> 0$, so after subtracting 1, it is still $\geq 0$. 

154
Example Program #1

Look at the following code snippet:

\[
\begin{align*}
    x &= a+b; \\
    y &= a-b; \\
    \text{while } ((a+b)(a-b) > 0) \{ \\
    &\quad a = a+1; \\
    &\quad x = a+b; \\
    &\quad y = a-b; \\
    \} 
\end{align*}
\]

We mark each row with a line number:

1. \( x = a+b; \)
2. \( y = a-b; \)
3. \( \text{while } ((a+b)(a-b) > 0) \{ \)
4. \( a = a+1; \)
5. \( x = a+b; \)
6. \( y = a-b; \)
7. \( \}

Figure 3 displays the code snippet’s execution plan (control flow).
We mark in red the information known at the entrance to each command:

1. \( x = a + b; \) \( \emptyset \)

2. \( y = a - b; \) \( a + b \)

3. while \( ((a+b)(a-b) > 0) \) \{ \( a+b, \ a-b \)

4. \( a = a+1; \) \( a + b, \ a-b, \ ((a+b)(a-b)) \)

5. \( x = a+b; \) \( \emptyset \)

6. \( y = a-b; \) \( a+b \)

7. \} \( a+b, \ a-b \)

We attempt to describe the world of our knowledge in terms of the paths that lead to each row and the information added in each row. For instance, see the following statement:
This means that when entering row #1, we add to any knowledge we had before entering row #1 the additional knowledge of \((a+b)\). Similarly, we derive the same type of statements for the other rows for which our knowledge changes:

\[ (1, P) \rightarrow (1, P \cup \{a+b\}) \]

The second statement means that we can reach row #3 from either rows #2 or #6. The reason for the use of intersection \(\cap\) between \(P\) and \(Q\) is that we want to know for certain what the information is when entering row #3. Only using the intersection \(\cap\) operator ensures that the information is surely known.

When activating row #3, we calculate the additional information: \((a+b), (a-b), (a+b)(a-b)\), which is therefore added to our knowledge. Similarly, the additional rows are

\[ (3, P), (4, Q) \rightarrow (4, P \cup Q \setminus \{\text{anything containing } a\}) \]

\[ (4, P), (5, Q) \rightarrow (5, P \cup Q \cup \{a+b\}) \]

\[ (5, P), (6, Q) \rightarrow (6, P \cup Q \cup \{a-b\}) \]

Exercise Summary

- Definition of the inductive set.
- Uses of inductive sets in proofs.

- Test case example of a small simple code snippet.
17 Exercise - Syntax and Semantics

17.1 Sample Code Snippet

Let’s look at the following program:

```c
if (a < b) {
    x = b-a;
    y = a-b;
} else {
    z = b-a;
    t = a-b;
}
```

We quickly realize that a more optimal execution would have done the following (assuming this code segment is within a loop):

```c
r = b-a;
f = a-b;
if (a < b) {
    x = b-a;
    y = a-b;
} else {
```

159
\( z = b-a; \)

\( t = a-b; \)

\}

If, when entering the first "if" statement, we had already calculated \((b-a)\) and \((a-b)\), we could refrain from calculating them again. We aim to see a visualization of this in the induction set, meaning that the knowledge of \((b-a)\) and \((a-b)\) at the "if" statement could help.

We create an association of each program location with the set of expressions that are used—if calculated—along any path, up to the exit from the code segment. Next, we mark each row with a line number:

1. if \((a < b)\) {
2. \hspace{0.5em} \(x = b-a;\)
3. \hspace{0.5em} \(y = a-b;\)

} else {
4. \hspace{0.5em} \(z = b-a;\)
5. \hspace{0.5em} \(t = a-b;\)

}

Figure 4 displays the code snippet’s execution plan (control flow).
We mark the rows as operations numbered 1 through 5.

We start with the set of atoms:

\[ A_0 = \{1, \emptyset\}, \{2, \emptyset\}, \{3, \emptyset\}, \{4, \emptyset\}, \{5, \emptyset\} \]

Next, we describe the world of our knowledge in terms of the paths that lead to each row and the information that is added in each row, but in reverse execution order:

- \((5, P) \rightarrow (5, P \cup \{a-b\})\)

This statement means that when entering row 5, we add to any knowledge that we had before, the knowledge of \((a-b)\).

Similarly:

- \((3, P) \rightarrow (3, P \cup \{a-b\})\)

- \((5, P), (4, Q) \rightarrow (4, P \cup Q \cup \{b-a\})\)
This means that in the execution order, we can get to row 5 only from row 4. So, in reverse execution order, row 4 follows row 5, and adds the information of \((b-a)\).

Similarly:

\[ (3, P), (2, Q) \rightarrow (2, P \cup Q \cup \{b-a\}) \]

Finally, we add:

\[ (2, P), (4, Q), (1, R) \rightarrow (1, (P \cap Q) \cup R) \]

This means that from row 1, we can go to either row 2 or row 4. So, in reverse execution order, we reached row 1 either from row 2 or row 4. If we want to know the information that is known for certain, it is the intersection \(\cap\) of the information known from rows 2 and 4, with the additional information added in row 1.

**NOTE:** If we had something like \(a = a + 1\), then \((a+b)\) and \((a-b)\) would have been deduced before exiting the code segment.

Now, let’s try to activate the rules. From the rule:

\[ (5, P) \rightarrow (5, P \cup \{a-b\}) \]

Activated on: \(\{5, \emptyset\}\)

\[ (5, \emptyset) \rightarrow (5, \{a-b\}) \]

From the rule:

\[ (3, P) \rightarrow (3, P \cup \{a-b\}) \]
Activated on: \(\{3, \emptyset\}\)

- \((3, \emptyset) \rightarrow (3, \{a-b\})\)

From the rule:

- \((5, P), (4, Q) \rightarrow (4, P \cup Q \cup \{b-a\})\)

Activated on: \(\{5, \{a-b\}\}\) and \(\{4, \emptyset\}\)

- \(\{5, \{a-b\}\}, \{4, \emptyset\} \rightarrow (4, \{a-b, b-a\})\)

From the rule:

- \((3, P), (2, Q) \rightarrow (2, P \cup Q \cup \{b-a\})\)

Activated on: \(\{3, \{a-b\}\}\) and \(\{2, \emptyset\}\)

- \(\{3, \{a-b\}\}, \{2, \emptyset\} \rightarrow (2, \{a-b, b-a\})\)

From the rule:

- \((2, P), (4, Q), (1, R) \rightarrow (1, (P \cap Q) \cup R)\)

Activated on: \(\{2, \{a-b, b-a\}\}, \{4, \{a-b, b-a\}\}\) and \(\{1, \emptyset\}\)

- \(\{2, \{a-b, b-a\}\}, \{4, \{a-b, b-a\}\}, \{1, \emptyset\} \rightarrow (1, \{a-b, b-a\})\)

We look at all the expression results that we will use in the future, expression results that we know will not change in any path until the exit of the code segment. We search for reoccurring phrases that can be calculated in advance. Eventually, we reach a Fixed Point, since after we activate the rules again, nothing changes.
17.2 Differences between Syntax and Semantics

Example

Assume the induction set \( I(A, P) \) is a description of a language: "\( pq \)" (a new language just invented):

\[
X = \{ e : \text{a series of signs } p, q \}.
\]

\[
X = \{ p, q, - \}^*.
\]

\[
A = \{ pq \}.
\]

\[
P = \{ f, g \}.
\]

Where \( f \) is the operation that adds a "\(-\)" to the beginning and end of the string, and \( g \) is the operation that adds a "\(-\)" to the end and in the middle of the string.

A few examples of "words" in this language are:

- \( pq \)
- \( -pq- \)
- \( --pq-- \)
- \( --p-q-- \)

When explaining formal languages, we can use explanations of

- syntax
- semantics (meaning)

Returning to the example, let’s try and give semantics to our newly created language, \( pq \). For instance, let’s try to see what we get from the semantics that define
• $- = 1$

• $-- = 2$

• ...

We further translate

• $p$ as $+$

• $q$ as $=$

Then, the outcome is mathematical exercises.

### 17.3 Power Set

**Definition**

A Power Set of Set $A$ is the set of all the possible subsets of $A$.

Notation: $\mathcal{P}(A) = \text{Power set of } A$.

**Exercise**

What is the $|\mathcal{P}(A)|$ (size of $\mathcal{P}(A)$)?

**Answer**

$|\mathcal{P}(A)| = 2^{|A|}$.

$\mathcal{P}(A) = \{ S : S \subseteq A \}$.

**Example**

Given: $A = \{1, 2\}$.

$\mathcal{P}(A) = \{\emptyset, \{1\}, \{2\}, \{1, 2\}\}$.

**Quick Quiz**

• Is $2 \in \mathcal{P}(A)$? - No.
• Is \( \{2\} \subseteq P(A) \)? - No.

• Is \( \{2\} \in P(A) \)? - Yes.

Example
Suppose we define the following inductive set:
\[
X = \mathbb{N} \text{ - the set of natural numbers.}
\]
\[
A = \{i : i > 1 \land i \text{ is odd}\}.
\]
\[
P = \{f_1\} \text{ where } f_1(x) = 2x.
\]

Question
What is \( I(A, P) \)?

Answer
Let’s mark \( I_{\text{first}} = I(A, P) \).

Let’s look at a few example sets

• \( I_1 = \{1, 3, 5, 7, \ldots\} \)

• \( I_2 = \{0, 1, 2, 3, \ldots\} \)

• \( I_3 = \{3, 5, 6, 7, 8, 10, 11, 13, \ldots\} = \{i \in \mathbb{N} : i \neq 0 \land i \text{ is not a power of 2}\} \)

Is \( I_1 = I_{\text{first}} \)?

\( I_1 \neq I_{\text{first}} \) because \( I_1 \) preserves the first trait, but \( I_1 \) does not preserve the second trait:
\[
f_1(1) = 2 \notin I_1.
\]

Is \( I_2 = I_{\text{first}} \)?

\( I_2 \) preserves the first and second trait, but \( I_2 \) does not preserve the third trait because \( I_3 \subset I_2 \) and \( I_3 \) preserves traits 1 and 2. Therefore, \( I_2 \) is not minimal.

Our guess is that \( I_3 = I_{\text{first}} \).
Example
Let $X$ be the set with members that are sets:

$$X = \{A_i\}_{i=0}^k - \forall i : A_i \text{ set.}$$

Furthermore, we define:

- $\bigcup X = \{a : a \in A_i \text{ and exists } A_i \in X\}$.
- $\bigcap X = \{a : a \in A_i \text{ for every } A_i \in X\}$.

Example

$$X = \{\{1, 2, 3\}, \{5, 6\}, \{4\}\}.$$  
$\bigcup X = \{1, 2, 3, 4, 5, 6\}.$  
$\bigcap X = \emptyset.$

We define for every $i \in \mathbb{N}$:

$$A = \{A_i\}_{i=0}^\infty.$$  
$A_i = \{1, 2, 3, \ldots, i\}.$

Questions

- What is: $A_0$? $A_0 = \emptyset$.
- What is: $A_5$? $A_5 = \{1, 2, 3, 4, 5\}$.
- Is: $\{1, 2, 3\} \in A$? $A_3 = \{1, 2, 3\} \in A$.
- Is: $\{2, 3, 4\} \in A$? No.
- Is: $\{1, 2, 3\} \subseteq A$? No, because for instance, $1 \notin A$.
- What is $\bigcup A$? $\mathbb{N}\setminus \{\emptyset\}$.
- What is $\bigcap A$? $\{1\}$.
Prove that $\cup A = \mathbb{N} \setminus \{\emptyset\}$?

The idea of the proof is to use bi-directional subsetting, beginning with the definition that for every $i \in \mathbb{N}$:

$$B_i = \{i \cdot n : n \in \mathbb{N}\}$$

**Questions**

- What is $B_3$? $\{0, 3, 6, 9, \ldots\}$.

- What is $\cup_{i=0}^{\infty} B_i$? $\mathbb{N}$.

- What is $\cap_{i=0}^{\infty} B_i$? $\{0\}$.

- What is $\cap_{i \in \mathbb{N}} (B_i \setminus \{0\})$? $\emptyset$.

**Exercise Summary**

- Take a test case example of a small simple computer program.

- Determine the differences between syntax and semantics.

- State the definition and characteristics of Power Sets.
18 Exercise - Home Exercise #1

Home Exercise #1 includes three topics:

- MI/MU example on induction.
- Propositional calculus.
- From simple code snippet to inductive sets.

18.1 Inductive Sets, MI/MU Example

The set of atoms is $A = \{MI\}$.

The operations $P$ are detailed below.

- $XI \rightarrow XIU$ – add a $U$ at the end of any string ending with $I$.
- $MX \rightarrow MXX$ – duplicate everything that comes after an $M$.
- $III \rightarrow U$ – take any sequence of three consecutive $I$’s and change them into a single $U$.
- $UU \rightarrow \text{’nothing’}$ – omit two consecutive $U$’s.

Some elements of $I(A, P)$ are

- $MI \rightarrow \text{rule 2} \rightarrow MII$.
- $MII \rightarrow \text{rule 2} \rightarrow MIII$.
- $MIII \rightarrow \text{rule 1} \rightarrow MIIIU$.
- $MIIII \rightarrow \text{rule 3} \rightarrow MIU$. 
MII, MIII, MIIIU, and MIU are all in I(A, P).

The Elements of I(A, P)

We start with the Atom: $A = \{MI\}$ and apply these rules:

- #1 : $MI \xrightarrow{1} MIU$.
- #2 : $MI \xrightarrow{2} MII$.
- #3 : $MI \xrightarrow{3} MI$.
- #4 : $MI \xrightarrow{4} MI$.

When applying all the possible rules on the member MIU:

- #1 : $MIU \xrightarrow{1} MIUU$.
- #2 : $MIU \xrightarrow{2} MIUIU$.
- #3 : $MIU \xrightarrow{3} MIU$.
- #4 : $MIU \xrightarrow{4} MIU$.

When applying all the possible rules on the member MII:

- #1 #1 : $MII \xrightarrow{1} MIU$.
- #1 #2 : $MII \xrightarrow{2} MIII$.
- #1 #3 : $MII \xrightarrow{3} MII$.
- #1 #4 : $MII \xrightarrow{4} MII$. 

170
We quickly realize that the set grows as we activate the operations, and the number of members in $I(A, P)$ is $\infty$.

**Problem**

Prove that $MU \not\in I(A, P)$, or in other words, that $MU$ is not in the inductive set defined by the atom $MI$ and the operations defined above. Another interpretation of the question is that if we start with the atom $MI$, and repeatedly activate the operations defined by $P$, in any possible order, we will never reach $MU$.

**Proof**

We prove this claim using induction.

We prove that $MU \not\in I(A, P)$ using the claim that any member $w \in I(A, P)$ has a number of I’s that is not a multiple of three. If we can prove this claim, it proves that $MU$ is not in $I(A, P)$ since the number of I’s in $MU$ (zero I’s) is a multiple of three.

To prove using induction:

1. We prove for the atoms.

2. We assume the claim is true for any member $w$ in the inductive set, and show that the claim stays true after applying any of the operations on $w$. We term this part of the proof as the *inductive step*.

We begin by proving for the atoms. We have a single atom in our example—$MI$—which has one I. One is not a multiple of three and therefore our claim holds.

Next, we assume that for any $w \in I(A, P)$, the number of I’s in $w$ is not a multiple of three. We need to show that the claim remains true after acti-
vating any of the operations. To prove the claim, we iterate all the possible
operations, and show that the claim is true for each possible operation:

- For the first rule, XI → XIU: adding a U at the end of a word does
  not change the number of I’s in the word. Since we assume that any
  word w does not have a number of I’s that is a multiple of three before
  activating the operation, the number of I’s after the operation remains
  not a multiple of three.

- For the second rule, MX → MXX: the number of I’s is multiplied by
  two. Since the number of I’s in any word w before the operation was
  not a multiple of three, we can mark the number of I’s in w—before
  the operation—as either 3*k + 1 or 3*k + 2 (for a natural number k).

  - If the number of I’s in w was 3*k + 1, then the number of I’s after
    applying the operation is multiplied by two: 2(3*k + 1) = 6*k +
    2, which is not a multiple of three.

  - If the number of I’s in w was 3*k + 2, then the number of I’s after
    applying the operation is multiplied by two: 2(3*k + 2) = 6*k +
    4 = 3(2*K + 1) + 1. Since k is a natural number, 2*k + 1 is a
    natural number, and 3(2*k + 1) + 1 is not a multiple of three.

- For the third rule: III → U, the number of I’s was reduced by three
  (three I’s were replaced by a U). If the number of I’s in w before the
  operation was not a multiple of three, after applying the operation, the
  number of I’s is reduced by three, and remains not a multiple of three.

- For the fourth rule, UU → ’nothing’: omitting two U’s does not change
the number of I’s in the word. Since we assumed that any word w does not have a number of I’s which is a multiple of three before activating the operation, the number of I’s after the operation remains unchanged and is still not a multiple of three.

This completes the induction step and the proof.

Applying the operations in any way on MI results in MII and MIU.

Applying the operations in any way to MI, MII and MIU results in MII, MIUIU, and MIIU, and so on and so forth.

18.2 Propositional Calculus

Assuming p is a theorem and q is a theorem, then p and q are theorems.

We assume the following axioms:

- Rule #1: \( \neg \neg \) can be added or deleted in a theorem and a theorem is obtained.

- Rule #2: If p is a theorem, and q can be derived from p, then \( p \rightarrow q \) is a theorem (applies recursively).

- Rule #3: If p is a theorem and \( p \rightarrow q \) is a theorem, then q is a theorem.

- Rule #4: \( p \rightarrow q \) is a theorem if and only if \( \neg q \rightarrow \neg p \) is a theorem.

- Rule #5: \( \neg p \land \neg q \) is a theorem if and only if \( \neg (p \lor q) \) is a theorem.

- Rule #6: \( p \rightarrow q \) is a theorem if and only if \( \neg p \lor q \) is a theorem.

We can derive different theorems based on these operations.

Claim
Assuming $p$ is a theorem, $p \lor \neg p$ is a theorem.

Proof
According to the assumption, $p$ is a theorem.

- Using rule #1 - $\neg \neg p$ is a theorem.
- Using rule #2 - $p \rightarrow \neg \neg p$ is a theorem.
- Using rule #4 - $\neg \neg \neg p \rightarrow \neg p$ is a theorem.
- Using rule #1 - $\neg p \rightarrow \neg \neg \neg p$ is a theorem.
- Using rule #6 - $\neg \neg \neg \neg \neg p \lor \neg \neg \neg p$ is a theorem.

Home Exercise
2a. Prove that $p \land q \rightarrow q \land p$.
2b. Prove that $p \rightarrow (q \rightarrow (p \land q))$.

Propositional Calculus Example
Assuming $p$ is a theorem and $q$ is a theorem, then $p$ and $q$ are theorems.
We assume the axioms from the beginning of Section 18.2.
We can derive different theorems based on these operations.

Claim
Assuming $p$ is a theorem, prove that $p \land q \rightarrow q \land p$ is a theorem.

Proof
We assume $p \land q$.

- Using rule #1 we derive $\neg \neg p \land q$.
- Using rule #5 we derive $\neg (\neg p \lor \neg q)$.
Using rule #6 we derive \( \neg (p \rightarrow \neg q) \).

Using rule #4 we derive \( \neg (\neg q \rightarrow \neg p) \).

Using rule #6 we derive \( \neg (\neg \neg q \lor \neg p) \).

Using rule #1 we derive \( \neg (\neg q \lor \neg p) \).

Using rule #5 we derive \( \neg \neg q \land \neg \neg p \).

Using rule #1 we derive \( q \land \neg \neg p \).

Using rule #1 we derive \( q \land p \).

Using rule #2 - since we derived \( q \land p \) from \( p \land q \) then \( p \land q \rightarrow q \land p \) is a theorem.

Simple Code Snippet to Inductive Sets

\[ x = \sin(t); \]

\[ y = r^2 - 2; \]

\[ \text{while } (x > y) \{ \]
\[ \quad x = x + y + 3; \]
\[ \} \]

We number the rows:

1. \( x = \sin(t); \)

2. \( y = r^2 - 2; \)
3. while (x > y) {
4.     x = x + y + 3;
5. }

Let’s look at the lines that could affect the value of x:

(1, x, {1}).
(2, x, {1}).
(3, x, {1, 4}).
(4, x, {1}).

These are the lines that can affect the value of x. That is, the lines that can have an affect on the value of x when we reach them.

Our Atoms:

(1, x, {∅}).
(2, x, {∅}).
(3, x, {∅}).
(4, x, {∅}).

Our Operations:

(1, x, S) \xrightarrow{1} (1, x, S \cup \{1\}).
(2, x, S) \xrightarrow{2} (2, x, S).
(3, x, S_1), (2, x, S_2), (4, x, S_3) \xrightarrow{3} (3, x, S_1 \cup S_2 \cup S_3).
(4, x, S) \xrightarrow{4} (4, x, S \cup \{4\}).

The process stops after you activate the operations on a line containing (1, 4).

**Exercise**

1. Define the system and explain the purpose of using the inductive set, including computing I(A, P).
2. Define the same type of system for y.
3. Define a different program with x and y dependencies and compute I(A, P).

```java
x = 10;
y = 100;

while (x < 50) {
    x = x + y;
    y = y - x;
    while (y > 50) {
        x = x - y;
        y = y + x;
    }
}
```
18.3 Inductive Set in Computer Science

- `int x;`
- `rand(x);`
- `while (x > 0) {`
  - `print (x);`
  - `x--;`
- `}`

**Exercise**
What is the inductive set?

**Solution**
The operation P is subtraction, as long as the member is larger than zero.
A is the initial random number x.
The inductive set will be all the natural integer numbers from rand(x) and down to 1.

**Exercise**
Define using an inductive set all the possible answers.
19 Exercise - Z Language Specification

19.1 Z Language Specification - 'Dictionaries'

19.1.1 Purpose

Keep a list of terms and their translation to foreign languages. The system should support the following characteristics:

- For any given term in one language, provide a list of possible translations for any of the other languages.
- Provide hints to choose between the possible translations.
- Handle phrases.
- List terms in alphabetical order.
- Exclude non-legal words; e.g., 'qz', 'xk', 'jf', and 'www'.
- Support the addition of pairs to the list. Pairs consist of a term and a possible translation of the term.
- Support random testing of user vocabulary (considered a 'nice-to-have' feature).

We start by formally defining 'well formed' word pairs that meet the purpose guidelines.

19.1.2 Specification of Well-Formed Pairs of Words

We use the following definitions:
• Set of words: **Native**, **Foreign**, both defined as $\bigcup_k A^k$ where $A$ is the appropriate alphabet.

• **OrthoNative**: Set of Native.

• **OrthoForeign**: Set of Foreign.

• **WellFormedVocabs**: $\{ V : \text{set of (Native, Foreign)} : \forall n : \text{Native}; \forall f : \text{Foreign} \bullet (n, f) \in V \Rightarrow n \in \text{OrthoNative} \land f \in \text{OrthoForeign} \}$.

19.1.3 Operations

An operation is something that changes the model state (‘world’).

We define the following operations: addPair, ToForeign, and ToNative.

**Notation**

Set states before the operation are denoted by their names.

Set states after the operation are denoted by their names and an apostrophe, e.g. $V'$.

Input objects are denoted by ”?”.

Output objects are denoted by ”!”.

**addPair Operation**

\[
\text{addPair} == \{ V, V' : \text{set of (Native, Foreign)} ; n? : \text{Native}; f? : \text{Foreign} \bullet V \subseteq \text{WellFormedVocabs} \land (n?, f?) \in \text{WellFormedVocabs} \land V' = V \\
\lor \{(n?, f?)\} \}
\]

**ToForeign Operation**

\[
\text{ToForeign} == \{ V, V' : \text{set of (Native, Foreign)} ; n? : \text{Native}; \text{ftrans!} : \text{Foreign} \bullet V \subseteq \text{WellFormedVocabs} \land V' = V \land n? \in \text{OrthoNative} \land
\]

180
\[ ftrans! = \{ f? \in \text{Foreign} : (n?, f?) \in V \}. \]

Note that the translation leaves the list unchanged.

### 19.1.4 Invariants

An Invariant is a rule that is true for the initial state, and preserved by any of the possible operations applied on any possible input.

We define an initial state:

\[
\text{InitialState: InitVocab} == \{ V' \in (\text{Native, Foreign}) : V' = \emptyset \}. 
\]

**Claim**

The list \( V \) is always a subset of \textbf{WellFormedVocabs} (this is an example of an invariant).

**Proof**

We iterate the possible operations. Since the operations ToForeign and ToNative do not change \( V \) (the list of words), we need to only check the addPair operation:

\[
\text{addPair: } n? \in \text{Native} \land f? \in \text{Foreign} \land V' = V \cup \{(n?, f?)\} \land V \subseteq \textbf{WellFormedVocabs}. 
\]

According to the addPair operation definition and the induction hypothesis, \( V' \subseteq \textbf{WellFormedVocabs} \) and the proof is complete.

But what if the input was illegal?

### 19.1.5 Error Handling

Error handling should preserve the following characteristics:

- In the case of a pair \((n?, f?)\) that is not well-formed, the list should not change.
• The user should be informed of the error.

• We define this operation (which results in an error): addPairError ==

\{V, V' : set of (Native, Foreign) ; rep! : Message ; n? : Native; f? : Foreign • V \subseteq WellFormedVocabs \land (n?, f?) \in OrthoNative \land f? \in OrthoForeign) \land V' = V \land rep! = ErrorInInput\}.

19.1.6 'Total Operations'

Total operations preserve the following characteristics:

• Handle all cases and produce error and success messages as appropriate.

• A sample modification of addPair to inform the user of success:

\begin{align*}
\text{addPair} & = \{V, V' : \text{set of (Native, Foreign)} ; \text{rep! : Message} ; n? : \text{Native} ; f? : \text{Foreign} • V \subseteq \text{WellFormedVocabs} \land (n?, f?) \in \text{WellFormedVocabs} \land V' = V \cup \{(n?, f?)\} \land \text{rep!} = \text{OK}\} \\
\end{align*}

• TotalAddPair == addPair if (n?, f?) ∈ WellFormedVocabs; otherwise, TotalAddPair == addPairError.

Exercise Summary

• Reviewed Home Exercise #1.

• Examined a dictionaries example using the Z Language Specification.

• Defined specification of Well-Formed Pairs of Words.
20 Exercise - Defining a Specification

In this exercise we will review Home Exercise #2.

Home Exercise #2 will be submitted in pairs. The exercise is on an IBM tool named FoCuS (http://www.alphaworks.ibm.com/tech/focus).

Defining a Specification

We formally define "all the computers" using [NODES] and [{LIVE, DEAD}], defining for each computer if it is live or not.

Communication_Group is defined as the Cartesian Product of [NODES] and [{LIVE, DEAD}]. Formally this is written as: \( P(\text{NODES} \times \{\text{LIVE, DEAD}\}) \) : Communication_Group.

The Communication_Group preserves the following:

\[ \forall x = (n, s) \text{ such that } n \in \text{NODES} \text{ and } s \in \text{STATUS}, x \in \text{Communication}_\text{Group} \iff s = \text{LIVE}. \]

Next, we define the operations of joining and leaving the Communication_Group.

The knowledge we keep for each node, or our world of knowledge on nodes, is for each node if it is "LIVE" or "DEAD". We formally write this as:

World_State : \( P(\text{NODES} \times \{\text{LIVE, DEAD}\}) \) : World_State is a full function that maps NODES \( \implies \) \{LIVE, DEAD\}.

The definition of an invariant for the Communication_Group is as follows:

\[ \forall n \in \text{NODES} : \text{if } (n, s) \in \text{Communication}_\text{Group} \text{ then World}_\text{State}(n) = \text{LIVE}. \]

The next step we perform in the next exercise is to define operations, starting with addMember and removeMember.

Exercise Summary
• Heads up towards Home Exercise #2, that will be submitted in pairs.
  
  The exercise is on an IBM tool named FoCuS
  

• Defining specifications, using the Communication Group example.
21 Exercise - Communication Group Example

In this exercise we continue the Communication Group example discussed in the lecture and the previous exercise, and begin discussing the term functional coverage.

21.1 Communication Group Example, Revisited

To recap the Communication Group example:

- We start with two groups: [Nodes] - the set of computers in the network and [States = \{Up, Down\}] - an attribute state of each computer (node).

- The symbol "[]" marks Basic Types.

- We refer to basic types and the inductive set induced by the basic types as atoms, and the operations Cartesian Product and Power Set.

The purpose of a formal definition is to define a situation where only machines that are in the "Up" state can be connected in a Communication Group. Machines in the "Down" state cannot be in the Communication Group (which makes sense intuitively).

We model changes in states in the Communication Group (machines changing status from up to down, and vice versa) using operations.

For simplicity, we mark the [Nodes] basic type as N, and the [States] basic type as S.
\[
\text{Connected} : \mathbb{P}((\text{NXS}) \times (\text{NXS}))
\]

\[
\forall (n, s) : \text{NXS} \\land \\forall (n', s') : \text{NXS} \\
\quad ((n, s), (n', s')) \in \text{Connected} \rightarrow \\
\quad s = s' = Up
\]

Next, we define a function \(s\text{State}\) that maps each node to its appropriate state:

\[s\text{State} : \text{Nodes} \rightarrow \text{States}\]

In this case the function is a \textit{full function} because for each member \(n\) of the source (Nodes in our case) there is a single value in the target (States) that is \text{state}(n).

\[
s\text{State} : \mathbb{P}(\text{Nodes} \times \text{States})
\]

\[
(\forall n \in \text{Nodes}, \exists s \in \text{States} \\
\quad (n, s) \in s\text{State}) \land \\
(\forall n : N, \forall s_1 : S, \forall s_2 : S, \\
\quad (n, s_1) \in s\text{State} \land \\
\quad (n, s_2) \in s\text{State} \rightarrow \\
\quad s_1 = s_2)
\]

To verify that \(s\text{State} : \mathbb{P}(\text{Nodes} \times \text{States})\) is a type to which we are allowed to refer, we need to check that \(\mathbb{P}(\text{Nodes} \times \text{States})\) is a \textit{type}.

Nodes is a basic type.

States is also a basic type.

Nodes \times States is also a type, since Cartesian Product is an allowed operation.
\(P(\text{Nodes} \times \text{States})\) is also a type, since the Power Set is an allowed operation.

**Initial State**

We define the Initial State:

\[
\begin{align*}
\text{sState : } N & \rightarrow S \\
\text{Connected} & = \emptyset \\
\forall (n, s) & \in \text{sState}, \\
s & = \text{Down}
\end{align*}
\]

The initial state is that all computers (nodes) are in the "Down" state.

Next, we model the operations and changes that can occur in the system, starting with the Fail operation:

\[
\begin{align*}
\text{Fail} \\
((n_1, s_1), (n_2, s_2)) : & \ P((\text{NXS}) \times (\text{NXS})) \\
(n_1, s_1), (n_2, s_2) & \in \text{Connected} \land \\
\text{Connected}' & = \text{Connected} - \{(n_1, s_1), (n_2, s_2)\}
\end{align*}
\]

\[
\begin{align*}
\text{Fail\_Node} \\
n_? & \in \text{Nodes} \\
\text{sState}(n?) & = \text{Up} \land \\
\text{sState}'(n?) & = \text{Down} \land \\
\forall n^? & \in \text{Nodes}, \\
\forall s^? & \in \text{States}, \\
((n?, s), (n^?, s^?)) & \in \text{Connected} \rightarrow \\
\text{Connected}' & = \text{Connected} - \{((n?, s), (n^?, s^?))\} \land \\
((n^?, s^?), (n?, s)) & \in \text{Connected} \rightarrow \\
\text{Connected}' & = \text{Connected} - \{((n^?, s^?), (n?, s))\}
\end{align*}
\]
We use assisting definitions of sets A and B:

\[
A = \{(n?, s), (n\sim, s\sim) \mid \mathbb{P}((N \times S) \times (N \times S)) \land ((n?, s), (n\sim, s\sim)) \in \text{Connected}\} \land \\
B = \{((n\sim, s\sim), (n?, s)) \mid \mathbb{P}((N \times S) \times (N \times S)) \land ((n\sim, s\sim), (n?, s)) \in \text{Connected}\} \land \\
\]

We can then use the sets A and B to write the Fail_Node operation in a simpler way, as Connected’ = Connected - (A \cup B).

\[
\begin{array}{ll}
\text{Node\_Back} \\
n? : \text{Nodes} \\
s\text{State’}(n?) = \text{Up} \land \\
\text{Connected’} = \text{Connected}
\end{array}
\]

\[
\begin{array}{ll}
\text{Connection\_Back} \\
((n, s), (n\sim, s\sim)) : \mathbb{P}((N\times S) \times (N\times S)) \\
s\text{State}(n) = \text{Up} \land \\
s = \text{Up} \land \\
s\text{State}(n\sim) = \text{Up} \land \\
s\sim = \text{Up} \land \\
\text{Connected’} = \text{Connected} \cup \{((n, s), (n\sim, s\sim))\}
\end{array}
\]

We find that we can also write the Connection\_Back operation in a simpler but equivalent way:

\[
\begin{array}{ll}
\text{Connection\_Back} \\
n : \text{Nodes}, n\sim : \text{Nodes} \\
s\text{State}(n) = \text{Up} \land \\
s\text{State}(n\sim) = \text{Up} \land \\
\text{Connected’} = \text{Connected} \cup \{((n, \text{Up}), (n\sim, \text{Up}))\}
\end{array}
\]
So far, we discussed a communication group modeled by a *directed* graph type; that is, there is an explicit direction between the nodes defining a source node and a target node.

In the following definitions we define the equivalent definitions for a non-directed graph model of the communication group example.

\[
\text{Connected} : \mathcal{P}(\mathcal{P}(\text{Nodes}))
\]

\[
\forall x \in \text{Connected},
| x | = 2 \land
\forall y \in x,
\text{sState}(y) = \text{Up}
\]

The Initial State is defined as:

\[
\forall n \in \text{Nodes}, \text{sState}(n) = \text{Down} \land \text{Connected} = \emptyset.
\]

\[
\text{Down}_\text{Node}
\]

\[
\begin{array}{l}
n? : \text{Nodes} \\
\text{sState}(n?) = \text{Up} \land \\
\text{sState'}(n?) = \text{Down} \land \\
\forall \{n?, n\} : \mathcal{P}(\mathcal{P}(\text{Nodes})) \\
\{n?, n\} \in \text{Connected} \rightarrow \{n?, n\} \notin \text{Connected'}
\end{array}
\]

\[
\text{Up}_\text{Node}
\]

\[
\begin{array}{l}
n? : \text{Nodes} \\
\text{sState'}(n?) = \text{Up} \land \\
\text{Connected'} = \text{Connected}
\end{array}
\]
### Up\_Connection

\[
\begin{array}{l}
\text{n}_1 : \text{Nodes, n}_2 : \text{Nodes} \\
\text{sState}(n_1) = \text{Up} \land \\
\text{sState}(n_2) = \text{Up} \land \\
\text{Connected}' = \text{Connected} ... \\
\end{array}
\]

### Down\_Connection

\[
\begin{array}{l}
\text{n}_1 : \text{Nodes, n}_2 : \text{Nodes} \\
\text{sState}(n_1) = \text{Up} \land \\
\text{sState}(n_2) = \text{Up} \land \\
\text{Connected}' = \text{Connected} - ... \\
\end{array}
\]

## 21.2 Functional Coverage

**Definition**

Functional coverage is a method that systematically creates functional coverage models, each containing a large sets of verification (coverage) tasks, and monitors the testing to see if these tasks were performed.

Functional coverage models come in many flavors:

- Models can cover the inputs and outputs of a program.
- Models can model the internal state of the program (e.g., values of variables).
- Snapshot models that model the state of the program at a certain point in time.
- Temporal models, dealing with scenarios.
• Usually, functional coverage models involve modeling several properties in parallel.

• Models can be based on the specifications of the application (Black-Box).

• Models can also be derived from implementation (WhiteBox).

The first and most important step in the functional coverage process is deciding what to cover or, more precisely, on what coverage models to measure coverage. To make coverage successful and use it to influence the testing process, it is important to create good coverage models, using the following guidelines:

1. It is important to choose coverage models for areas which the user thinks are risky or error prone.

2. The size of the model should be chosen in accordance with the testing resources. The model size (number of tasks) should not be too large, making it impossible to cover the model, given the testing time and resources available.
22 Exercise - Functional Coverage Models

In this exercise we begin with an example of analyzing a system for extracting a functional coverage model.

Example

Consider the following system (see Figure 5).

![Figure 5: Sample front end/back end System](image)

The system includes a front end and a back end, each with two components termed Software and OS.

Exercise

Analyze the system in terms of bad-path testing.

Bad-path testing relates to the options that can go wrong, and how the system handles recovery from these errors. Figure 6 displays the transactions...
status values and possible failure reasons.

Figure 6: Transactions status and failure reasons

When analyzing the status of the system during the failure, we see that the system deals with transactions, that can be at one of a given six possible status options (when the system fails):

1. TransactionState1
2. TransactionState2
3. TransactionState3
4. TransactionState4
5. TransactionState5
6. NoTransaction
Each of the four components can fail due to a given predefined reason. The OS component can fail due to one of three given reasons:

1. NoFailure
2. Failure1
3. Failure2

The Software component can fail due to one of four given reasons:

1. NoFailure
2. Failure1
3. Failure2
4. Failure3

Several components can fail, each for any given reason. Each of the failing components then needs to be brought back up (only the failing components). The order in which they are brought up is also important.

In testing, we aim to test all the possible options. When we try to test all possible options of all failure types, all the order types, and so on, we reach a very large number. When considering that setting up, analyzing, and testing each option takes time, we conclude that an exponential time frame is needed for testing.

To overcome the time obstacle, we relax some of the constraints. For instance, we decide to cover each pair of test options. Figure 7 displays the combinatorial test design report results.
Figure 7: Combinatorial test design report

<table>
<thead>
<tr>
<th>frontSoftware</th>
<th>backSoftware</th>
<th>frontOS</th>
<th>backOS</th>
<th>recoverFirst</th>
<th>recoverSecond</th>
<th>recoverThird</th>
<th>recoverFourth</th>
</tr>
</thead>
<tbody>
<tr>
<td>Failure3</td>
<td>Failure3</td>
<td>Failure1</td>
<td>Failure2</td>
<td>backOS</td>
<td>frontSoftware</td>
<td>backSoftware</td>
<td>frontOS</td>
</tr>
<tr>
<td>notFailed</td>
<td>notFailed</td>
<td>Failure1</td>
<td>notFailed</td>
<td>frontOS</td>
<td>none</td>
<td>none</td>
<td>none</td>
</tr>
<tr>
<td>Failure1</td>
<td>Failure1</td>
<td>Failure2</td>
<td>backOS</td>
<td>frontSoftware</td>
<td>backSoftware</td>
<td>frontOS</td>
<td>backOS</td>
</tr>
<tr>
<td>notFailed</td>
<td>Failure1</td>
<td>notFailed</td>
<td>backOS</td>
<td>none</td>
<td>none</td>
<td>none</td>
<td>none</td>
</tr>
<tr>
<td>Failure3</td>
<td>notFailed</td>
<td>notFailed</td>
<td>backOS</td>
<td>frontSoftware</td>
<td>frontSoftware</td>
<td>none</td>
<td>none</td>
</tr>
<tr>
<td>notFailed</td>
<td>Failure2</td>
<td>notFailed</td>
<td>backOS</td>
<td>frontSoftware</td>
<td>backSoftware</td>
<td>frontOS</td>
<td>backOS</td>
</tr>
<tr>
<td>Failure2</td>
<td>Failure3</td>
<td>Failure2</td>
<td>backOS</td>
<td>frontSoftware</td>
<td>backSoftware</td>
<td>frontOS</td>
<td>backOS</td>
</tr>
<tr>
<td>notFailed</td>
<td>Failure1</td>
<td>notFailed</td>
<td>backOS</td>
<td>frontSoftware</td>
<td>frontSoftware</td>
<td>backSoftware</td>
<td>frontOS</td>
</tr>
<tr>
<td>Failure1</td>
<td>Failure1</td>
<td>Failure1</td>
<td>backOS</td>
<td>frontSoftware</td>
<td>backSoftware</td>
<td>frontOS</td>
<td>backOS</td>
</tr>
<tr>
<td>notFailed</td>
<td>Failure2</td>
<td>notFailed</td>
<td>backOS</td>
<td>frontSoftware</td>
<td>backSoftware</td>
<td>frontOS</td>
<td>backOS</td>
</tr>
<tr>
<td>notFailed</td>
<td>Failure2</td>
<td>Failure2</td>
<td>backOS</td>
<td>frontSoftware</td>
<td>backSoftware</td>
<td>frontOS</td>
<td>backOS</td>
</tr>
<tr>
<td>notFailed</td>
<td>Failure2</td>
<td>Failure2</td>
<td>backOS</td>
<td>frontSoftware</td>
<td>backSoftware</td>
<td>frontOS</td>
<td>backOS</td>
</tr>
</tbody>
</table>
23 Exercise - FoCuS Models

In this exercise we learn how to define attributes and constraints in FoCuS.

23.1 Defining Attributes

- frontSoftware = \{"Failure1", "Failure2", "Failure3", "NotFailed"\}.
- backSoftware = \{"Failure1", "Failure2", "Failure3", "NotFailed"\}.
- frontOS = \{"Failure1", "Failure2", "NotFailed"\}.
- backOS = \{"Failure1", "Failure2", "NotFailed"\}.
- recoverFirst = \{"frontSoftware", "backSoftware", "frontOS", "backOS", "none"\}.
- recoverSecond = \{"frontSoftware", "backSoftware", "frontOS", "backOS", "none"\}.
- recoverThird = \{"frontSoftware", "backSoftware", "frontOS", "backOS", "none"\}.
- recoverFourth = \{"frontSoftware", "backSoftware", "frontOS", "backOS", "none"\}.

23.2 Defining Constraints

- No Failure Constraint: frontSoftware.equals("NotFailed") && backSoftware.equals("NotFailed") && frontOS.equals("NotFailed") && backOS.equals("NotFailed").
• Can’t Restart a Not Killed Component Constraint: \( \text{frontSoftware}.equals(”NotFailed”) \) 
  \&\& (\( \text{recoverFirst}.equals(”frontSoftware”) \) || \( \text{recoverSecond}.equals(”frontSoftware”) \) 
  || \( \text{recoverThird}.equals(”frontSoftware”) \) || \( \text{recoverFourth}.equals(”frontSoftware”) \)).

• Another Can’t Restart a Not Killed Component Constraint: \( \text{backSoftware}.equals(”NotFailed”) \) 
  \&\& (\( \text{recoverFirst}.equals(”backSoftware”) \) || \( \text{recoverSecond}.equals(”backSoftware”) \) 
  || \( \text{recoverThird}.equals(”backSoftware”) \) || \( \text{recoverFourth}.equals(”backSoftware”) \)).

• Another Can’t Restart a Not Killed Component Constraint: \( \text{frontOS}.equals(”NotFailed”) \) 
  \&\& (\( \text{recoverFirst}.equals(”frontOS”) \) || \( \text{recoverSecond}.equals(”frontOS”) \) 
  || \( \text{recoverThird}.equals(”frontOS”) \) || \( \text{recoverFourth}.equals(”frontOS”) \)).

• Another Can’t Restart a Not Killed Component Constraint: \( \text{backOS}.equals(”NotFailed”) \) 
  \&\& (\( \text{recoverFirst}.equals(”backOS”) \) || \( \text{recoverSecond}.equals(”backOS”) \) 
  || \( \text{recoverThird}.equals(”backOS”) \) || \( \text{recoverFourth}.equals(”backOS”) \)).

• Can’t Restart A Duplicate Component Constraint: \( \text{recoverFirst}.equals(\text{recoverSecond}) \) 
  || \( \text{recoverFirst}.equals(\text{recoverThird}) \) || \( \text{recoverFirst}.equals(\text{recoverFourth}) \).

• Another Can’t Restart A Duplicate Component Constraint: !(\( \text{recoverSecond}.equals(”none”) \)) 
  \&\& (\( \text{recoverSecond}.equals(\text{recoverFirst}) \) || \( \text{recoverSecond}.equals(\text{recoverThird}) \) || \( \text{recoverSecond}.equals(\text{recoverFourth}) \)).

• Another Can’t Restart A Duplicate Component Constraint: !(\( \text{recoverThird}.equals(”none”) \)) 
  \&\& (\( \text{recoverThird}.equals(\text{recoverFirst}) \) || \( \text{recoverThird}.equals(\text{recoverSecond}) \) || \( \text{recoverThird}.equals(\text{recoverFourth}) \)).
• Another Can’t Restart A Duplicate Component Constraint: !(recoverFourth.equals("none")) && (recoverFourth.equals(recoverFirst) || recoverFourth.equals(recoverSecond) || recoverFourth.equals(recoverThird)).

• Duplicate ”None“ permutations Constraints: recoverSecond.equals("none") && !(recoverThird.equals("none")).

• Another Duplicate ”None“ permutations Constraints: recoverThird.equals("none") && !(recoverFourth.equals("none")).

• ”none“ - 2 NotFailed - Constraints: ((frontSoftware.equals("NotFailed"))
&& (backSoftware.equals("NotFailed")) && !(frontOS.equals("NotFailed"))
&& !(backOS.equals("NotFailed"))) && (recoverSecond.equals("none")).

• Another ”none“ - 2 NotFailed - Constraints: ((frontSoftware.equals("NotFailed"))
&& !(backSoftware.equals("NotFailed")) && (frontOS.equals("NotFailed"))
&& (backOS.equals("NotFailed"))) && (recoverSecond.equals("none")).

• Another ”none“ - 2 NotFailed - Constraints: ((frontSoftware.equals("NotFailed"))
&& !(backSoftware.equals("NotFailed")) && !(frontOS.equals("NotFailed"))
&& (backOS.equals("NotFailed"))) && (recoverSecond.equals("none")).

• Another ”none“ - 2 NotFailed - Constraints: (!(frontSoftware.equals("NotFailed"))
&& (backSoftware.equals("NotFailed")) && (frontOS.equals("NotFailed"))
&& !(backOS.equals("NotFailed"))) && (recoverSecond.equals("none")).

• Another ”none“ - 2 NotFailed - Constraints: (!(frontSoftware.equals("NotFailed"))
&& (backSoftware.equals("NotFailed")) && !(frontOS.equals("NotFailed"))
&& !(backOS.equals("NotFailed"))) && (recoverSecond.equals("none")).
• Another "none" - 2 NotFailed - Constraints: (!frontSoftware.equals("NotFailed")) && !backSoftware.equals("NotFailed") && (frontOS.equals("NotFailed")) && (backOS.equals("NotFailed")) && (recoverSecond.equals("none")),
• "none" - 1 NotFailed Constraint: (frontSoftware.equals("NotFailed")) && !backSoftware.equals("NotFailed") && !frontOS.equals("NotFailed") && !backOS.equals("NotFailed") && (recoverSecond.equals("none")) || recoverThird.equals("none")),
• Another "none" - 1 NotFailed Constraint: (!frontSoftware.equals("NotFailed")) && backSoftware.equals("NotFailed") && !frontOS.equals("NotFailed") && !backOS.equals("NotFailed") && (recoverSecond.equals("none")) || recoverThird.equals("none")),
• Another "none" - 1 NotFailed Constraint: (!frontSoftware.equals("NotFailed")) && !backSoftware.equals("NotFailed") && (frontOS.equals("NotFailed")) && !backOS.equals("NotFailed") && (recoverSecond.equals("none")) || recoverThird.equals("none")),
• Another "none" - 1 NotFailed Constraint: (!frontSoftware.equals("NotFailed")) && !backSoftware.equals("NotFailed") && !frontOS.equals("NotFailed") && (backOS.equals("NotFailed")) && (recoverSecond.equals("none")) || recoverThird.equals("none")),
• "none" - 0 Not Killed - Constraint: (!frontSoftware.equals("NotFailed")) && !backSoftware.equals("NotFailed") && !frontOS.equals("NotFailed") && !backOS.equals("NotFailed") && (recoverSecond.equals("none")) || recoverThird.equals("none")) || recoverFourth.equals("none")) .

199
24 Exercise - Home Exercise #2

In this exercise we discuss Home Exercise #2 on FoCuS and functional coverage, including the uses of the cartesian product and combinatorial test design reports in FoCuS.

Example
Consider a system that includes a Client and a Server. On each (Client/Server) there’s a communication table. The table can be in a corrupted state or a non-corrupted state. On each system (Client/Server), the communication table can be in a state of being copied to the disk, or not. Also, on each system (Client/Server), the communication table can be in a state of handling IO operations, or not. And finally, on each system (Client/Server), there could be a correction operation already running in parallel, or not.

A correction operation can be activated on the Client or the Server. The purpose of the correction operation is to override a corrupted communication table with a non-corrupted table. The correction operation performs the correction only if possible; for instance, if the table on the Server is corrupted, and the table on the Client it is not corrupted, the table from the Client can be used to override the corrupted table on the Server. But if both tables are corrupted, a correction cannot be performed.

Between the Client and Server are active transactions. Each transaction can be in one of five states, defined as Initiate, Rollback, Commit, Doubt, and None. The case in which no transaction is communicated between the Client and Server (state: None) is not interesting. On the other hand, we are able to generate 50 transactions in parallel in each possible state of Client and
Server, in any transaction state we define.

When activating the correction operation, we would like to receive one of the following results:

- Successfully Corrected.
- Correction Already In Progress.
- Try Later (IO or Copy in progress on either Client or Server).
- Not Corrected (both copies are corrupted).
- Nothing to do (both copies are not corrupted).

Exercises

1. Define an INPUT/OUTPUT model in FoCuS.

2. Define a CTD model in which each pair of options of IO state and table copied to disk state are covered at least once.
25 Exercise - Home Exercise #3

In this exercise we cover questions towards Home Exercise #3 and summarize the course (going over what we did in the lecture).

Exercise

Given $A \subseteq B \subseteq C$, prove that $A \cup B = B \cap C$.

Remark: why do we use this type of theorem proofing? Because if we make it just a bit more formal, a computer could read and analyze it.

Proof

When we try to picture a diagram of the groups, we realize that since $A \subseteq B \subseteq C$, then $A \cup B = B$ and $B \cap C = B$, which when combined imply that $A \cup B = B = B \cap C$.

To prove that $A \cup B = B \cap C$, we actually need to prove two claims:

- $A \cup B = B$.
- $B = B \cap C$.

Claim #1: $A \cup B = B$.

(Later, to complete the proof, we will also define and prove the complementary Claim #2 $B = B \cap C$).

Usually when we need to prove that there is an equality between two expressions, we use bi-directional subsetting, that is, we prove (Claim #1a): $A \cup B \subseteq B$, and then (Claim #1b): $B \subseteq A \cup B$. If we prove the two, then we can conclude that $A \cup B = B$.

How do we prove Claim #1a: $A \cup B \subseteq B$?

We select any member of $A \cup B$ and show that this member must also be a
Starting with a member \( x \in A \cup B \), by the definition of the union \( \cup \) operator, it implies that \( (x \in A) \cup (x \in B) \).

Using the definition of subset \( \subseteq \), \( (x \in A) \land (A \subseteq B) \Rightarrow (x \in B) \).

Starting with \( (x \in A) \), we derive that \( (x \in B) \), denoted in formal writing as \( (x \in A) \rightarrow (x \in B) \).

From \( ((x \in A) \lor (x \in B)) \) we derive that \( (x \in B) \).

We start with \( (x \in A \cup B) \) and show that we can derive \( (x \in B) \) from \( (x \in A) \), and we also know that \( (x \in A) \lor (x \in B) \), so how do we derive that \( (x \in B) \)?

If we mark \( p \equiv (x \in A) \) and we mark \( q \equiv (x \in B) \), we can say that

\[
[((x \in A) \lor (x \in B)) \land ((x \in A) \rightarrow (x \in B))] \rightarrow (x \in B) \lor (x \in B)
\]

is like saying that

\[
[(p \lor q) \land (p \rightarrow q)] \rightarrow q \lor q.
\]

From \( q \lor q \) to derive \( q \) we use a truth table (see Table 19).

<table>
<thead>
<tr>
<th>( q )</th>
<th>( q \lor q )</th>
<th>( q \rightarrow q \lor q )</th>
</tr>
</thead>
<tbody>
<tr>
<td>T</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>F</td>
<td>F</td>
<td>T</td>
</tr>
</tbody>
</table>

The same method can be used to prove that \( [(p \lor q) \land (p \rightarrow q)] \rightarrow q \lor q \) (see Table 20).

This method enables us to conclude that \( (x \in B) \) from \( (x \in A \cup B) \), which is what we wanted to prove in Claim #1a.

Claim #1b is that for any member \( (x \in B) \), we derive that \( (x \in A \cup B) \).

Proof of Claim #1b
Table 20: Truth Table for Proof

<table>
<thead>
<tr>
<th>p</th>
<th>q</th>
<th>p ∨ q</th>
<th>p → q</th>
<th>(p ∨ q) ∧ (p → q)</th>
<th>q ∨ q</th>
<th>(p ∨ q) ∧ (p → q) → q ∨ q</th>
</tr>
</thead>
<tbody>
<tr>
<td>T</td>
<td>F</td>
<td>T</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>T</td>
</tr>
<tr>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>T</td>
</tr>
<tr>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>F</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
</tr>
</tbody>
</table>

We start with \((x \in B)\). We prove that we can derive \(((x \in B) \lor (x \in A))\), which by the definition of \(\cup\) is equivalent to \(x \in (A \cup B)\).

We need to prove that we are allowed to derive \(((x \in B) \lor (x \in A))\) from \((x \in B)\).

Marking \(q\) as \((x \in B)\) and \(p\) as \((x \in A)\) we need to prove that we are allowed to derive \(q \rightarrow (p \lor q)\).

From \(q\) to derive \(p \lor q\) we use a truth table (see Table 21).

Table 21: Derivation Truth Table

<table>
<thead>
<tr>
<th>q</th>
<th>p</th>
<th>p ∨ q</th>
<th>q → (p ∨ q)</th>
</tr>
</thead>
<tbody>
<tr>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>F</td>
<td>F</td>
<td>F</td>
<td>T</td>
</tr>
<tr>
<td>T</td>
<td>F</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>F</td>
<td>T</td>
<td>T</td>
<td>T</td>
</tr>
</tbody>
</table>

Reminder

Truth table for \(\rightarrow\) is: "if then" relationship: \(p \rightarrow q\) (see Table 22).

Table 22: If Then Truth Table

<table>
<thead>
<tr>
<th>p</th>
<th>q</th>
<th>p → q</th>
</tr>
</thead>
<tbody>
<tr>
<td>T</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>T</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>F</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>F</td>
<td>F</td>
<td>T</td>
</tr>
</tbody>
</table>
This completes the proof of Claim #1b, and along with Claim #1a, it completes the proof of Claim #1.

Proof of Claim #2 is very similar.
26 Exercise - Defining a Library Example

In this exercise we cover

- Function Types.
- Questions towards Home Exercise #3 and the final test.
- A summary of the course (recalling what we did in the lectures).

26.1 Functions

Definition: A function is a relation that maps at most one single member in the destination set for each member in the source set.

Example: \text{IsIn} == \{\text{London} \mapsto \text{England}, \text{Oxford} \mapsto \text{England}, \text{Bonn} \mapsto \text{Germany}, \text{Paris} \mapsto \text{France}\}.

The function IsIn is of type Cities \rightarrow Countries, and it maps certain cities to the countries in which they reside.

\text{dom(IsIn)} = \{\text{London, Oxford, Bonn, Paris}\}.

\text{ran(IsIn)} = \{\text{England, Germany, France}\}.

Formal Definition: A relation \( R \) of type \( X \leftrightarrow Y \) is a function if and only if:

\[ \forall x : X; y, z : Y \bullet (x \mapsto y) \in R \land (x \mapsto z) \in R \Rightarrow y = z. \]

In words this definition says that if \( x \) is mapped to \( y \), and \( x \) is mapped to \( z \), then \( y \) and \( z \) must be the same member.

Partial Function

Definition: A function where the domain is a subset of the source.

We mark \( f : X \mapsto Y \).

\text{dom}(f) \subset X.
Example: The function IsIn is a partial function because we see that \( \text{dom}(\text{IsIn}) \)
is a subset (not equal) to the source set Cities (all the cities).

Formal Definition: \( X \twoheadrightarrow Y = \{ R : X \leftrightarrow Y \mid \forall x : X; y, z : Y \circ x \mapsto y \in R \wedge x \mapsto z \in R \Rightarrow y = z \} \).

**Full Function**

Definition: A full function is a function whose source is equal to the source set.

\( f : X \rightarrow Y \), where \( \text{dom}(f) = X \).

Formal Definition: \( X \rightarrow Y = \{ f : X \leftrightarrow Y \mid \text{dom}(f) = X \} \).

Exercise: Given a function that maps an age to each person, is this definition a full function?

Solution: Yes, it defines a full function, because every person has an age.

Exercise: Is the ”IsIn” function, which maps the country to each city, a full function?

Solution: Yes, IsIn is a full function, because every city resides in a country.

**Bijective (also known as a 1-to-1) Function**

Definition: A function that for every member in the destination, there is at most one member in the source that maps to that member.

\( f : A \twoheadrightarrow B \) defines a full and bijective function.

\( f : A \mapsto B \) defines a partial and bijective function.

Formal Definition: \( A \twoheadrightarrow B = \{ f : A \rightarrow B \mid \forall a_1, a_2 : \text{dom}(f) \circ f(a_1) = f(a_2) \Rightarrow a_1 = a_2 \} \).

\( A \mapsto B = \{ f : A \rightarrow B \mid \forall a_1, a_2 : \text{dom}(f) \circ f(a_1) = f(a_2) \Rightarrow a_1 = a_2 \} \).

Example
Exercise: Is this function full/bijective/partial?

Solution: The function is full, because it is defined for any \( n \). The function is bijective, because it defines a single member in the source for any output \( n \), which is exactly the same \( n \).

Exercise: Explain why \((A \mapssto B) \cap (A \to B) = (A \mapssto B)\).

Surjective (also known as "on to") Function

Definition: A function that for every member in the destination, there is a member in the source that maps to that member.

\( f : A \to B \) defines a full and surjective function.

\( f : A \mapssto B \) defines a partial and surjective function.

Formal Definition: \( A \mapssto B == \{ f : A \to B \mid \text{ran}(f) = B \} \).

\( A \to B == \{ f : A \to B \mid \text{ran}(f) = B \} \).

Exercise: For each function, state its characteristics:

1. \( \{0 \mapssto 1, 1 \mapssto 2\} \).
2. \( \{0 \mapssto 1, 1 \mapssto 1, 2 \mapssto 1\} \).
3. \( \emptyset \).
4. \( \{0 \mapssto 0, 0 \mapssto 1, 0 \mapssto 2\} \).
5. \( \{0 \mapssto 1, 1 \mapssto 2, 2 \mapssto 0\} \).

Answers:

1. partial, bijective.
2. full.

3. ?

4. not a function...

5. full, bijective, surjective.

### 26.2 Library Definition From Lecture #12

We aim to define books and users who can loan books. We also aim to define the operations of loaning a book, returning a book, adding a user, removing a user, and so on.

We define the basic types [Books] and [Users].

We add a definition of a basic type: \(\text{bookState} = \{\text{Taken}, \text{notTaken}\}\).

We aim to represent all the books that a user holds.

\[ f : \text{[User]} \rightarrow \mathcal{P}(\text{books}). \]

All the taken books can then be defined as: \(\bigcup_{u \in \text{User}} f(u)\).

\(\text{inLib}: \) of type \(\mathcal{P}(\text{books})\) defines all the books in the library as all the non-taken books. All the books: \(\bigcup_{u \in \text{User}} f(u) \cup \text{inLib} = \text{Books}\).

We define an invariant: \(\emptyset = (\bigcup_{u \in \text{User}} f(u)) \cap \text{inLib} \land (\bigcup_{u \in \text{User}} f(u) \cup \text{inLib} = \text{Books})\).

**Exercise 1**

Write a \(\mathcal{Z}\) specification for a function called Return that returns a book to the library.

**Possible Solution**
Exercise 2
What can be said about the state of the library after an execution of two Taken and two Return functions?

Partial Solution
After a single Taken and Return of the same book, the library state doesn’t change.

After a Taken and Return, the number of books on the shelves won’t change.

Exercise 3
Can anything else be said about the state of the library?
Part III

Home Exercises

27 Home Exercise #1 - Induction and Propositional Calculus - Version 1

Mathematical Logic and its Applications to Computer Science

27.1 Induction Set

We introduce a single operation according to the following truth table (Table 23).

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>T</td>
<td>T</td>
<td>F</td>
</tr>
<tr>
<td>T</td>
<td>F</td>
<td>T</td>
</tr>
<tr>
<td>F</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>F</td>
<td>F</td>
<td>F</td>
</tr>
</tbody>
</table>

Atoms are a series of T and F; for instance, T T T F F F.

Example of a consecutive series of applying the relation:

T T F → (T T) F → F F → F

T T F → T (T F) → T T → F

Formally prove the following claims:

1.a. Any series of applications of the rule decreases the length of the string.
1.b. Given any finite string of T’s and F’s (for example, T T T F F F), the rule can only be applied a finite number of times.
1.c. The length of the string when stopping is exactly 1.

1.d. The order of activation of the rule is not important as it always reaches the same resulting string.
27.2 Propositional Calculus

We assume the following axioms:

- \(\neg\neg\) can be deleted in a theorem and the statement remains a theorem.
- If \(q\) can be derived from \(p\) then \(p \rightarrow q\) is a theorem (this applies recursively).
- If \(p\) is a theorem and \(p \rightarrow q\) is a theorem then \(q\) is a theorem.
- \(p \rightarrow q\) is a theorem if and only if \(\neg q \rightarrow \neg p\) is a theorem.
- \(\neg p \land \neg q\) is a theorem if and only if \(\neg (p \lor q)\) is a theorem.
- \(p \rightarrow q\) is a theorem if and only if \(\neg p \lor q\) is a theorem.
- \(\neg p \lor \neg q\) is a theorem if and only if \(\neg (p \land q)\) is a theorem.

2.a. Prove that \(p \land q \rightarrow q \land p\).
2.b. Prove that \((p \rightarrow q \rightarrow (p \land q)) \rightarrow \neg (p \land q \land \neg (p \land q))\).
2.c. Prove that \((p \rightarrow q) \rightarrow (\neg (\neg q \land p))\).
27.3 Simple Program

For each question, define the system (as explained in Exercise # 4) and compute I(A, P):

1. Given this program:

    \[ x = \sin(t); \]
    \[ y = r^2 - 2; \]
    
    while \((x > y)\) {
        \[ x = x + y + 3; \]
    }

Define the system for \(x\) and explain its purpose in words. Compute \(I(A, P)\).

2. Define for \(y\) and compute \(I(A, P)\).

3. Define the system for \(x\) and compute \(I(A, P)\) in the following program:

    \[ x = 10; \]
    \[ y = 100; \]
    
    while \((x < 50)\) {
        \[ x = x + y; \]
        \[ y = y - x; \]
        
        while \((y > 50)\) {
            \[ x = x - y; \]
        }
    }
\[ y = y + x; \]

\[
}\]
}
}
28 Home Exercise #1 - Induction and Propositional Calculus - Version 2

28.1 Induction Set

Given a single string of six digits (the Atom): '123456', and two operations:

1. Move the third digit to be the first digit

2. Move the sixth digit to be the fourth digit

For instance, applying the first operation on '123456' gives '312456'. Applying the second operation on '123456' gives '123645'.

1.a. Describe the Inductive Set that is created by the above two operations from the Atom '123456' (I(A, P)).

1.b. How many members are in the Inductive Set (|I(A, P)|)?

1.c. We add an additional operation, swapping the third and fourth digits. For instance, applying the third operation on '123456' gives '124356'. Answer the first two questions again, including the third operation.

1.d. Is it possible to reach the string '654321' with just the original first two operations? If it is possible, supply the derivation order. If not, prove it.
1.e. Answer 1.d. for all three operations.
28.2 Propositional Calculus

We assume the following axioms:

1. \( \neg \neg \) can be added or deleted in a theorem and a theorem is obtained.

2. If \( p \) is a theorem, and \( q \) can be derived from \( p \) then \( p \rightarrow q \) is a theorem (this applies recursively).

3. If \( p \) is a theorem and \( p \rightarrow q \) is a theorem then \( q \) is a theorem.

4. \( p \rightarrow q \) is a theorem if and only if \( \neg q \rightarrow \neg p \) is a theorem.

5. \( \neg p \land \neg q \) is a theorem if and only if \( \neg (p \lor q) \) is a theorem.

6. \( p \rightarrow q \) is a theorem if and only if \( \neg p \lor q \) is a theorem.

7. \( \neg p \lor \neg q \) is a theorem if and only if \( \neg (p \land q) \) is a theorem.

Assume \( p \) is a theorem and \( q \) is a theorem.

2.a. Prove that \( p \lor q \rightarrow q \lor p \).

2.b. Prove that \( (p \rightarrow q \rightarrow (p \land q)) \rightarrow (\neg ((p \land q) \land \neg (p \land q))) \).

2.c. Prove that \( (p \rightarrow q) \rightarrow (\neg (\neg (p \rightarrow q))) \).
28.3Simple Programs and Inductive Sets

Given this program:

read a;

read d;

sum = a;

read n;

for (i = 0; i < n; i++) {
    sum = sum + d;
}

1. Define an Inductive Set that characterizes the set of possible values of the variable 'sum', for each possible initial values of a, d, and n.

2. Prove by induction that the value of 'sum' for the $i^{th}$ iteration of the loop is $a + i \cdot d$.

3. What can you deduce from Question #2 about the value of 'sum' at the end of the loop?
Consider the following system:
System includes a Client and a Server. Each has a communication table. The table can be in a corrupted or non-corrupted state; in a state of being copied to the disk, or not; and in a state of handling IO operations, or not. Each Client/Server could be already a correction operation in parallel, or not.
A correction operation can be activated either on the Client or the Server. The purpose of the correction operation is to override a corrupted communication table with a non-corrupted table. The correction operation performs the correction operation if possible. For instance, if the table on the Server is corrupted, and on the Client it is not corrupted, the table from the Client can be used to override the corrupted table on the Server.
Between the Client and Server there are active transactions. Each transaction can be in one of five states: Initiate, Rollback, Commit, Doubt, and None. The case in which no transaction is communicated between the Client and Server (state: None) is not interesting. On the other hand, we are able to generate fifty transaction in parallel for each possible state of Client and Server, in any transaction state we define.
When activating the correction operation, we would like to receive one of the
following results:

- Successfully Corrected
- Correction Already In Progress
- Try Later (IO or Copy in progress on either client or server)
- Not Corrected (both copies are corrupted)
- Nothing to do (both copies are not corrupted)

Exercises

1. Define an INPUT/OUTPUT model in FoCuS.

2. Define a CTD model in which each pair of options of IO state and table copied to disk state are covered at least once.
We would like to test a network system (see Figure 8). The system includes 1-15 racks. Each rack includes 1-15 computers (Linux boxes). Each such Linux box has two network cards.

Each rack has an additional network component which we term "switch", that chooses at any given point in time, one network card from one of the boxes and a single rack, and provides network access for that card only. The box whose network card receives network access is termed "Master". All the other boxes are termed "Standby". Among the standby boxes, there...
are priorities that dictate which box will be the next one to receive network access.

The network system can also handle failures. We would like to test its ability to handle a single failure in up to five seconds, and two concurrent failures in up to fifteen seconds. The network system does not ensure a time frame for handling more than two parallel failures.

Each Linux box has a single module, which is running. The Master has one of the following modules running:

- ”Hello World”
- ”I am here”
- ”IO”
- ”Working Hard”
- ”Idle”

Each of the standby boxes has only one of two modules running:

- ”Hello World”
- ”I am here”

Each network failure can occur in a Linux Box (either Master or not, running any possible module), in the switch, or between the switches of the different racks.

1. Create a FoCuS Coverage module that describes this system. How many legal tasks are in the Cartesian Product?
2. Define a CTD model in which each pair of attributes is covered at least once.
31 Home Exercise #3 - Formal Specification

- Version 1

Mathematical Logic and its Applications to Computer Science

Home Exercise # 3 - Formal Specification

Define a formal specification of the possible operations \( \cap \), \( \cup \), and the inverse (complementary) on the set of natural numbers so that the result of the actions is a set with less than a thousand members, and every member in the resulting set is an even number.

Hints

Use \( \% \) for modulo

Use \( |x| \) for the size of set \( x \)

Use \( [\mathbb{N}] = [1, 2, 3, 4, ...] \) for the natural numbers
32 Home Exercise #3 - Formal Specification

- Version 2

Given a system of N hosts \{1, 2, 3, ..., n\}, each host has a state: "Running" or "Not Running". Communication is possible between every two hosts, if and only if both hosts are in the "Running" state. Furthermore, the communication is directed, and each host can communicate only with hosts that have a higher number, except for the first host who can receive communication from any other host.

Example: Host 2 can communicate with host 3 (but not vice versa), and hosts 7 and 8 can communicate with host 1 (assuming all these hosts are in the "Running" state).

1. Define a Formal Specification of the system, including any relation/function needed, an initial state, and an invariant, if applicable.

2. For each type used in the definition, supply a justification for being able to refer to this type.

3. Define operations for hosts changing states, and hosts connecting and disconnecting from one another.
Part IV

References

References

