Lecture 1

Introduction

Reasoning of our programs

- will be less intense than previous years
- Linking discrete maths to programming

Why do we want to reason about programs?

- We can't just write an infinite amount of test and use cases to make sure our program works
- We want to be sure our programs will always meet the requirements and will not fail on bound cases
- It's the next step in our programs to MEET REQUIREMENTS
- Provable behaviour
- Provable security
    - seL4 formally verified operating system (security enhancement level 4)
    - every function in the kernel has been formally verified to perform its task
    - therefore, we cannot manipulate the functions to exploit vulnerabilities
- Identify boundary cases and errors
    - Pentium floating point error (division errors) compounding errors
    - Intel had to recall many chips due to this error (intel had to invest in formal verification for their chips)
- Identify optimization
    - Example (if true: s; else: T) which simplifies to T
    - The T will use storage in memory, we can optimize the code to be written differently but we need to verify that behaviour is maintained when optimizing

How can this be done?

- Acquire and understand
    - Languages to formally specify systems
    - Structures to formally model systems
- Learn how to prove a program satisfies the requirements.
- We want to abstract away from the specifics of the language and break the problem down, we want to turn from the syntax and look at the functionality
- Proof is done through the combination of the first two

Why do we have to be so formal?

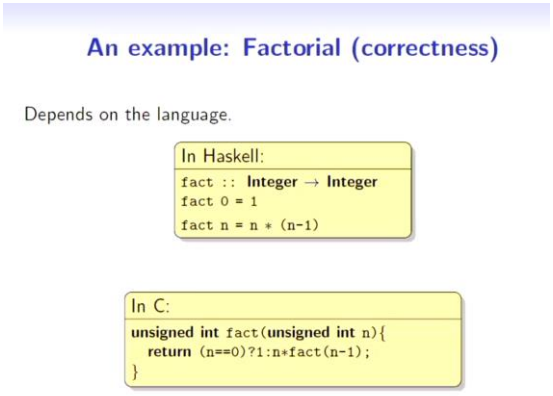
- We want to avoid ambiguity
- We also want to automate the procedure
    - Example specification: if break is pressed in car then pads will be applied eventually
    - Code: slowCarDown()
    - We can write an automatic verifier which will take both the specification and code which will automatically verify the code!
    - It will also tell you cases which have failed so that it provides automatic debugging results, or it will tell you that your program meets the specification
- Example of ambiguity which can cause problem
    - Specification: no food or drink in lecture
    - Problem: a student can bring both food and drink and this will hold true since English is an ambiguous language

An example: factorial function

- The formula for the factorial function f(n) can be defined as:
- 0 if n = 0
- Otherwise f(n+1) = (n+1)*f(n)
- The first line will give us how to compute a boundary weird case where n = 0
- The second line tells us how to compute the factorial of a positive number if we know the factorial of its predecessor
- This is proof by induction, case n = 0, case n = k and case n = k + 1
- This is the inductive definition of the factorial function

An example: factorial function specification to implementation

- Task: Given a number n where n is a natural number, compute its factorial without changing n in the process
- Game plan
    - First, we can't to compute 0 factorial
    - Next, we want to repeatedly use the previous factorial computation to compute factorials of the next number (simple)
    - Return 1 if n == 0: else return n*factorial(n-1)
    - However, values have a limit depending on the architecture



    - In the C solution we would run out of stack space before it can compute in larger factorials e.g., 100! Will have over 100 stack frames
    - Some languages may optimise however we are likely to run out of stack space
    - We say this can find n! for all numbers n however it doesn't if we tried 2^100! We would run into huge issues with stack space
    - In the case in C if n > max size of unsigned integer (2^32 – 1) then it will fail since the result would be too large to return in an unsigned integer
    - These programs will meet the specification under a certain set of values of n
- However, if we were to do this iteratively
    - Use a variable f to save the result of the last factorial computed
    - Use an additional variable k to keep track of the number so that f = k!
    - So, in dynamic programming
    - Achieve f = k! by setting f = 1 and k = 0
    - As long as k is not = to n (k != 0) we want to increase k and change f in a way that preserves f = k!

    - Code:
        - F = 1;
        - K = 0;
        - While(k != n){
            - K++;
            - F = f * k;
        - }
        - Return f;
    - This is a dynamic approach rather than recursive (uses less stack space however more computation power required)
- How do we know the dynamic approach is correct? How do we know that it will return the same values as the recursive approach?
- The property that f = k! is a loop invariant, the loop body will change the state of our variables however the loop invariants are statements that will always be true in every iteration. At the end of the loop we reach k = n so the loop invariant states f = n! as required so the code will be correct.
- To argue that the loop terminates we use variants.
    - Functions that map program states to a natural number (a value), or a well-founded domain, e.g. After 150 iterations terminate the loop
    - To show the loop terminates we have to prove that every iteration of the loop decreases the value of the variant
    - A suitable variant here would be n – k because we increase k and the value of n – k will decrease since n will not change.

What we've learnt so far

- Induction
- Specifications
- Implementations
- Correctness
- Variants and invariants
- We will delve deeper into this all

Course aims:

- Reinforce concepts from Discrete Mathematics
- Emphasise the connection between Discrete Mathematics and Computer Science
- Use mathematical concepts to **reason about programs**

Wk1-4: discrete maths and link to computer science

Assignment 1 will cover up till propositional logic.

Wk 5-7: program semantics

Wk 8-10: models, invariants and proofs, course recap

Assignment 1 due Wk 4, assignment 2 due Wk 8, assignment 3 due Wk 10

Loose guideline for assignment

Three assignments:

- Assignment 1 (due 17 March): worth 20%
- Assignment 2 (due 7 April): worth 15%
- Assignment 3 (due 28 April): worth 15%

Lateness penalty: 10% (of raw mark) per 12 hour period.

Final exam: worth 50%
You **must** achieve a score of 40% or higher on your final exam in order to pass the course.

Submissions through give!!! wooooo

Should be really good course!

Discrete mathematics

Sets

- Collection of elements
- Described by
    - Explicit enumeration of their elements

$$S_1 = \{a, b, c\} = \{a, a, b, b, b, c\}$$
$$= \{b, c, a\} = \ldots \quad \text{three elements}$$
$$S_2 = \{a, \{a\}\} \quad \text{two elements}$$
$$S_3 = \{a, b, \{a, b\}\} \quad \text{three elements}$$
$$S_4 = \{\} \quad \text{zero elements}$$
$$S_5 = \{\{\{\}\}\} \quad \text{one element}$$
$$S_6 = \{\{\}, \{\{\}\}\} \quad \text{two elements}$$

    - Unordered and no duplicates
    - Composed of elements or even other sets (collections in java)
    - Comma separates each elements of a set
    - Properties of their elements must satisfy a specification of the set, the elements are taken from some universal domain, U, a typical description involves a logical property P(x) for each element

$$S = \{ x : x \in \mathcal{U} \text{ and } P(x) \} = \{ x \in \mathcal{U} : P(x) \}$$

    - We can construct sets from other sets
        - Unions, intersections, set difference, symmetric difference, complement
        - Power sets (the set of all subsets of the current sets)
            - Example
                - X = {a, b, c}
                - Pow(x) = {EMPTYSET, {a}, {b}, {c}, {a, b, c}, {a, b}, {a, c}, {b, c}
                - Size of power set is always 2^size of set

$$\text{Pow}(X) = \{ A : A \subseteq X \}$$

- Cartesian product
- Empty set
  - This is a subset of EVERY set
- Subset notations

  $S \subseteq T$ — S is a **subset** of T; includes the case of $T \subseteq T$

  $S \subset T$ — a **proper** subset: $S \subseteq T$ and $S \neq T$
- Every set is a subset of itself since the elements of the set can be represented in its own set.
- A subset can be defined as the following
  - Let's say S contains elements and T contains element
  - S is a subset of T if all elements of S are present in T
- An element of a set just means that it exists within the set
- Examples
  - A is an element of {a, b}, notation below

    $$a \in \{a, b\}$$
  - A is not a subset of {a, b}, notation below

    $$a \not\subseteq \{a, b\}$$
  - {a} is a subset of {a, b}, notation below

    $$\{a\} \subseteq \{a, b\}$$
  - {a} is not an element of {a, b} notation below

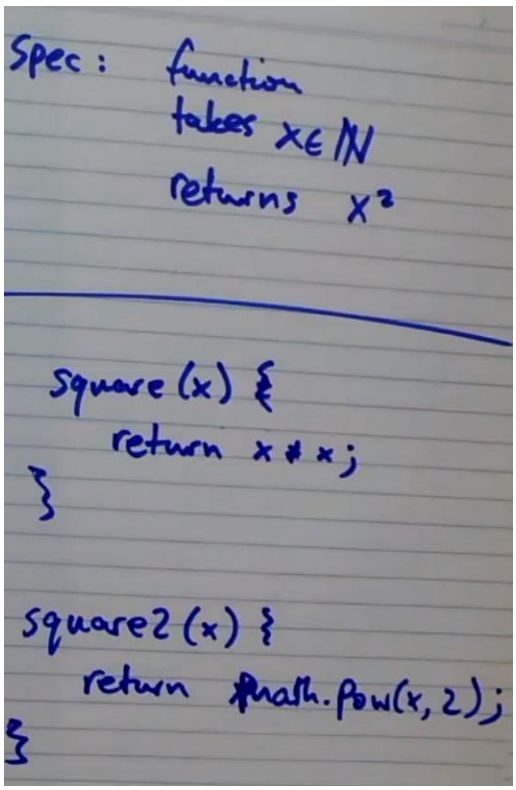
    $$\{a\} \notin \{a, b\}$$
  - an element cannot be a subset since an element is not a set

Cardinality of sets

- the number of elements in a set
- |X| = number of elements in X = cardinality of X
- |POWERSET(X)| = 2^(|X|)
- |EMPTYSET| = 0, POWERSET(EMPTYSET) = {EMPTYSET}, |POWERSET(EMPTYSET)| = 1
- |{a}| = 1, pow({a}) = {EMPTYSET, {a}}, |pow({a}) | = 2
- [m, n] denotes an interval of integers, and it is empty is n < m
- | [m, n] | = n – m + 1, for n >= m

Natural numbers $\mathbb{N} = \{0, 1, 2, \ldots\}$
Positive integers $\{1, 2, \ldots\}$
Common notation $\mathbb{N}_{>0} = \mathbb{Z}_{>0} = \mathbb{N} \setminus \{0\}$

Integers $\mathbb{Z} = \{\ldots, -n, -(n-1), \ldots, -1, 0, 1, 2, \ldots\}$
Rational numbers (fractions) $\mathbb{Q} = \left\{ \frac{m}{n} : m, n \in \mathbb{Z}, n \neq 0 \right\}$
Real numbers (decimal or binary expansions) $\mathbb{R}$
$r = a_1 a_2 \ldots a_k \,.\, b_1 b_2 \ldots$

- 

- Square(x) and square2(x) are technically different programs that perform different things, they are compiled differently, however their behaviour is identical and what we are trying to prove is that square(x) and square2(x) are the same.
- How can we reason that square(x) and square2(x) are the same, will it always have the same result, because the computer does both differently.

Intervals

- Notation

  Intervals of numbers (applies to any type)

  $$[a, b] = \{x | a \leq x \leq b\}; \quad (a, b) = \{x | a < x < b\}$$

  - We can use round brackets for exclusive and use a combination
  - $[a, b] \supseteq [a, b), (a, b] \supseteq (a, b)$
  - The exclusive interval (a, a) is equivalent to (a, a] and [a, a) however this all results to the empty set since no values fall in the intervals
  - However [a, a] an inclusive interval = {a} and has cardinality 1

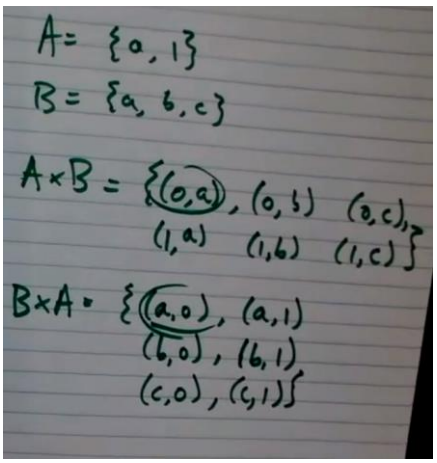
- Intervals can be said to be finite in the case [m, n] if m <= n
- | [m, n] | = n – m + 1
- **NB**

  $(a, a) = (a, a] = [a, a) = \emptyset$; *however* $[a, a] = \{a\}$.

  Intervals of $\mathbb{N}, \mathbb{Z}$ are finite: if $m \leq n$

  $$[m, n] = \{m, m+1, \ldots, n\} \qquad |[m, n]| = n - m + 1$$

Set operations

- Union: A U B (cup = union in latex, cap = intersection in latex)
- Intersection: A n B
- We say set A and B are disjoint if A n B = EMPTYSET
- if A is a subset of B (this is equivalent to saying A u B = B)
- since all elements of A are in B that means we are adding nothing new from A
- the reverse is true
- if A n B = B (this is equivalent to saying A is a superset of B)
- since A has all elements of B example [1,2,3,4] = A, and B = [1,2,3]
- A is a superset since it contains all of B and an extra element 4
- A\B: set difference, relative complement, so all elements in A but not in b
- A (+) B: symmetric difference
  - All elements in A or B but not in both
  - A (+) B = (A\B) U (B\A)
  - This is the exclusive or operation in a or b exclusively
- A^c: set complement
  - The different with respect to the universe (set)
  - Not inside the universe or our set A
- Cartesian product: set of pairs where the first element comes from the first set and the second element comes from the second set
  - A * B = {(x, y): x is element of A, y is element of B}
  - Cardinality of A * B (|A*B|) = |A| * |B|
  - Order matters! The reverse is not true A*B != B * A all elements are reversed!!!!
  - 

Laws of set operations LEARN FOR EXAM

Laws of Set Operations

Commutativity	$A \cup B = B \cup A$
	$A \cap B = B \cap A$
Associativity	$(A \cup B) \cup C = A \cup (B \cup C)$
	$(A \cap B) \cap C = A \cap (B \cap C)$
Distribution	$A \cup (B \cap C) = (A \cup B) \cap (A \cup C)$
	$A \cap (B \cup C) = (A \cap B) \cup (A \cap C)$
Identity	$A \cup \emptyset = A$
	$A \cap \mathcal{U} = A$
Complementation	$A \cup (A^c) = \mathcal{U}$
	$A \cap (A^c) = \emptyset$

We need to be strict and state all the rules we use in our proofs!

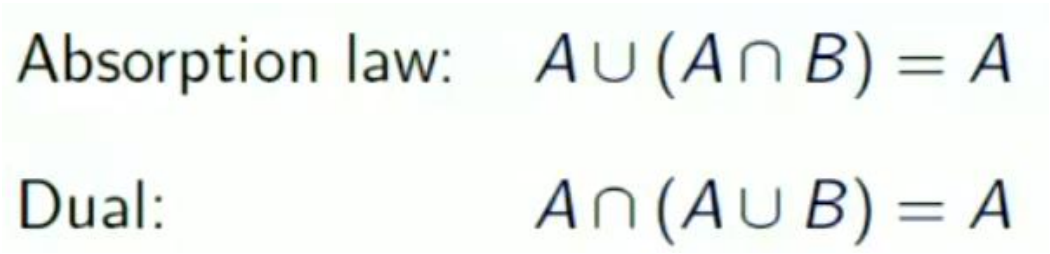
Order matters!

Other useful set laws

The following are all derivable from the previous 10 laws.

Idempotence	$A \cap A = A$
	$A \cup A = A$
Double complementation	$(A^c)^c = A$
Annihilation	$A \cap \emptyset = \emptyset$
	$A \cup \mathcal{U} = \mathcal{U}$
de Morgan's Laws	$(A \cap B)^c = A^c \cup B^c$
	$(A \cup B)^c = A^c \cap B^c$

This allows us to do really cool absorptions laws



- If A is a set defined using union, intersection, EMPTYSET, universe, then dual(A) is the expression by doing the swaps (union swapped with intersection, and EMPTYSET swapped with universe).
- Principle of duality
    - If you can prove A1 = A2 using the laws of set operations, then you can prove that dual(A1) =dual(A2)
    - Every law holds true for the opposite definition

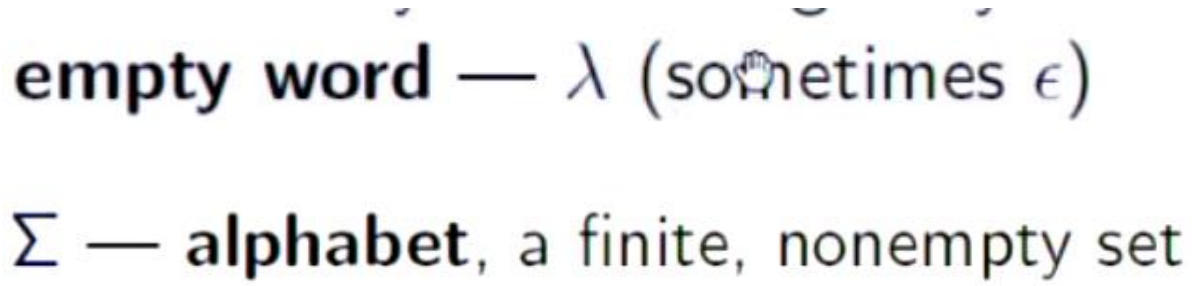
Formal language

Sigma is a finite non-empty set known as an alphabet

An alphabet can be thought of as a set of symbols (elements of alphabet)

A word is a finite string of symbols from the alphabet

An empty word can be referred to as lambda sometimes epsilon. (a word with no symbols)



Example

- W = aba, w = 0110101111
- Length(W) = number of symbols in w
- Length(aaa) = 3, length(lambda) = 0
- The only operation on words is concatenation (adding it to the end of another word)
- Example cat (concat) dog = catdog ORDER MATTERS
- Lambda (concat) w = w = w (concat) lambda (since lambda is nothing, we are adding nothing lol
- Length(vw) = length(v)+length(w)
- Length(catdog) = length(cat)+length(dog) NOTE TOGETHER IS NOT MULTIPLY BUT ADD

Notation: Σ^k — set of all words of length k

Σ^* — set of all words (of all [finite] lengths)

Σ^+ — set of all nonempty words (of any positive length)

$$\Sigma^0 = \{\lambda\},\ \Sigma^1 = \Sigma$$

$$\Sigma^* = \Sigma^0 \cup \Sigma^1 \cup \Sigma^2 \cup \ldots; \quad \Sigma^{\leq n} = \bigcup_{i=0}^{n} \Sigma^i$$

$$\Sigma^+ = \Sigma^1 \cup \Sigma^2 \cup \ldots = \Sigma^* \setminus \{\lambda\}$$

A set of all words length 1 is the same as alphabet

Set of all words length 0 is an empty word

SIGMA* note is not infinite, however all words in the set are arbitrary length, that means we can't put a number on it but its finite.

SIGMA* and SIGMA+ work in same way as greps * and + operators

A language is a subset of SIGMA*. Only the subsets that can be formed according to rules are interesting. This collection of descriptive/formative rules is called grammar

Examples: programming syntax, database queries

The set of all C programs is a language.

Example (HTML documents)

Take $\Sigma = \{$"<html>", "</html>", "<head>", "</head>", "<body>", ...$\}$.

The (language of) **valid HTML documents** is loosely described as follows:

- Starts with "<html>"
- Next symbol is "<head>"
- Followed by zero or more symbols from the set of HeadItems (defined elsewhere)
- Followed by "</head>"
- Followed by "<body>"
- Followed by zero or more symbols from the set of BodyItems (defined elsewhere)
- Followed by "</body>"
- Followed by "</html>"

Lecture 2

Formal languages

- Languages are sets, so the standard set operations can be used to build new languages

Let $A = \{aa, bb\}$ and $B = \{\lambda, c\}$ be languages over $\Sigma = \{a, b, c\}$.

- $A \cup B = \{\lambda, c, aa, bb\}$
- $AB = \{aa, bb, aac, bbc\}$
- $AA = \{aaaa, aabb, bbaa, bbbb\}$

- Two set operations that apply to languages uniquely
  - Concatenation (written as juxtaposition)
    - XY = {xy: x is in X and y is in Y}
  - Kleene star: X* is the set of words that are made up by concatenating 0 or more words in X (grep matches 0 or more) so multiple concatenations of each elements

- $A^* = \{\lambda, aa, bb, aaaa, aabb, bbaa, bbbb, aaaaaa, \ldots\}$
- $B^* = \{\lambda, c, cc, ccc, cccc, \ldots\}$

$$\{\lambda\}^* = \{\lambda\}$$

$$\emptyset^* = \{\lambda\}$$

    - Star does 0 -> infinite matches

Relations

Relations are an abstraction used to capture the idea that objects from certain domains are related.

These objects may

- Influence one another or each other
- Share some common properties
- Correspond to each other precisely when constraints are satisfied

Functions capture the idea of transforming inputs into outputs → f(x) = y, input x → output y

Functions and relations formalise the concept of interaction among objects from various domains, however there must be a specified domain for each type of object.

Relations and functions are the foundations of Computer Science

- Databases are collections of relations
- Common data structures (graphs, trees, lists) are relations
- Any ordering is a relation
- Functions/procedures/programs compute relations between the input and output

Relations are used in most problem specifications and to describe formal properties of programs. Studying relations and their properties helps with formalisation, implementation and verification of programs.

A n-ary relation is a subset of the cartesian product of n sets.

$$R \subseteq S_1 \times S_2 \times \ldots \times S_n$$

$$x \in R \to x = (x_1, x_2, \ldots x_n) \text{ where each } x_i \in S_i$$

If n = 2, we have a binary relation, r is a subset of S * T where S and T are sets

$$\text{equivalent notations: } (x_1, x_2, \ldots x_n) \in R \iff R(x_1, x_2, \ldots x_n)$$
$$\text{for binary relations: } (x, y) \in R \iff R(x, y) \iff xRy.$$

Examples

These are the common relations

$$\text{Equality: } =$$
$$\text{Inequality: } \leq, \geq, <, >, \neq$$
$$\text{Divides relation: } | \text{ (recall } m|n \text{ if } n = km \text{ for some } k \in \mathbb{Z})$$
$$\text{Element of: } \in$$
$$\text{Subset, superset: } \subseteq, \subset, \supseteq, \supset$$
$$\text{Size functions (sort of): } |\cdot|, \text{length}(\cdot)$$

Note 6/3 = 2, the divides are an operator in this case, however 3|6 is the relation that 6 is divisible by 3 and the relation!

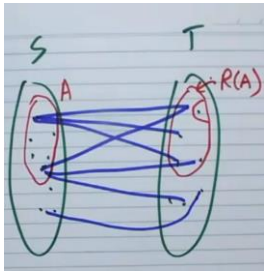
The most common relations are binary relations

$$R \subseteq S \times T; \quad R = \{(s, t) : \text{“some property that links } s, t\text{”}\}$$

For related s, t we can write $(s, t) \in R$ or sRt; for unrelated items either $(s, t) \notin R$ or $s \not R t$.

Let’s say we have a relation R which is a subset of S * T, and the values A are a subset of S and the values B are a subset of T

We can say that R(A) = {values t in the set T such that (s, t) are a relation for some value s in A}



- Converse relation $R^{\leftarrow} \subseteq T \times S$:

$$R^{\leftarrow} \stackrel{\text{def}}{=} \{(t, s) \in T \times S \ : \ (s, t) \in R\}$$

- $R^{\leftarrow}(B) = \{s \in S \ : \ (s, t) \in R \text{ for some } t \in B\}$

Observe that $(R^{\leftarrow})^{\leftarrow} = R$.

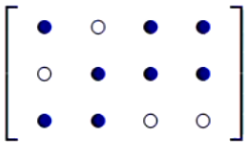
Binary Relations

A binary relation where R is a subset of S * T can be represented as a matrix with rows which are elements of S and columns which are elements of T. example, S = {s1, s2, s3} and T = {t1, t2, t3, t4}

S * T = {t1s1, t2s1, t3s1, t4s1}

{t1s2, t2s2, t3s2, t4s2}

{t1s3, t2s3, t3s3, t4s3}



The binary relations can be thought of as the coloured in dots while the non-related are empty points.

Relations on a single domain

Binary relationships between elements of the same set are very important. We say that 'R is a relation of S' if

R is a subset of S * S

These relations can be visualized as a directed graph which have

Vertices as elements S

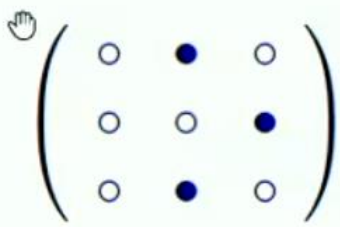
Edges which are the relation between different vertices R

Adjacency matrix

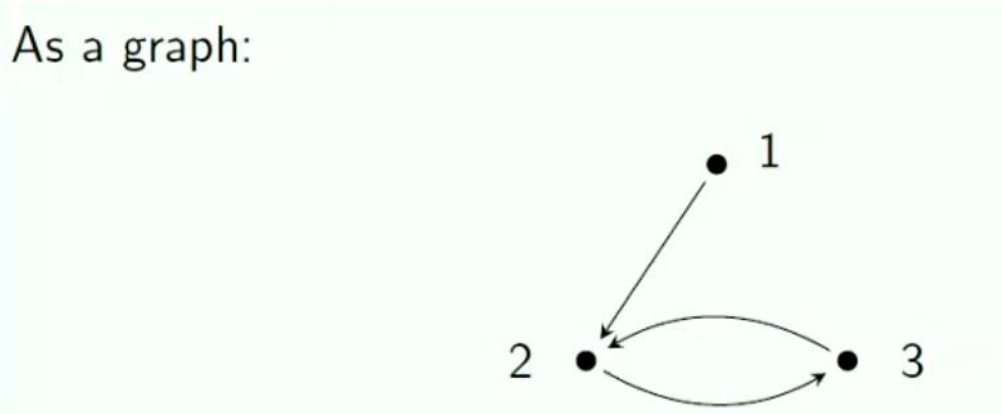
$S = \{1, 2, 3\}$

$R = \{(1, 2), (2, 3), (3, 2)\}$

As a matrix:



As a graph:



Special trivial relations

For all examples we use set S

- identity relation (diagonal, equality)
  - relationship where everything is related to itself
  - E = {(x, x): X is an element of S}
- Empty nothing is related to anything
- Universal relation U = S * S everything is related

Properties of binary relations

- Reflexive if every element is related to itself
- Anti-reflexive if no element is related to itself
- Symmetric, if a is related to B then b is related to A
- Anti-symmetric, if x is related to y, and y is related to x, then x = y
- The ^^ only time x is related to y and y is related to x is when x = y
- Transitive if x is related to z, and y is related to z, then x is related y
- The $\forall$ means for all values in

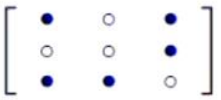
(R)	reflexive	$(x, x) \in R$	$\forall x \in S$
(AR)	antireflexive	$(x, x) \notin R$	$\forall x \in S$
(S)	symmetric	$(x, y) \in R \rightarrow (y, x) \in R$	$\forall x, y \in S$
(AS)	antisymmetric	$(x, y), (y, x) \in R \rightarrow x = y$	$\forall x, y \in S$
(T)	transitive	$(x, y), (y, z) \in R \rightarrow (x, z) \in R$	$\forall x, y, z \in S$

NB

An object, notion etc. is considered to satisfy a property if none of its instances violates any defining statement of that property.

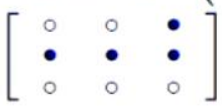
(R) reflexive $(x, x) \in R$ for all $x \in S$ 

(AR) antireflexive $(x, x) \notin R$ 

(S) symmetric $(x, y) \in R \rightarrow (y, x) \in R$ 

(AS) antisymmetric $(x, y), (y, x) \in R \rightarrow x = y$

(T) transitive $(x, y), (y, z) \in R \rightarrow (x, z) \in R$

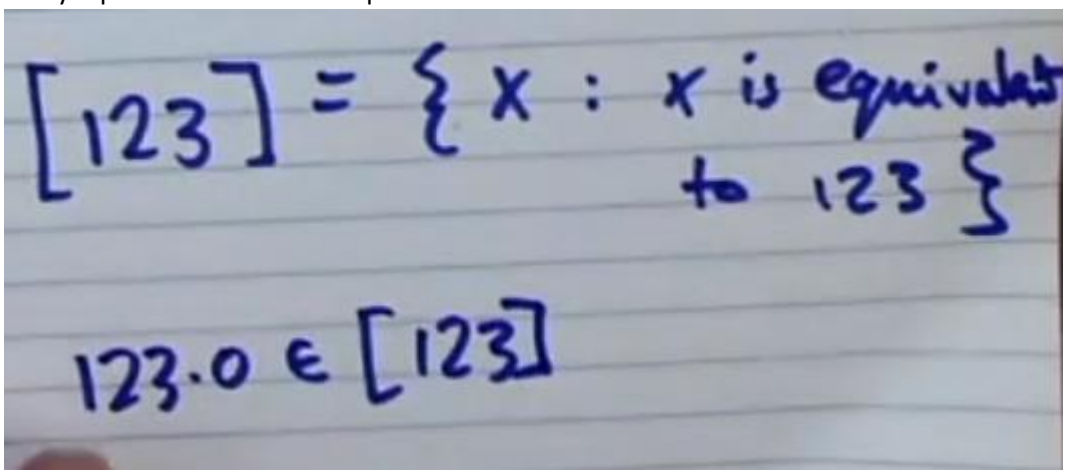


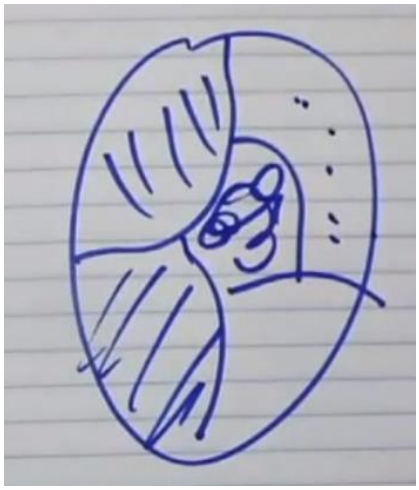
	(R)	(AR)	(S)	(AS)	(T)
$=$	✓		✓	✓	✓
$\leq$	✓			✓	✓
$<$		✓		✓	✓
$\emptyset$		✓	✓	✓	✓
$\mathcal{U}$	✓		✓		✓
$\mid$	✓			✓	✓

A relation can be both symmetric and anti-symmetric. A relation CANNOT be simultaneous reflexive as antireflexive unless it is the empty set

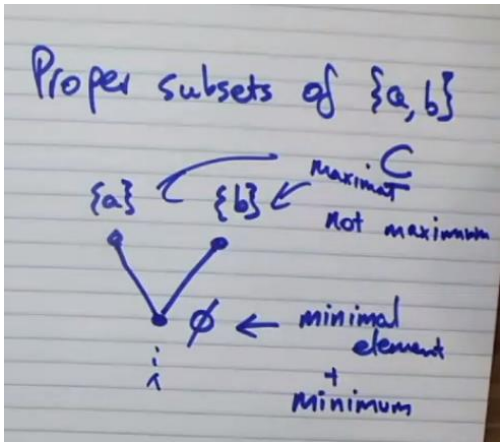
Types of relations

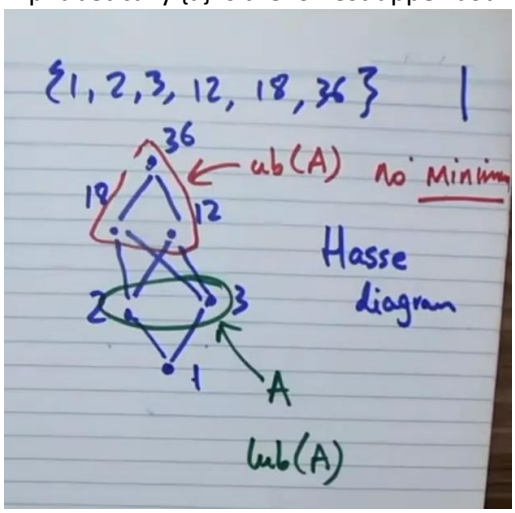
Equivalence relation and partitions

- A relation is called an equivalence relation if it is
    - Reflexive
    - Symmetric
    - Transitive
- Every equivalence R defines equivalence classes it's on domain S
- 

- Equivalence classes group together all elements which are related to a certain element
- For example, equivalence class can be {123, 123.0, 123.00, one hundred twenty-three} where all elements are equivalent to each other
- Equivalence classes partition objects so there is no overlap. If an element is in two different equivalent classes, then the equivalent classes are the same.
- 
- No element lies between two different sections
- This means equivalence classes are disjoint regions that cover the entire domain, so every element belongs to one and only one equivalence class.
- We call s1, s2 representatives of different equivalence classes. (usually the smallest value in the equivalence class set.

Partial order relation and ordering

- This call (s, partial order) a poset – partially ordered set
- A partial order on a set satisfies the following properties
    - Reflexive
    - Anti-symmetric
    - Transitive
- Every finite poset can be represented as a Hasse diagram here a line is drawn upward from x to y if
- Minimal and maximal elements (always exist in every finite poset)
- Minimum and maximum – unique minimal and maximal element which might not exist
- Lowest upper bound, and greatest lower bound of a subset
- Lowest upper bound of a = minimum of all upper bounds of A
- Greatest lower bound of a = maximum of all lower bounds of A
- 

- Alphabetically {a} is the lowest upper bound element
- 

Functions

A function f: S → T is a binary relation $f \subseteq S \times T$ where for all s that are elements of S, there exactly one t which is an element of T, such that (s, t) are elements of f

We write f(s) for the unique element related to s

A partial function $f : S \nrightarrow T$ is a binary relation $f \subseteq S \times T$ such that for all s which are elements of S, there is at most one t that is an element of T such that (s, t) are elements of our function

That is, it is a function $f : S' \longrightarrow T$ for $S' \subseteq S$

A function that is defined on some subset of S

F: S → T describes pairing of sets, it means that our function f will assign each element in S a unique element in T. to emphasise where a specific element is sent, we write f: x |→ y, which means the same as f(x) = y

S is the domain of F (the inputs) Dom(f)

T is the co-domain of F (possible outputs) Codom(f)

F(S) is the image of f (the output of our function) = {f(x): x is in the domain of f}

the domain and co-domain are critical aspects of a function's definition

f: N → Z (natural → integer) given by f(x) |→ x^2

and

g: N → N (natural → natural) given by g(x) |→ x^2

are different functions even though they have the same behaviour

domain can be thought of as pre-condition, and co-domain are the post condition, guaranteed inputs and guaranteed outputs

composition of functions

g ***of*** f: x |→g(f(x)), requiring the image of F to be in the domain of G

$$\mathrm{Im}(f) \subseteq \mathrm{Dom}(g)$$

this may also be associative

h ***of*** (g ***of*** f) = (h ***of*** g) ***of*** f, we can write h ***of*** g ***of*** f

if a function maps a set into itself, when the domain of the function is the same as the co-domain of the function, the function can be composed with itself

$f \circ f, f \circ f \circ f, \ldots,$ also written f^2, f^3

Identity function on S

$$\mathrm{Id}_S(x) = x, x \in S; \mathrm{Dom}(i) = \mathrm{Codom}(i) = \mathrm{Im}(i) = S$$

For $g : S \longrightarrow T \quad g \circ \mathrm{Id}_S = g,\ \mathrm{Id}_T \circ g = g$

The identity relation maps all functions to themselves

Composition of Binary Relations

If R1 is a subset of S * T, and R2 is a subset of T * U, then the composition of R1 and R2 is the relation

R1; R2: = {(a, c): there is a value a and b in T such that (a, b) is in R1 and (b, c) is in R2}

$$R_1; R_2 := \{(a, c) : \quad \text{there is a } b \in T \text{ such that } (a, b) \in R_1 \text{ and } (b, c) \in R_2\}.$$

Do R1 then do R2

If f: S → T, and g: T → S are functions, then f; g = g ***of*** f

Properties of functions

Surjective functions

A function is called surjective or onto if every element of the codomain is mapped by at least one x in the domain, so Im(f) = Codom(f)

F(x) = x^2 is not onto, because 3 is not an output of x^2

Examples (of functions that are surjective)

- $f : \mathbb{N} \longrightarrow \mathbb{N}$ with $f(x) \mapsto x$
- Floor, ceiling

Examples (of functions that are not surjective)

- $f : \mathbb{N} \longrightarrow \mathbb{N}$ with $f(x) \mapsto x^2$
- $f : \{a, \ldots, z\}^* \longrightarrow \{a, \ldots, z\}^*$ with $f(\omega) \mapsto a\omega e$

Injective functions

A function is called injective or 1-1 (one to one) if different x implies different f(x), for example

F(x) = f(y) implies x = y

Examples (of functions that are injective)

- $f : \mathbb{N} \longrightarrow \mathbb{N}$ with $f(x) \mapsto x$
- set complement (for a fixed universe)

Examples (of functions that are not injective)

- absolute value, floor, ceiling
- length of a word

A function is bijective if it is both injective and surjective!

Question

$f^{\leftarrow}$ is a relation; when is it a function?

Answer

When f is a bijection.

Bijection is when the domain and co-domain have every value mapped to a unique value! So, f(x) = x is a bijection

Inverse functions

Inverse function f^-1: T → S

For a given f: S → T exists exactly when f is bijective

Image of a subdomain A under a function

The inverse function undoes the mapping of T → S. sort of like + undoes a -

$$f(A) = \{ f(s) : s \in A \} = \{ t \in T : t = f(s) \text{ for some } s \in A \}$$

The converse always exists but it is not always a function!

Inverse functions ONLY exists when f is a bijection

Inverse image

It is defined for every f (recall: converse of a relation)

it is defined for every f (recall: converse of a relation)

If f^{-1} exists then $f^{\leftarrow}(B) = f^{-1}(B)$

$f(\emptyset) = \emptyset, f^{\leftarrow}(\emptyset) = \emptyset$

Propositional logic

A sentence of a natural language which is declarative, it can be said to be true or false. Like a statement!

Examples

- Richard Nixon was president of Ecuador.
- A square root of 16 is 4.
- Euclid's program gets stuck in an infinite loop if you input 0.
- Whatever list of numbers you give as input to this program, it outputs the same list but in increasing order.
- $x^n + y^n = z^n$ has no nontrivial integer solutions for $n > 2$.
- 3 divides 24.
- K_5 is planar.

The following are *not* declarative sentences:

- Gubble gimble goo
- For Pete's sake, take out the garbage!
- Did you watch MediaWatch last week?
- Please waive the prerequisites for this subject for me.
- x divides y.
- $x = 3$ and x divides 24.

Propositional functions are known as predicates where you give a proposition and it returns a predicate

We can build up propositions. Propositional logic is a formal representation of constructions where the truth value of the whole is determined from the truth value of its components, for example.

- Chef is a bit of a Romeo *and* Kenny is always getting killed.
- Either Bill is a liar *or* Hillary is innocent of Whitewater.
- *It is not the case that* this program always halts.

Not all constructions of natural language are truth-functional.

Not all constructions of natural language are truth-functional:

- *Obama believes that* Iran is developing nukes.
- *Chef said* they killed Kenny.
- This program always halts *because* it contains no loops.
- The disk crashed *after* I saved my file.

NB

Various **modal logics** *extend classical propositional logic to represent, and reason about, these and other constructions.*

The statements above are based on beliefs, it turned a proposition (Iran is developing nukes) into a belief that Obama believes.

We can add modalities to extend propositional logic to represent and reason about behaviours.

The Three Basic Connectives of Propositional Logic

symbol	text
$\wedge$	"and", "but", ";", ":"
$\vee$	"or", "either . . . or . . ."
$\neg$	"not", "it is not the case that"

Truth tables:

A	B	A $\wedge$ B
F	F	F
F	T	F
T	F	F
T	T	T

A	B	A $\vee$ B
F	F	F
F	T	T
T	F	T
T	T	T

A	$\neg$ A
F	T
T	F

Program logic example

If x > 0 or (x <= 0 and y > 100)

Let $p \stackrel{\text{def}}{=} (\texttt{x} > 0)$ and $q \stackrel{\text{def}}{=} (\texttt{y} > 100)$

$p \vee (\neg p \wedge q)$

p	q	$\neg p$	$\neg p \wedge q$	$p \vee (\neg p \wedge q)$
F	F	T	F	F
F	T	T	T	T
T	F	F	F	T
T	T	F	F	T

This is equivalent to $p \vee q$. Hence the code can be simplified to

```
if x > 0 or y > 100:
```

These following logical statements are all the same

- if A then B
- A only if B
- B if A
- A implies B
- it follows from A that B
- whenever A, B
- A is a sufficient condition for B
- B is a necessary condition for A

Same thing different wording. All these break down to A implies B

Vacuous truth

How to interpret A → B when A is false

A → B means a implies B, so if (condition A is met) then B (conclusion)

Material implication is false only when the premise condition holds and the conclusion does not

If the premise is false, the implication is true no matter how absurd the conclusion is

Consider this

Both the following statements are true:

- If February has 30 days then March has 31 days.
- If February has 30 days then March has 42 days.

If our premise is false then our conclusion is true.

Another would be

Each element in the empty set is even, its true since there are no elements in the empty set

Hilton is good at league, and bob is a billionaire, true since Hilton sucks

A	B	A → B
F	F	T
F	T	T
T	F	F
T	T	T

The Three Basic Connectives of Propositional Logic

symbol	text
$\wedge$	"and", "but", ";", ":"
$\vee$	"or", "either . . . or . . ."
$\neg$	"not", "it is not the case that"

Truth tables:

A	B	A $\wedge$ B
F	F	F
F	T	F
T	F	F
T	T	T

A	B	A $\vee$ B
F	F	F
F	T	T
T	F	T
T	T	T

A	$\neg$ A
F	T
T	F

We want to identify the propositions and THEN discover the relations.

Unless

A unless B can be approximated as ~B → A

Example

I go swimming unless it rains, this also means (if it is not raining then I go swimming).

Correctness of the translation is better when the person is not always swimming

A unless B can be approximated as $\neg B \rightarrow A$

E.g.
I go swimming unless it rains = If it is not raining then I go swimming.
Correctness of the translation is perhaps easier to see in:
I don't go swimming unless the sun shines = If the sun does not shine then I don't go swimming.

Note that "I go swimming unless it rains, but sometimes I swim even though it is raining" makes sense, so the translation of "A unless B" should not imply $B \rightarrow \neg A$.

The last point basically says that swimming does not imply that it is raining

Just in case or (iff (if and only if))

A just in case B usually means A if and only if B written as A ←→B

The program terminates just in case the input is a positive number. This is equivalent to saying the program terminates if and only if the input is positive

I will have an entrée just in case I won't have desert. This is equivalent to saying, if I have desert, I will not have an entrée and vice versa.

It has the following truth table:

A	B	$A \leftrightarrow B$
F	F	T
F	T	F
T	F	F
T	T	T

Same as $(A \rightarrow B) \wedge (B \rightarrow A)$

A propositional formula is made up of propositional variables (statements) and logical connectives.

A truth assignment assigns true or false to each propositional variable and using the logical connectives gives a truth value to all propositional formulas.

Lecture 3

Logical Equivalence

Two formulas (phi, si) ϕ, ψ and theta (/) are logically equivalent if they have the same truth value for all truth valuations.

Application

If the two formulas are logically equivalent then the digital circuits corresponding to each formula compute the same function, thus proving equivalence of formulas can be used to optimise circuits.

examples

Excluded Middle	$p \vee \neg p \equiv \top$
Contradiction	$p \wedge \neg p \equiv \bot$
Identity	$p \vee \bot \equiv p$
	$p \wedge \top \equiv p$
	$p \vee \top \equiv \top$
	$p \wedge \bot \equiv \bot$
Idempotence	$p \vee p \equiv p$
	$p \wedge p \equiv p$
Double Negation	$\neg\neg p \equiv p$
Commutativity	$p \vee q \equiv q \vee p$
	$p \wedge q \equiv q \wedge p$

Associativity	$(p \vee q) \vee r \equiv p \vee (q \vee r)$
	$(p \wedge q) \wedge r \equiv p \wedge (q \wedge r)$
Distribution	$p \vee (q \wedge r) \equiv (p \vee q) \wedge (p \vee r)$
	$p \wedge (q \vee r) \equiv (p \wedge q) \vee (p \wedge r)$
De Morgan's laws	$\neg(p \wedge q) \equiv \neg p \vee \neg q$
	$\neg(p \vee q) \equiv \neg p \wedge \neg q$
Implication	$p \to q \equiv \neg p \vee q$
	$p \leftrightarrow q \equiv (p \to q) \wedge (q \to p)$

Satisfiability of formulas

A formula is satisfiable if it evaluates to T for some assignment of truth values to its basic propositions. This encapsulates what is considered hard problems.

Example

You are planning a party, but your friends are a bit touchy about who will be there.

1. If John comes, he will get very hostile if Sarah is there.
2. Sarah will only come if Kim will be there also.
3. Kim says she will not come unless John does.

Who can you invite without making someone unhappy?

Invite john and Kim, can use truth tables use J S K, there is a very easy configuration that satisfies this however it is trivial as all not coming is true, however to satisfy truly we need logical proposition.

Validity, entailment, arguments

An argument consists of a set of declarative sentences called premises and a declarative sentence called the conclusion

Example

Premises:	Frank took the Ford or the Toyota.
	If Frank took the Ford he will be late.
	Frank is not late.
Conclusion:	Frank took the Toyota

3 propositions, and 1 conclusion. We can prove this is true using logical propositions like above

The above is very simple since frank only really had to take the ford or Toyota but if he took the ford, he would be late, he wasn't late so only option is he took the Toyota

An argument is valid if the conclusions are true whenever all the premises are true, thus if we believe the premises, we should also believe the conclusion. If one premise is false then we don't care as it is not valid.

- The conclusion logically follows from the premises
- The conclusion is a logical consequence of the premises
- The premises entail the conclusion

Entailment is a relation between a set of premises and a set of conclusions

For arguments in propositional logic we can capture validity as

- Let Formula 1 -> Formula n be the propositional logic
- Draw a truth table with columns for each formula
- The argument with premises formula 1 -> formula n and conclusion are valid denoted as

$$\phi_1, \ldots, \phi_n \models \phi$$

- If in every row of the truth table where each formula is all true then the conclusion is also true
- All we are doing is looking for the rows in the truth table where all are true

Reasoning about requirements/specifications

Suppose we can formulate a set of requirements Formula 1 -> formula n

Now suppose C is a statement formalised by a formula f1. Then

- The requirements cannot be implemented if all formulas cannot be satisfied
- If all formulas entail f1, then every correct implementation of the requirements will be such that C is always true in the resulting system

The requirements cannot be implemented if $\phi_1 \wedge \ldots \wedge \phi_n$ is not satisfiable.

If $\phi_1, \ldots \phi_n \models \psi$ then every correct implementation of the requirements R will be such that C is always true in the resulting system.

If $\phi_1, \ldots \phi_{n-1} \models \phi_n$, then the condition ϕ_n of the specification is redundant and need not be stated in the specification.

Example

Requirements R: A burglar alarm system for a house is to operate as follows. The alarm should not sound unless the system has been armed or there is a fire. If the system has been armed and a door is disturbed, the alarm should ring. Irrespective of whether the system has been armed, the alarm should go off when there is a fire.

Conclusion C: If the alarm is ringing and there is no fire, then the system must have been armed.

Questions

1. Will every system correctly implementing requirements R satisfy C?
2. Is the final sentence of the requirements redundant?

Our two questions then correspond to

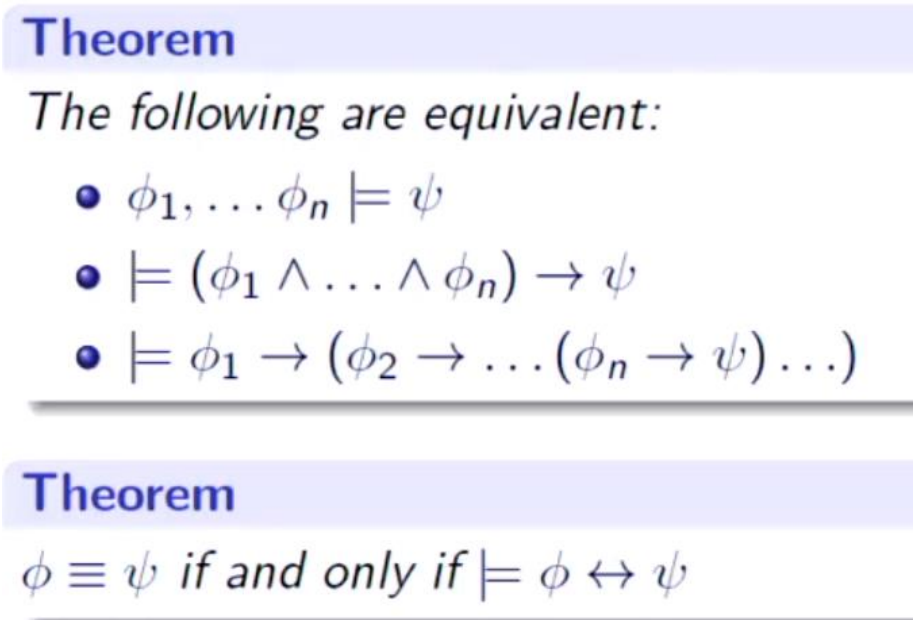
1. Does $S \rightarrow (A \vee F), (A \wedge D) \rightarrow S, F \rightarrow S \models (S \wedge \neg F) \rightarrow A$?
2. Does $S \rightarrow (A \vee F), (A \wedge D) \rightarrow S \models F \rightarrow S$?

Validity of formulas

A formula is valid or a tautology, denoted by $\models \phi$, if it evaluates to true for all assignments of truth values to its basic propositions, for example

A	B	$(A \rightarrow B) \rightarrow (\neg B \rightarrow \neg A)$
F	F	T
F	T	T
T	F	T
T	T	T

The two formulas are logically equivalent if and only if one formula is valid



Deeper Reasoning

Entailment captures a form of logical reasoning but it cannot handle relatively simple logic arguments

Example

- Socrates is a man
- All men are mortal
- Therefore, Socrates is mortal

We need to add expressiveness to propositional logic so that we can capture notions such as the relation between man and men in the first two statements, and the quantified statement “all men”

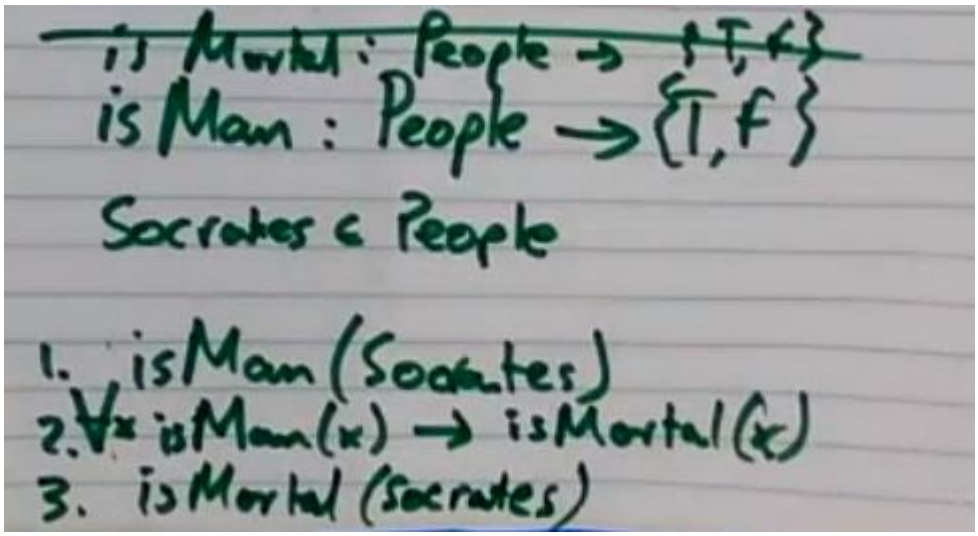
First order logic → easy to decide validity

Second order logic → undecidable (when we add more expressions to first order logic)

Predicates (relations)

Predicates are functions that take inputs from a set of individuals and return true/false. They are relations between the individuals. Predicates enable us to establish relationships between different propositions such as the man/men connection between Socrates and all men, allowing deeper reasoning than propositional logic can give.

Example



1 and 2 are the premises from statement that Socrates is mortal, and that for all men, all men satisfy the mortal requirement.

Therefore, the conclusion 3 is entailed by our premises 1 and 2

However, this comes at a cost of verifying that the statement 2 is true FOR ALL MEN (cost heavy)

Quantifiers

Quantifiers allow us to make quantified statements over predicates

Example

- If there exists a satisfying assignment
- Every natural number greater than 2

The two standard quantifiers are

- $\forall$: "for all", "for any", "every"
- $\exists$: "there exists", "there is", "for some", "at least one"

Goldbach's conjecture states

For all numbers inside the set of even numbers where the numbers are greater than 2, there exists a pair of numbers in the natural number set that are both prime, such that the even number can is the sum of the pair

Example

Goldbach's conjecture

$$\forall n \in 2\mathbb{N}\,(n > 2 \rightarrow \exists p, q \in \mathbb{N}\,(p, q \in \text{PRIMES} \wedge n = p + q))$$

Everything is assessible but this stuff should be second nature

Recursion

We can prove recursive programs are correct!

Reduces problems to smaller cases

- Factorial
- Towers of Hanoi
- Merge sort, Quicksort
- Tree algorithms

Recursion in data structures allow finite definitions for arbitrary large objects for example

- Trees
- Linked lists
- Words
- Natural numbers
- Propositional formulas

Analysis of recursion allows proof of properties

- Recursive sequences (Fibonacci for example)
- Structural induction

Recursion consists of

- Base cases B
- Recursive processes R which are built off earlier terms and previous input
- R is also called the recurrence formula especially when dealing with sequences

Example

Factorial using recursion

- Base case B: 0! = 1
- Recursive process: (n+1)! = (n+1) * n!
- If(n==0):1; else: return n*fact(n-1);

Towers of Hanoi

- There are 3 towers
- There are n disks of decreasing size placed on the first tower
- You need to move all the disk from the first tower to the last tower
- Large disks cannot be placed on-top of smaller disks
- The third tower can be used to temporarily hold disks

Recursive data types

Allow us to describe arbitrary large (infinite) data in a finite manner

Example

For example, natural numbers are either 0 or one more than another natural number

- Base case: 0
- Recursive process: n+1

Formal definition of $\mathbb{N}$:

- (B) $0 \in \mathbb{N}$
- (R) If $n \in \mathbb{N}$ then $(n+1) \in \mathbb{N}$

The Fibonacci sequence starts at 0,1 and all terms after are the sum of the two precious terms

- Base case: F (0) = 0 and F (1) = 1
- Recursive process: F(n) = F(n-1) + F(n-2)

We can define these sequences as a function for example the Fibonacci sequence can be defined as F: Natural → Fibonacci number. Choice of perspective depends on what structure we view as our base object (ground type).

A linked list

- A linked list is a list of zero or more nodes
- Contains a head which is the initial element of the list
- Each element contains the link to the next element (or previous if double linked)
- We can view the list as
    - Base case: empty list (null)
    - Recursive pair: an ordered pair (data, list)
        - So, each element points to a sub list (sort of like BST subtrees)

Words over an alphabet

- A word over an alphabet is either
- Base case: lambda (empty)
- Recursive process: a symbol followed by a word
- This also matches the recursive definition of a linked list data type
- A linked list can be though of as a word over an alphabet

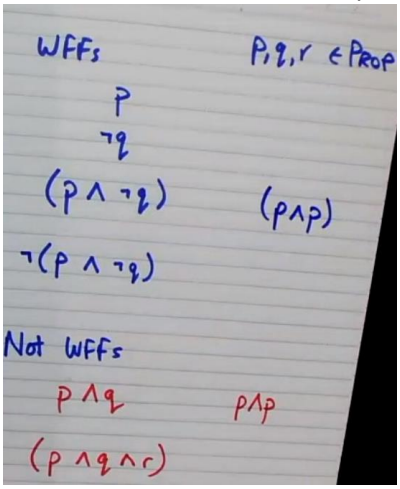
Formal definition of Σ^*:

- (B) $\lambda \in \Sigma^*$
- (R) If $w \in \Sigma^*$ then $aw \in \Sigma^*$ for all $a \in \Sigma$

Propositional formulas

- A well-formed formula (wff) over a set of propositional variables is defined as
  - Base case1: true is a well-formed formula
  - Base case 2: false is a well-formed formula
  - Base case 3: p is a well-formed formula for all propositional formulas
  - Recursive process 1: if formula 1 is a well-formed formula than not formula 1 is a well-formed formula
  - Recursive process 2: if formula 1 and formula 2 are both well-formed formulas then
    - Formula 1 AND formula 2 are well formed formulas
    - Formula 1 OR formula 2 are well formed formulas
    - Formula 1 implies formula 2 is a well-formed formula
    - Formula 1 bi-implies formula 2 is also a well-formed formula

    - (B) $\top$ is a wff
    - (B) $\bot$ is a wff
    - (B) p is a wff for all $p \in \text{PROP}$
    - (R) If φ is a wff then $\neg\varphi$ is a wff
    - (R) If φ and ψ are wffs then:
      - $(\varphi \wedge \psi)$,
      - $(\varphi \vee \psi)$,
      - $(\varphi \rightarrow \psi)$, and
      - $(\varphi \leftrightarrow \psi)$ are wffs.
  -
- Placing brackets is very strict to avoid ambiguity
- (p AND p) are a wff but p AND p without brackets is not a wff
- Only one brackets for two maximum (binary) that is why p AND q AND r are not wff
- For example, if we had p AND q OR R, we don't know how to interpret that and the result would be different, do we do p AND q first or, q OR r first

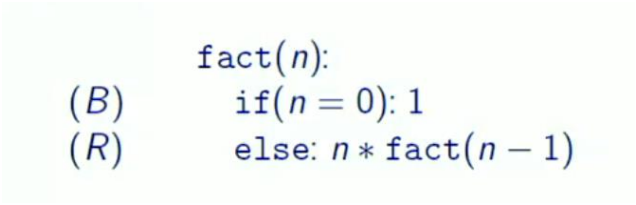


Programming over recursive datatypes

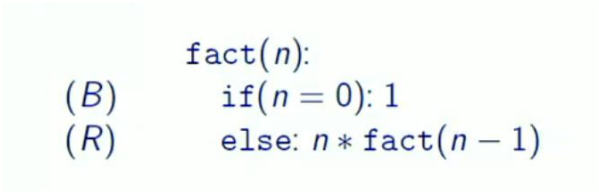
Recursive datatypes make recursive programming easy

Example

The factorial function



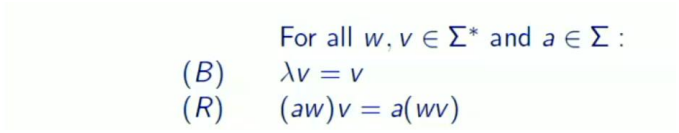
summing the first n natural numbers

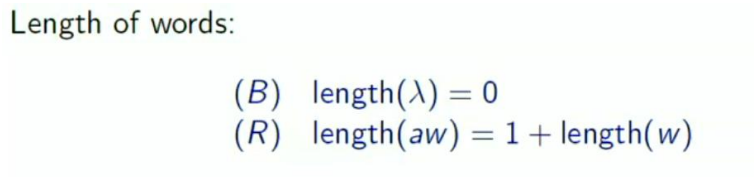


Summing elements of a linked list

- Sum (node *list):
- If (list == NULL): 0
- Else: list.data + sum(list.next)
- List {
    - Int data;
    - List* next;
- }

Concatenation of words





Evaluation of a propositional formula

Recursive datatypes

Describe arbitrarily large objects in a finite way

Recursive functions

Define behaviour for these objects in a finite way

Induction

Reason about these objects in a finite way

Induction

There are infinite natural numbers, and we need a way to reason about things, example if we want to prove that the sum of n natural numbers is ((n)*(n+1))/2 we can either prove it true by manual proof which is impossible as there are infinite n's, so we use induction as a finite proof.

Suppose we would like to teach a conclusion of the form predicate P(x) for all x of some type

Inductive reasoning proceeds from examples.

Example

This swan is white

That swan is white

Every swan I have seen so far is white

Every swan is white

This is a more philosophical induction, however mathematically this does not add up as the conclusion is not entailed from the premises. There are cases where we have black swans which disproves the conclusion as the premises did not take into account ALL swans but every swan the person has seen so far which is only a subset of the set of all swans.

Mathematical induction is a variant that is valid that is based not just on a set example but a rule for deriving cases of our predicate for which our set of premises hold

The general idea of induction is similar to recursion

- Base case: show true for our set of premises for small case (n=0,1)
- Inductive step: a general rule showing that if our premises hold for X, then our premises hold for Y which is constructed from X
- Conclusion: starting with a simple example, and applying the construction of y from existing general set of values we can eventually construct all values in the domain

Example

Prove for n = 0;

Assume it holds for n = k

Prove it also holds for n = k + 1

Conclusion: will hold for n

Basic induction is that principle being applied to natural numbers

- Goal: show P(n) holds for all-natural numbers
- Approach
    - Show that base case holds p (0) holds or p (1) any number
    - Inductive case: if P(k) holds then p(k+1) holds

Example

Example below shows two different approaches to summing, the first approach is more logical however it is a lot more inefficient and uses a lot of memory + stack frames, the second approach is constant time and MUCH more efficient however we need to prove the second one correctly gives the same result as the first one using induction.

Example

Summing the first n natural numbers:

```
sum(n):
  if(n = 0): 0
  else: n + sum(n − 1)
```

Another attempt:

Example

```
sum2(n):
  return n ∗ (n + 1)/2
```

Let $P(n)$ be the proposition that:

$$P(n): \quad \sum_{i=0}^{n} i = \frac{n(n+1)}{2}.$$

We will use proof by induction to prove the predicate above

Show P (0) holds

0 = 0(0+1)/2 = 0 TRUE

Assume P (k) holds

P(k) = k(k+1)/2

Show P (k+1) holds

P(k+1) = (k + 1) * (k+2)/2 = p(k) + k + 1

P(k)+k+1 = (k^2+k)/2 + k + 1 = ((k+1) (k+2))/2

(k^2+k+2k+1)/2 = (k^2+2k+k+1)/2

Therefore, it is true for p (k+1)

Hence true by induction.

Lecture 4

Assignment 1 available at the end of this week.

- **Due Sunday 17th March, 23:59**
- Submit single pdf via webCMS/give
- Worth 20%
- Collaboration ok, but final result should be your own work.

LEARN LATEX

Example

Let $P(n)$ be the proposition that:

$$P(n): \quad \sum_{i=0}^{n} i = \frac{n(n+1)}{2}.$$

We will show that $P(n)$ holds for all $n \in \mathbb{N}$ by induction on n.

Proof.

[B] $P(0)$, i.e.

$$\sum_{i=0}^{0} i = \frac{0(0+1)}{2}$$

[I] $\forall k \geq 0\,(P(k) \rightarrow P(k+1))$, i.e.

$$\sum_{i=0}^{k} i = \frac{k(k+1)}{2} \quad \rightarrow \quad \sum_{i=0}^{k+1} i = \frac{(k+1)(k+2)}{2}$$

(proof?) □

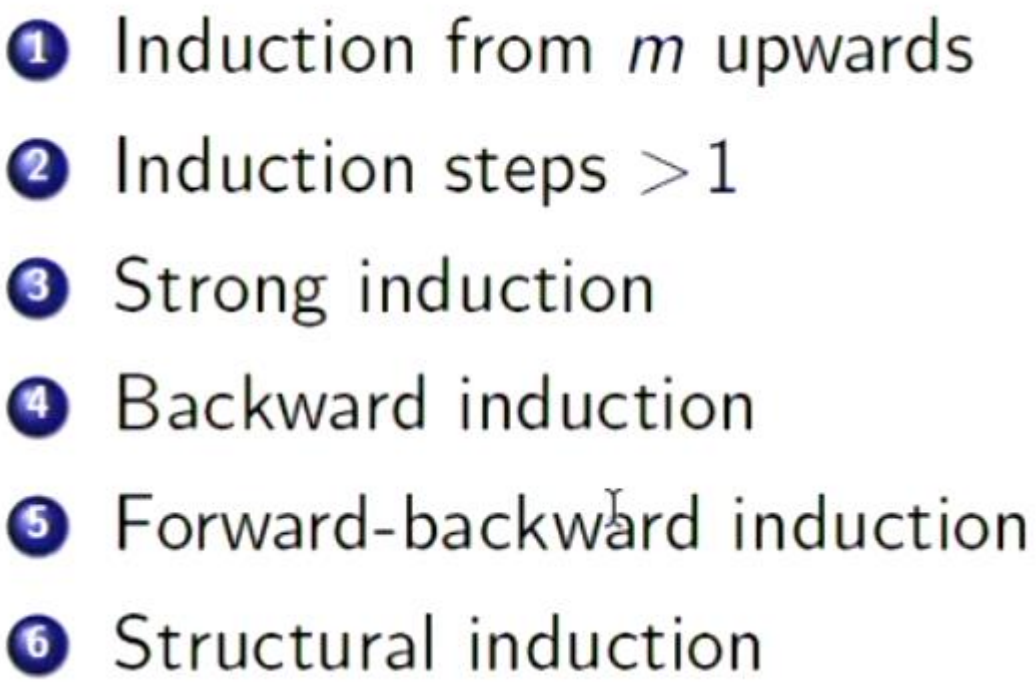
Proof.

Inductive step [I]:

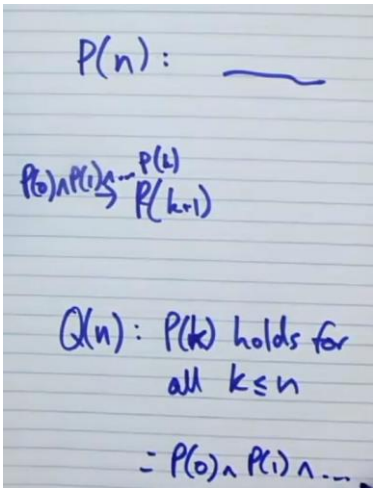
$$\begin{aligned}
\sum_{i=0}^{k+1} i &= \left(\sum_{i=0}^{k} i\right) + (k+1) \\
&= \frac{k(k+1)}{2} + (k+1) \quad \text{(by the inductive hypothesis)} \\
&= \frac{k(k+1) + 2(k+1)}{2} \\
&= \frac{(k+1)(k+2)}{2}
\end{aligned}$$

Variations on induction

Covering different cases in a more sensible way



1. Induction from m upwards is
    a. Holds from some value >= m only
    b. Prove holds for m
    c. Then perform regular induction
    d. Example for all n >= 1, number 8^n – 2 ^n is divisible by 6
    e. Base case is now 1 not 0
2. Induction steps > 1
    a. If our increment is more than 1 (e.g. Even numbers 2)
    b. Prove holds for m
    c. Then assume k, and prove k + l works, were l is our increment
    d. Concludes holds for every l'th element
3. Strong induction
    a. This is a version where the inductive hypothesis is stronger, rather than saying p(k) holds for a single value we use all values up to k
    b. Prove holds for m
    c. If it holds for all values from [m, k] assumption, prove works for k + 1 for all k > m
    d. Concludes holds for all n >= m



Claim: All integers ≥ 2 can be written as a product of primes.

[B] 2 is a product of primes

[I] If all x with $2 \leq x \leq k$ can be written as a product of primes, then $k+1$ can be written as a product of primes, for all $k \geq 2$

Proof for [I]?

4. Negative integers, backward induction
    a. Induction can be conducted over any subset of integers with least element, so we can work with negatives
    b. One can apply the induction in the opposite direction p(m) → p(m-1)
    c. To prove in general, we need to prove it works in both directions, increment up and increment down
5. Forward-Backward induction (extremely important for the analysis of algorithms
    a. To prove p(n) for all n >= k0
    b. Verify P(k0)
    c. Prove P(ki) for infinitely many k0 < k1 < k2 …. < kn
    $P(k_1) \to P(k_1-1) \to P(k_1-2) \to \ldots \to P(k_0+1)$
    $P(k_2) \to P(k_2-1) \to P(k_2-2) \to \ldots \to P(k_1+1)$
    d. Fills in all the gaps
6. Structural induction
    a. Basic induction allows us to assert properties over all-natural numbers and uses the recursive definition of natural numbers
    b. The induction schemes can be applied to any partially ordered set in general, especially ones which have been defined recursively
    c. The basic approach is the same
        i. Verify that the property holds for all minimal objects that have no predecessors (base case). These are very simple objects allowing immediate verification
        ii. Verify the inductive case for any given object if the property holds for all its predecessors then it holds for the object itself.

Example: induction on alphabet

Recall definition of Σ^*:

$$\lambda \in \Sigma^*$$

If $w \in \Sigma^*$ then $aw \in \Sigma^*$ for all $a \in \Sigma$

Show that P(w) holds for all w in the alphabet

Base case: prove p(lambda) holds

Inductive case: if p(w) holds then p(aw) holds for all a that are symbols

Formal definition of concatenation:

(concat.B) $\lambda v = v$

(concat.I) $(aw)v = a(wv)$

Formal definition of length:

(length.B) $\text{length}(\lambda) = 0$

(length.I) $\text{length}(aw) = 1 + \text{length}(w)$

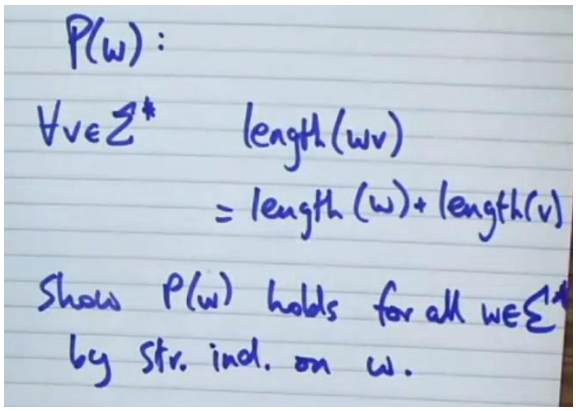
Prove:

$$\text{length}(wv) = \text{length}(w) + \text{length}(v)$$

Formal proof

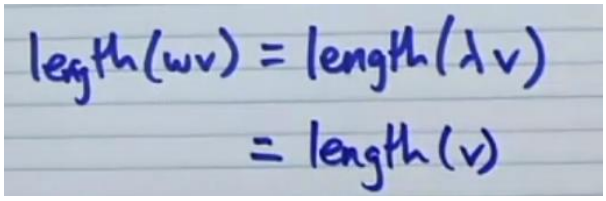
Let P(w) be the proposition that for all words v in an alphabet

Length(wv) = length(w) + length(v)



Base case empty words (lambda)

W = lambda



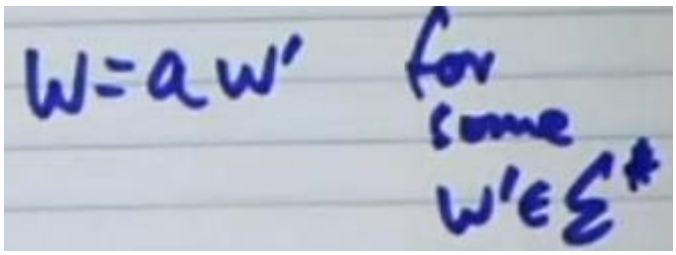
This is true from the base definition of concatenation above

Length(v) = 0 + length(v)

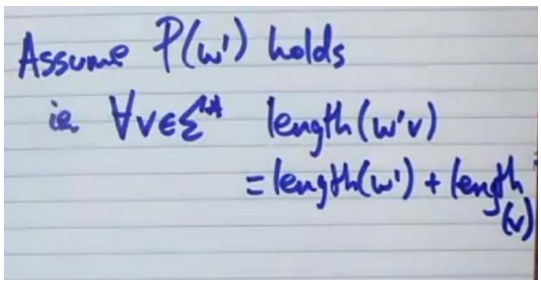
= length(lambda) + length(v) (w = lambda)

Our base case holds! P(lambda) holds

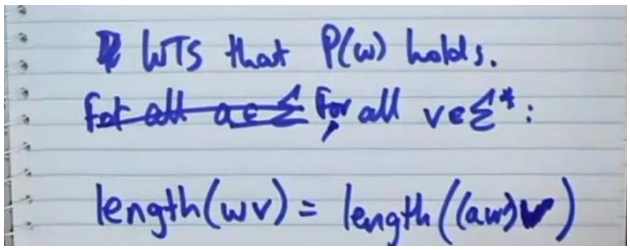
Inductive case

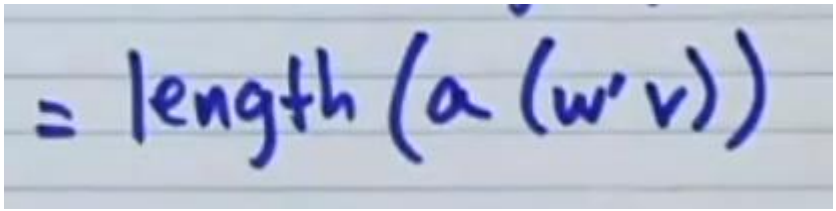


If P(w) holds prove p(w') holds

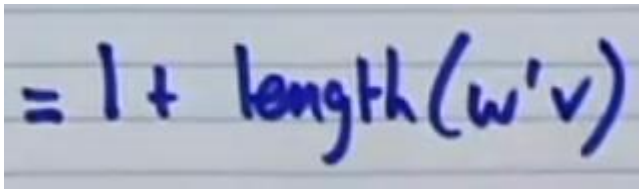


Our inductive hypothesis

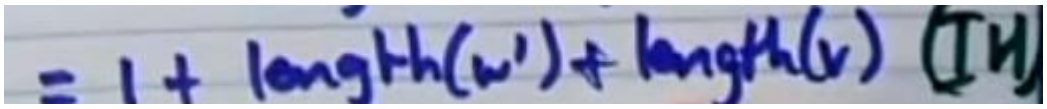




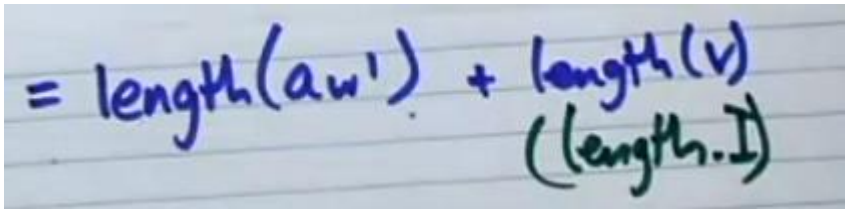
This is true from the inductive definition of concatenation



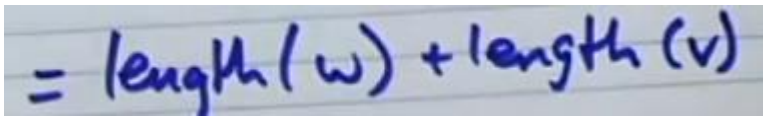
This is true from the inductive definition of length



This is from our inductive hypothesis



This is from the reverse of our inductive rule of length (1 + length w') = length (aw')



Finished proof p(w) holds

Proven p (lambda) holds, proven p(w') → p(aw')

Therefore, by principle of mathematic induction p(w) holds. We used induction on a word structure

Example 2 reversing a string

definition

Define reverse : $\Sigma^* \to \Sigma^*$:

$$\textbf{(rev.B)}\quad \text{reverse}(\lambda) = \lambda,$$

$$\textbf{(rev.I)}\quad \text{reverse}(a \cdot w) = \text{reverse}(w) \cdot a$$

Theorem

For all $w, v \in \Sigma^*$, $reverse(wv) = reverse(v) \cdot reverse(w)$.

Proof: By induction on w...

[B]

$$\begin{aligned} \text{reverse}(\lambda v) &= \text{reverse}(v) && \text{(concat.B)} \\ &= \text{reverse}(v)\lambda && (*) \\ &= \text{reverse}(v)\text{reverse}(\lambda) && \text{(reverse.B)} \end{aligned}$$

[I]

$$\begin{aligned} \text{reverse}((aw')v) &= \text{reverse}(a(w'v)) && \text{(concat.I)} \\ &= \text{reverse}(w'v) \cdot a && \text{(reverse.I)} \\ &= \text{reverse}(v)\text{reverse}(w') \cdot a && \text{(IH)} \\ &= \text{reverse}(v)\text{reverse}(aw') && \text{(reverse.I)} \end{aligned}$$

Mutual recursion

We can employ a technique of two procedures calling each other. It is designed so that each consecutive call refers to even smaller parameters (e.g. List calls sub lists), this eventually leads to process termination.

Example

Fibonacci numbers can be defined as

Base case 1: f (1) = 1

Base case2: g (1) = 1

Recursive case 1: f(n) = f(n-1) + g(n-1)

Recursive case 2: g(n) = f(n-1)

F depends on G, and G depends on F

In matrix form:

$$\begin{pmatrix} f(n) \\ g(n) \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} f(n-1) \\ g(n-1) \end{pmatrix}$$

Corollary:

$$\begin{pmatrix} f(n) \\ g(n) \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^n \begin{pmatrix} f(0) \\ g(0) \end{pmatrix}$$

This is a very quick way to compute the n'th Fibonacci numbers.

Recursively it takes 2^n steps to compute, however this approach is more iterative as matrix exponentiation is very fast with computers, and is log(n) time which is exponentially faster

What is assessible?

- Recursive definitions
- Structural induction

Propositional logic

A more formal approach to logic, computational oriented mathematics

Well-formed formulas

Let $\text{PROP} = \{p, q, r, \ldots\}$ be a set of propositional letters.
Consider the alphabet

$$\Sigma = \text{PROP} \cup \{\top, \bot, \neg, \wedge, \vee, \rightarrow, \leftrightarrow, (,)\}.$$

The **well-formed formulas** (wffs) over PROP is the smallest set of words over Σ such that:

- $\top$, $\bot$ and all elements of PROP are wffs
- If φ is a wff then $\neg\varphi$ is a wff
- If φ and ψ are wffs then $(\varphi \wedge \psi)$, $(\varphi \vee \psi)$, $(\varphi \rightarrow \psi)$, and $(\varphi \leftrightarrow \psi)$ are wffs.

Example if P and T are well-formed formulas

The following are well-formed formulas:

- $(p \wedge \neg\top)$
- $\neg(p \wedge \neg\top)$
- $\neg\neg(p \wedge \neg\top)$

The following are **not** well-formed formulas:

- $p \wedge \wedge$
- $p \wedge \neg\top$
- $(p \wedge q \wedge r)$
- $\neg(\neg p)$

It is very strict similar to code (no semi-colon) fails, in here needs to be as strict

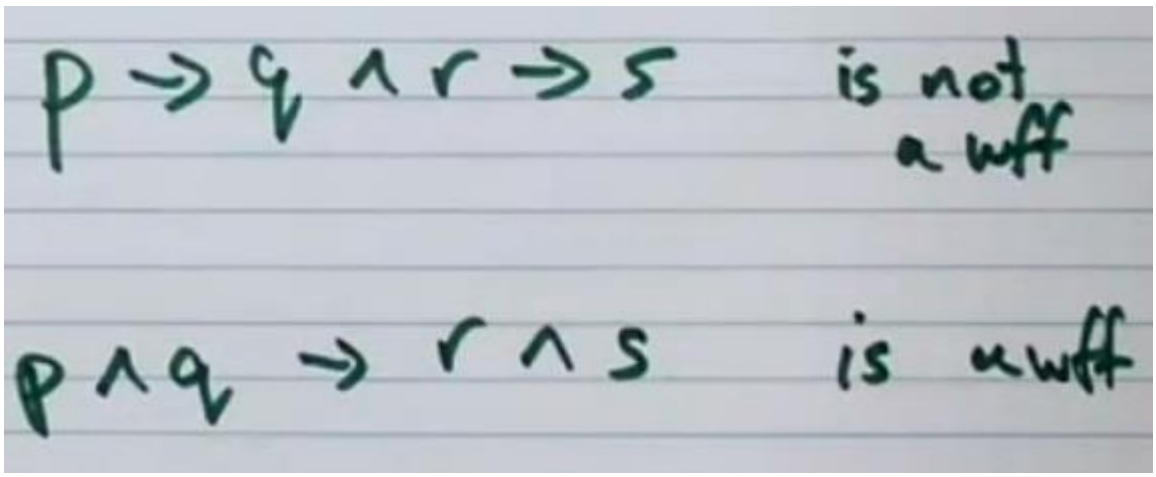
Conventions

To aid readability of some conventions and binding rules can and will be used

- Brackets can be omitted if there is no ambiguity (e.g. p AND q)
- Not binds more tightly than AND or OR, which bind more tightly than implies or bi-implies, for example
  (e.g. $p \wedge q \rightarrow r$ instead of $((p \wedge q) \rightarrow r)$

Other conventions (rarely used/assumed in this course):

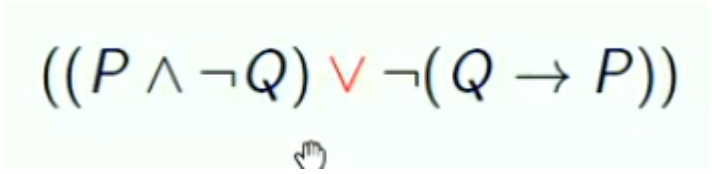
- $'$ or $\bar{\ }$ for $\neg$
- $+$ for $\vee$
- $\cdot$ or juxtaposition for $\wedge$
- $\wedge$ binds more tightly than $\vee$
- $\wedge$ and $\vee$ associate to the left: $p \vee q \vee r$ instead of $((p \vee q) \vee r)$
- $\rightarrow$ and $\leftrightarrow$ associate to the right: $p \rightarrow q \rightarrow r$ instead of $(p \rightarrow (q \rightarrow r))$

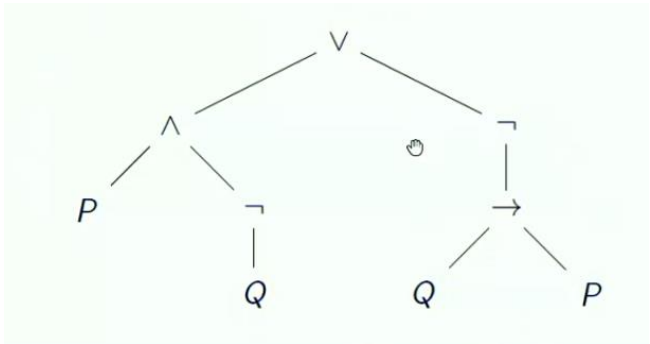


Parse trees

The structure of well-formed formulas and the grammar defined syntaxes can be shown with a parse tree.

Example, consider the two formulas conjoined by a logical OR





Parse tree allows evaluation of our propositional formulas. By going up the leaves to the root

Formal definitions for parse trees

A parse tree is either:

- (B) A node containing $\top$;
- (B) A node containing $\bot$;
- (B) A node containing a propositional variable;
- (R) A node containing $\neg$ with a single parse tree child;
- (R) A node containing $\wedge$ with two parse tree children;
- (R) A node containing $\vee$ with two parse tree children;
- (R) A node containing $\rightarrow$ with two parse tree children; or
- (R) A node containing $\leftrightarrow$ with two parse tree children.

This is also a recursive definition for parse tree which is also a well-formed formula

Boolean Algebra

The formulas p AND q is different to q AND p however they have the same result

A Boolean algebra is a structure of (set T, 3 operations, 2 constants),

$$(T, \vee, \wedge, ', 0, 1)$$

Where

- 0,1 are elements of the underlying set T
- Or joins: T * T → T
- AND meets: T * T → T
- ‘complements: T → T

And the following law holds for all x, y, z in our underlying set T

commutative:
- $x \vee y = y \vee x$
- $x \wedge y = y \wedge x$

associative:
- $(x \vee y) \vee z = x \vee (y \vee z)$
- $(x \wedge y) \wedge z = x \wedge (y \wedge z)$

distributive:
- $x \vee (y \wedge z) = (x \vee y) \wedge (x \vee z)$
- $x \wedge (y \vee z) = (x \wedge y) \vee (x \wedge z)$

identity: $x \vee 0 = x, \quad x \wedge 1 = x$

complementation: $x \vee x' = 1, \quad x \wedge x' = 0$

Laws of Boolean algebra follow from laws of set operations.

X is the universal set in here, these all follow set operation laws

The set of subsets of a set X:

- $T : \mathrm{Pow}(X)$
- $\wedge : \cap$
- $\vee : \cup$
- $' : {}^c$
- $0 : \emptyset$
- $1 : X$

Example of Boolean algebra

The two element Boolean algebra

$$\mathbb{B} = (\{\mathtt{true}, \mathtt{false}\}, \&\&, \|, !, \mathtt{false}, \mathtt{true})$$

! true = false

! false = true;

True && true = true;

True || true = true;

We often use B for the two-element set {true, false}.

For simplicity this can be abbreviated to {T, F} or {0, 1}.

Example

Cartesian products of B that is n-tuples of 0's and 1's with Boolean operations, for example B^4:

$$\begin{aligned}
\textit{join:} \quad & (1,0,0,1) \vee (1,1,0,0) = (1,1,0,1) \\
\textit{meet:} \quad & (1,0,0,1) \wedge (1,1,0,0) = (1,0,0,0) \\
\textit{complement:} \quad & (1,0,0,1)' = (0,1,1,0) \\
0: \quad & (0,0,0,0) \\
1: \quad & (1,1,1,1).
\end{aligned}$$

Think of bits!!!

Example

Functions from any set S → B denoted by Map (S, B)

If f, g: S → B then

- $(f \vee g) : S \longrightarrow \mathbb{B}$ is defined by $s \mapsto f(s) \parallel g(s)$
- $(f \wedge g) : S \longrightarrow \mathbb{B}$ is defined by $s \mapsto f(s)$ && $g(s)$
- $f' : S \longrightarrow \mathbb{B}$ is defined by $s \mapsto !f(s)$
- $0 : S \longrightarrow \mathbb{B}$ is the function $f(s) = \texttt{false}$
- $1 : S \longrightarrow \mathbb{B}$ is the function $f(s) = \texttt{true}$

Example

If we use our initial example of $(T, \vee, \wedge, ', 0, 1)$ as a Boolean algebra, we can than take the dual (dual algebra) $(T, \wedge, \vee, ', 1, 0)$ by swapping everything around. Exact same as the principle of duality!

- $T : \mathrm{Pow}(X)$
- $\wedge : \cup$
- $\vee : \cap$
- $' : {}^{c}$
- $0 : X$
- $1 : \emptyset$

Every finite Boolean algebra satisfies the following property

|T| = 2^k for some k. (the cardinality of our set will always be 2^k)

All algebras with the same number of elements are isomorphic, (structurally similar)

written $\simeq$.

Therefore, by studying one such algebra we can describe the properties of all

The algebras mentioned above are all of this form

- n-tuples $\simeq \mathbb{B}^n$
- $\mathrm{Pow}(S) \simeq \mathbb{B}^{|S|}$
- $\mathrm{Map}(S, \mathbb{B}) \simeq \mathbb{B}^{|S|}$

Boolean algebra as the calculus of two values is fundamental to computer circuits and programming, for example encoding subsets as bit vectors.

Lecture 5

Boolean Algebra

Below are laws that hold for Boolean algebra

and the following laws hold for all $x, y, z \in T$:

commutative:
- $x \vee y = y \vee x$
- $x \wedge y = y \wedge x$

associative:
- $(x \vee y) \vee z = x \vee (y \vee z)$
- $(x \wedge y) \wedge z = x \wedge (y \wedge z)$

distributive:
- $x \vee (y \wedge z) = (x \vee y) \wedge (x \vee z)$
- $x \wedge (y \vee z) = (x \wedge y) \vee (x \wedge z)$

identity: $x \vee 0 = x, \quad x \wedge 1 = x$

complementation: $x \vee x' = 1, \quad x \wedge x' = 0$

These rules are almost identical to laws of set operation.

The laws of set operation fall under a Boolean algebra.

Some examples of Boolean algebras include

- Cartesian products of B
- Function from a set S to B
- Sets of natural number

The following derived laws are also the same as for set laws as Boolean algebra.

Derived laws

The following are all derivable from the Boolean Algebra laws.

Idempotence	$x \wedge x = x$
	$x \vee x = x$
Double complementation	$(x')' = x$
Annihilation	$x \wedge 0 = 0$
	$x \vee 1 = 1$
de Morgan's Laws	$(x \wedge y)' = x' \vee y'$
	$(x \vee y)' = x' \wedge y'$

Principle of duality

If E is an expression for some Boolean algebra made up by ^, v, ‘, 0, 1, then we can take the dual of that expression from the following dual(E) where we swap ^, v.

If we can show E1 = E2 holds in all Boolean algebras, then dual(E1) = dual(E2) holds in all Boolean algebras. So, if we prove E holds, we can also say the dual(E) holds.

A Boolean Algebra expression is defined as

- 0,1 are expressions
- A variable x, y, … are expressions
- If E is an expression then E' is an expression
- If E1 and E2 are expressions then (E1 ^ E2) and (E1 v E2) are expressions

If we say EXPRESSIONS is the set of all expressions in Boolean algebra, we define dual as the following, dual: EXPRESSIONS → EXPRESSIONS as

- Dual (0) = 1, Dual (1) = 0
- Dual (x) = x for all variables x
- Dual(E') = Dual(E) ' for all expressions E
- Dual ((E1 ^ E2)) = (dual(E1) v dual(E2)) for all expressions E1 and E2
- Dual ((E1 v E2)) = (dual(E1) ^ dual(E2)) for all expressions E1 and E2

Example

Dual ((x v (x ^ y))) = (dual(x) ^ dual (x ^ y))

= (x ^ (dual(x) v dual(y))

= (x ^ (x v y))

To perform this recursively simply define dual(I) = I and flip the signs for each child portion

This is how compilers can interpret your code and rewrite the code to another language. Essentially what we did was rewrite our initial statement (x v (x ^ y)).

Valuations

A truth assignment (or model) is a function v: Prop → B

We can extend v to a function $[\![\cdot]\!]_v : \mathrm{WFFs} \to \mathbb{B}$ recursively:

In the assignment essentially our model was the WIFI configuration and we mapped our configuration recursively.

- [T] = true (top)
- [F] = false (bottom)
- [p] = v(p) (evaluate P)
- [~PSI]v != [psi]v
- [(PSI ^ SIGMA)]v = [PSI]v && [SIGMA]v
- [(PSI v SIGMA)]v = [PSI]v || [SIGMA]v
- [(PSI → SIGMA)]v = ![PSI]v || [SIGMA]v (p → q = !p or q)
- [(PSI ←→ SIGMA)]v = (![PSI]v || [SIGMA]v) && ([PSI]v || ![SIGMA]v)

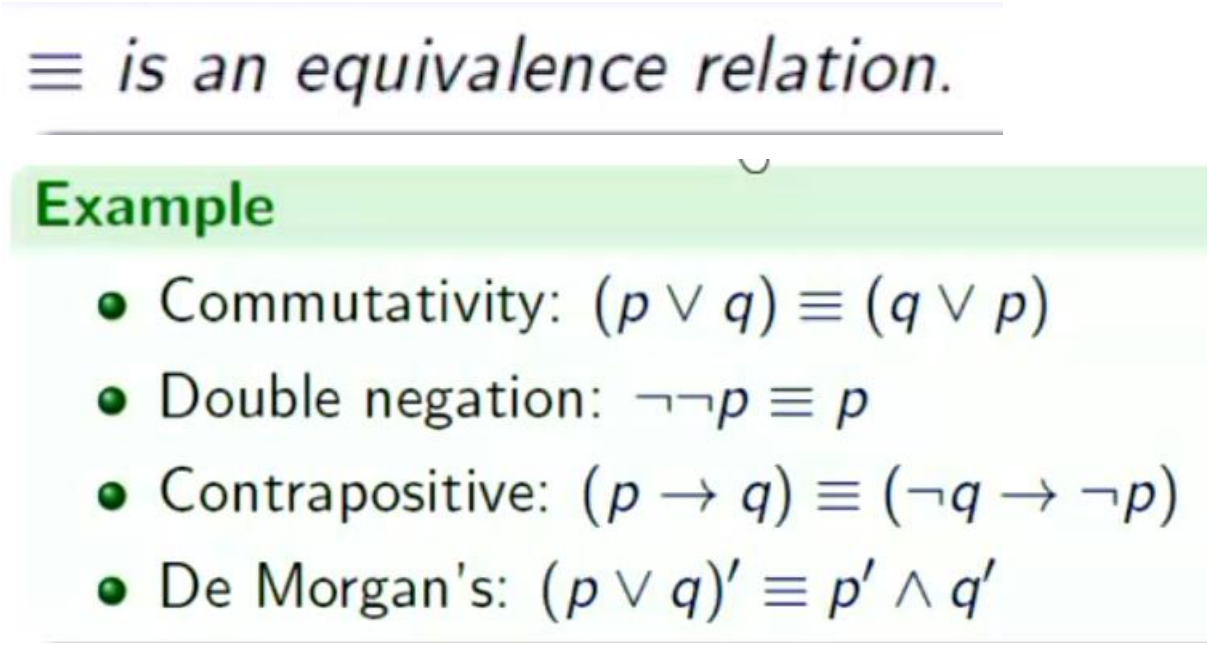
A formula x is

- **Satisfiable** if [x]v = true for some model v (v satisfies x)
- **valid or tautology** if [x]v = true for all models v

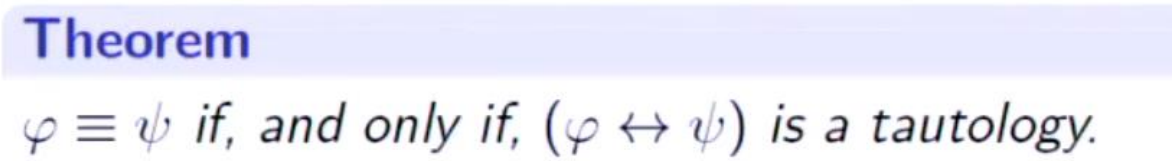
- **unsatisfiable or a contradiction** if [x]v = false for all models v

Logical Equivalence

Two formulas x and y are logically equivalent (x == y) if [x]v = [y]v for all models v.



We can say x is equivalent to y, if and only if x bi implies y is a tautology

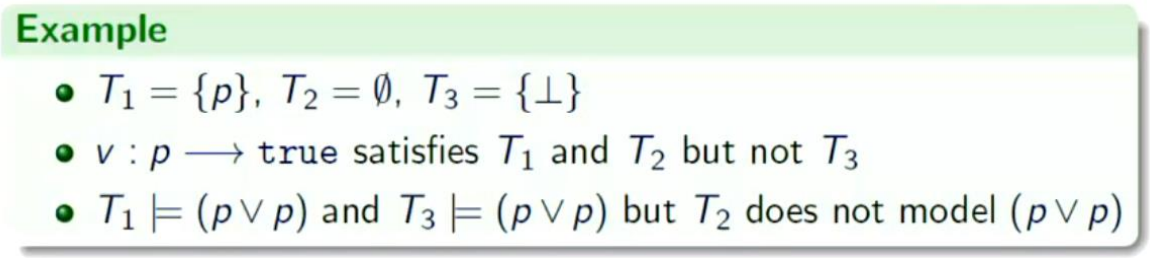


Theories and entailment

A set of formulas is a theory

A model v satisfies a theory T if [x]v = true for some x in T

A theory T entails a formula x, T=| x, if [x]v = true for all models v which satisfy T



Theorem

The following are equivalent:

- $\varphi_1, \varphi_2, \ldots, \varphi_n \models \psi$
- $\emptyset \models ((\varphi_1 \wedge \varphi_2) \wedge \ldots \varphi_n) \rightarrow \psi$
- $((\varphi_1 \wedge \varphi_2) \wedge \ldots \varphi_n) \rightarrow \psi$ *is a tautology*
- $\emptyset \models \varphi_1 \rightarrow (\varphi_2 \rightarrow (\ldots \rightarrow \varphi_n) \rightarrow \psi)) \ldots)$

Lecture 6

Formal Proofs

Given a theory T and a formula x, how do we show T entails x (t =| x)

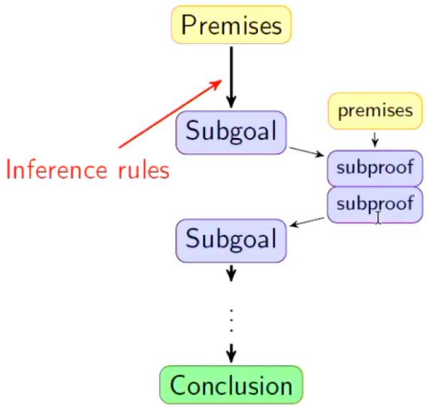
- consider all valuations v (semantic approach) using truth table
- use a sequence of deductive rules to show x is a logical consequence of T (syntactic approach, natural deduction op)

a theory may not be a finite set of formulas, making approach 1 invalid and unfeasible. This is brute forcing and only works with propositional logic, but when we come to first order logic it is totally unreasonable.

A formal way to show a formula is a consequence from a theory is

- highly disciplines way of reasoning (good for computers)
- a sequence of formulas where each step is a deduction based on earlier step (recursive)
- Based entirely on rewriting formulas, no semantic interpretations needed
- No assumptions that are not discharged (all assumptions must be validated by deductive reasoning)

Proof structure



Inference rules basically say “if I have a proof for this, then I have a proof for that” (consequence). Note that all assumptions must be discharged!!!!

In a complicated way “if I have a proof under these assumptions, then I have a proof under these* assumptions).

Yet more complicated form:
If I have a proof of this (under these assumptions)
and I have a proof of this (under these assumptions)
and ...
then I have a proof of that (under these assumptions)

All assumptions must be discharged!!!!!!! And they can be different assumptions!!!

Inference rules

If T is a theory and x is a formula, we write T |- x to denote a proof of x under the assumptions T.

So, an inference rule is a statement of the form

If T1 |- x1 and T2 |- x2 all the way up to Tn |- xn, then T |- y

Our conclusion is y, from the assumptions x

And elimination

If we have a statement A ^ B, then that means under the assumption A ^ B is true, we can eliminate the and sign and conclude both are true (A ^ B) = T only if A = B = T

(A ^ B) |- A, B

And introduction

If we have a proof of A and a proof of B (or given to us), we can combine them together to say they imply A ^ B. Since A is true and B is true then A ^ B is true

A, B |- A ^ B

Or elimination

If we have A OR B, and we want to eliminate the OR sign to a general conclusion for both sides of the OR, we can do a sub proof on A and B and see if they both arrive at the same conclusion, for example if we are trying to prove A v B |- C, we will first check if A concludes C, then check if B concludes C, and if they both conclude to the same thing we can eliminate the or sign.

Or introduction

If we have A which is true, we can introduce ANYTHING and it will still be true since A is true the other side of the or does not matter, so A -| A v UNIVERSE

Modus ponens

If we have A → B and we know A holds, then we can say B holds, for example “if it rains it is wet”, if it is raining, we can assume it is wet, HOWEVER WE CANNOT DO THE REVERSE WE CANNOT ASSUME ITS RAINING JUST BECAUSE IT IS WET!!! B DOES NOT IMPLY A IN THIS SITUATION

A → B

A |- B here

Modus Tollens

If we have A → B and we know B does not hold, then we can say A does not hold, for example “if it rains it is wet”, if it is not wet, we can assume it is not raining as that is the implication we have given between A and B

A → B

! B |- ! A

→ Introduction

If we make an assumption A and arrive to conclusion B, then we can say A → B

ALL ASSUMPTIONS MUST BE DISCHARGED

There are many proof systems defined by the default axioms (“free” premises), and inference rules, these include

- Natural deduction
- Hilbert systems
- Sequent calculus
- Resolution

Properties of proof systems

- Soundness
  - The system only provides valid statements, if T|-x then T|= x
- Completeness
  - The system can prove any valid statement, if T |= x then T |- x

Soundness is straightforward, we check every axiom and inference rule, completeness is trickier, and may not even exist.

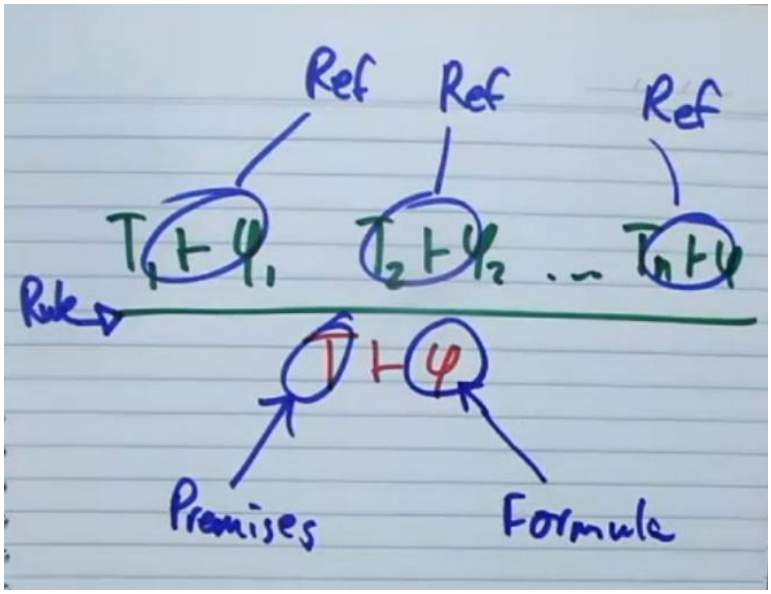
Natural Deduction

Using the rules stated above we can perform a proof system known as natural deduction. Natural deduction is designed to mirror natural reasoning

There are

- No axioms (no inherent truths)
- 15 inference rules to introduce/eliminate Boolean operators from chain of reasoning

Operator	Introduction	Elimination
$\wedge$	$\wedge$-I	$\wedge$-E1 $\wedge$-E2
$\vee$	$\vee$-I1 $\vee$-I2	$\vee$-E
$\rightarrow$	$\rightarrow$-I	$\rightarrow$-E
$\leftrightarrow$	$\leftrightarrow$-I	$\leftrightarrow$-E1 $\leftrightarrow$-E2
$\neg$	$\neg$-I	$\neg$-E IP
$\bot$	$\neg$-E	X



FINAL EXAM ALLOWS FOR CHEAT SHEET DOUBLE SIDED A4, WE WILL BE GIVEN LAWS OF SET OPERATION AND NATURAL DEDUCTION!!!

$\leftrightarrow$ Introduction and Elimination

$\leftrightarrow$-introduction:

$$\frac{\begin{array}{cc}[A] & [B] \\ \vdots & \vdots \\ B & A\end{array}}{A \leftrightarrow B}\ (\leftrightarrow\text{-I})$$

$\leftrightarrow$-elimination (1):

$$\frac{A \leftrightarrow B \qquad A}{B}\ (\leftrightarrow\text{-E1})$$

$\leftrightarrow$-elimination (2):

$$\frac{A \leftrightarrow B \qquad B}{A}\ (\leftrightarrow\text{-E1})$$

Lecture 7

CNFs and DNFs are *syntactic* forms:

Literal: A propositional variable or the negation of a propositional variable

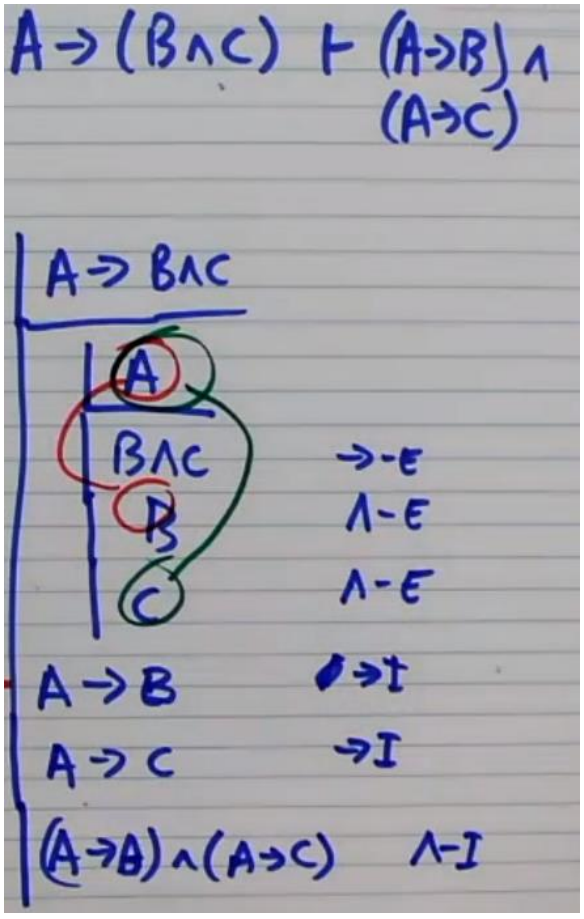
Clause: A CNF-clause is a disjunction ($\vee$) of literals. A DNF-clause is a conjunction ($\wedge$) of literals

CNF/DNF: A formula is in CNF (DNF) if it is a conjunction (disjunction) of CNF-clauses (DNF-clauses).

NB

Formulas in DNF are easy to check for satisfiability. Formulas in CNF are easy to check for validity.

We can discharge a proof in natural deduction more than once, in the picture below, we discharge A to B and A to C



Need to know for this course

- Difference between syntax and semantics
- CNF/DNF definitions and (any) technique for converting a formula into CNF/DNF
- How to do proofs in Natural Deduction and (any) proof style

Natural deduction is sound and complete.

T |- x if and only if T |= x

The theorem is sound and complete under natural deduction

Predicate Logic

Predicate logic adds expressiveness to propositional logic, it lets us form larger logics from smaller logics. It examines how or why a proposition is true and defines relationships between propositions.

Example

Consider the following

For all x, y in X: (y= x + 1) → (y >= x) where X = {1,2,3}

$$\begin{array}{llll} P_{11} & = \text{“}1 = 1 + 1\text{”} & S_{11} & = \text{“}1 \leq 1\text{”} \\ P_{12} & = \text{“}2 = 1 + 1\text{”} & S_{12} & = \text{“}1 \leq 2\text{”} \\ \vdots & \vdots & \vdots & \vdots \end{array}$$

Final result: $(P_{11} \overset{I}{\rightarrow} S_{11}) \wedge (P_{12} \rightarrow S_{12}) \wedge \cdots \wedge (P_{33} \rightarrow S_{33})$

NB

“Normal arithmetic”, where P_{11} is false, P_{12} is true, etc is one of many possibilities.

For a small set of {1,2,3} we get 18 Propositional variables! What if we used the set of all natural numbers instead?

Final result: $(P_{00} \overset{I}{\rightarrow} S_{00}) \wedge (P_{01} \rightarrow S_{01}) \wedge \cdots$

The final result above is not permitted as it is infinitely sized. To solve this, we use predicate logic.

Predicate logic introduces 5 concepts built on propositional logic to help solve this problem.

- Predicates (Boolean relations) in RED

$$\text{For all } x, y \in X : (y = x+1) \rightarrow (x \leq y)$$

- Functions in YELLOW

$$\text{For all } x, y \in X : (y = x+1) \rightarrow (x \leq y)$$

- Constants in GREEN

$$\text{For all } x, y \in X : (y = x+1) \rightarrow (x \leq y)$$

- Variables in BLUE

$$\text{For all } x, y \in X : (y = x+1) \rightarrow (x \leq y)$$

- Quantifiers in PURPLE

$$\text{For all } x, y \in X : (y = x+1) \rightarrow (x \leq y)$$

Fundamental to interpreting formulas is the domain of discourse, which is the set of "ground objects" we are referring to.

- Predicates are relations on the domain
- Functions are operators on the domain
- Constants are named elements of the domain
- Variables are unnamed elements of the domain (placeholders)
- Quantifiers are the range over domain elements

The range of the quantifiers determine what order logic we are dealing with, if we are dealing only with ground objects then this is first order logic. Second order logic ranges over predicates. Computer objects can be seen as higher order logics.

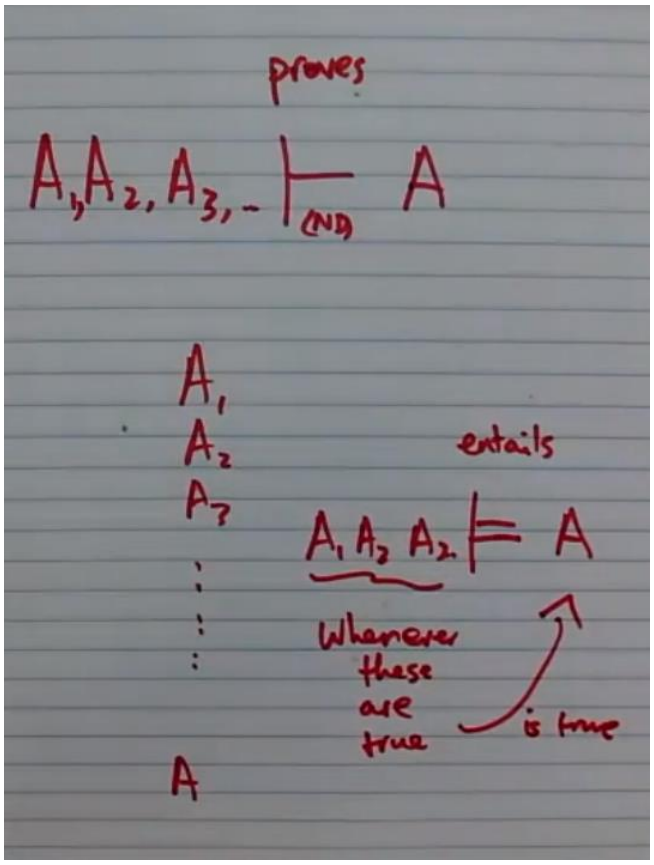
Example

Consider: $\forall x C(x)$ where $C(x)$ represents "x studies COMP2111"
It is `true` if the domain of discourse is the set of students in this room.

FALSE BTW IM STUDYING IT AND IM NOT IN THE ROOM EHM.

The domain of discourse dictates whether a formula is true or false.

The difference between |- and |=



Soundness and complete is when both the proof and entailment are equivalent.

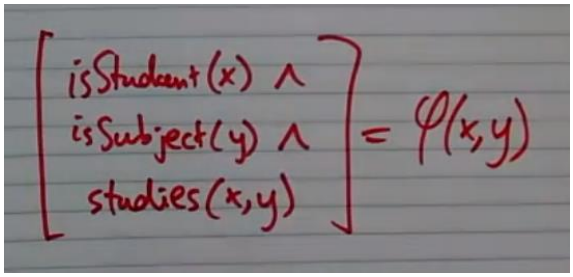
Multiple Domains of Discourse

Is it possible to have multiple domains?

For example: the predicate studies (x, y) representing “x (a student) studies y (a subject)”. This is very useful for programs.

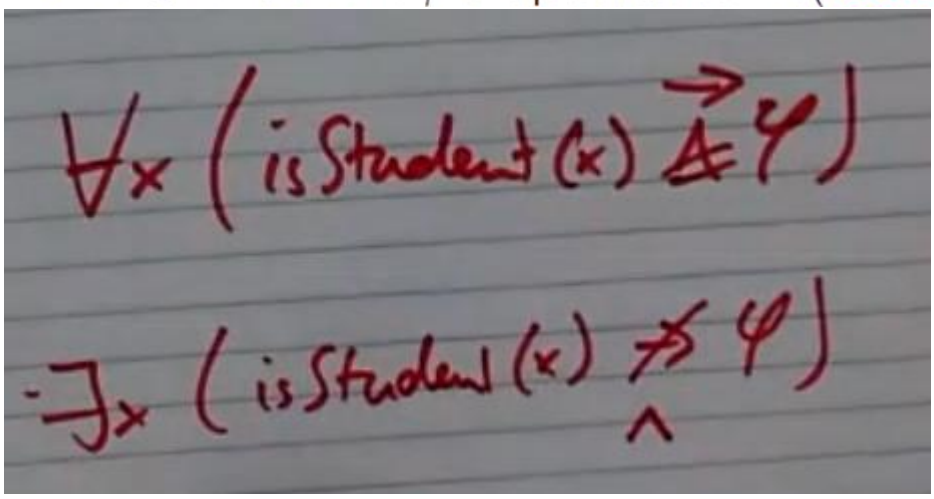
This is possible

- We can take x U y as the domain (students union subject)
- We can use unary predicates, e.g. isStudent(x) to restrict our domain.



Unary predicates allow us to force x to be a student in the domain, and y to be a subject in the domain.

To restrict quantifiers (applies to any subset of the domain defined by a unary predicate)

- $\exists x \in \text{STUDENTS} : \varphi$ is equivalent to: $\exists x(\text{isStudent}(x) \wedge \varphi)$
- $\forall x \in \text{STUDENTS} : \varphi$ is equivalent to: $\forall x(\text{isStudent}(x) \rightarrow \varphi)$
- 

In a domain

- Function outputs, constants and variables are interpreted as the elements of the domain
- Predicates are truth functional mapping of elements of the domain to true or false (a relation)
- Quantifiers (and the Boolean connectives) are predicate operators: they transform predicates into other predicates.

Example

Consider the following predicates and constants:

$K(x, y)$: x knows y
$S(x, y)$: x is not the son of y

J: Jon Snow
N: Ned Stark
B: Bran Stark

Domain of discourse: PEOPLE
The following are OK:

- $S(B, J)$: Bran is not the son of Jon
- $K(N, J)$: Ned knows Jon
- $\forall x \neg K(J, x)$: Jon Snow knows nothing.

The following however is not okay K (B, S (J, N)): Brian knows that Jon is not the son of Ned. As can be seen in our domain of discourse this is invalid since S (J, N) is not a person!!! We cannot parse in an object as an integer! If we extend our domain of discourse to people AND Facts and add in F (x, y) as the fact that x is not the son of y (functional) then we can instead write it as K (B, F (J, N))

Vocabulary

A vocabulary indicates what predicates, functions and constants we can use to build up our formulas. Very similar to C header files or java interfaces.

A vocabulary V is a set of

- Predicate "symbols" P, Q, …., each with associated arity (number of arguments)
    - No-ary predicate symbols are propositional variables, either true or false
- Function "symbols" f, g, …., each with associated arity (number of arguments)
- Constant "symbols" c, d, …. (also known as 0-arity functions)

Example

$V = \{\leq, +, 1\}$ where $\leq$ is a binary predicate symbol, $+$ is a binary function symbol, and 1 is a constant symbol.

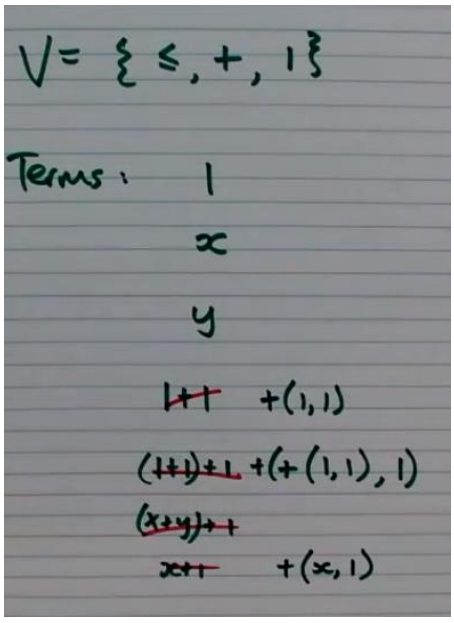
<= is just a binary symbol, with no meaning till assigned, in programming languages for objects we often define what it means to be <= or ==.

Terms

A term is defined as the following

- A variable is a term
- A constant symbol is a term
- If f is a function with arity k, and t1, ….., tk are terms, then f (t1, t2, ……, tk) is a term.
- Terms will be interpreted as elements of the domain of discourse.

For example



Formulas

A **formula of predicate logic** is defined recursively as the following:

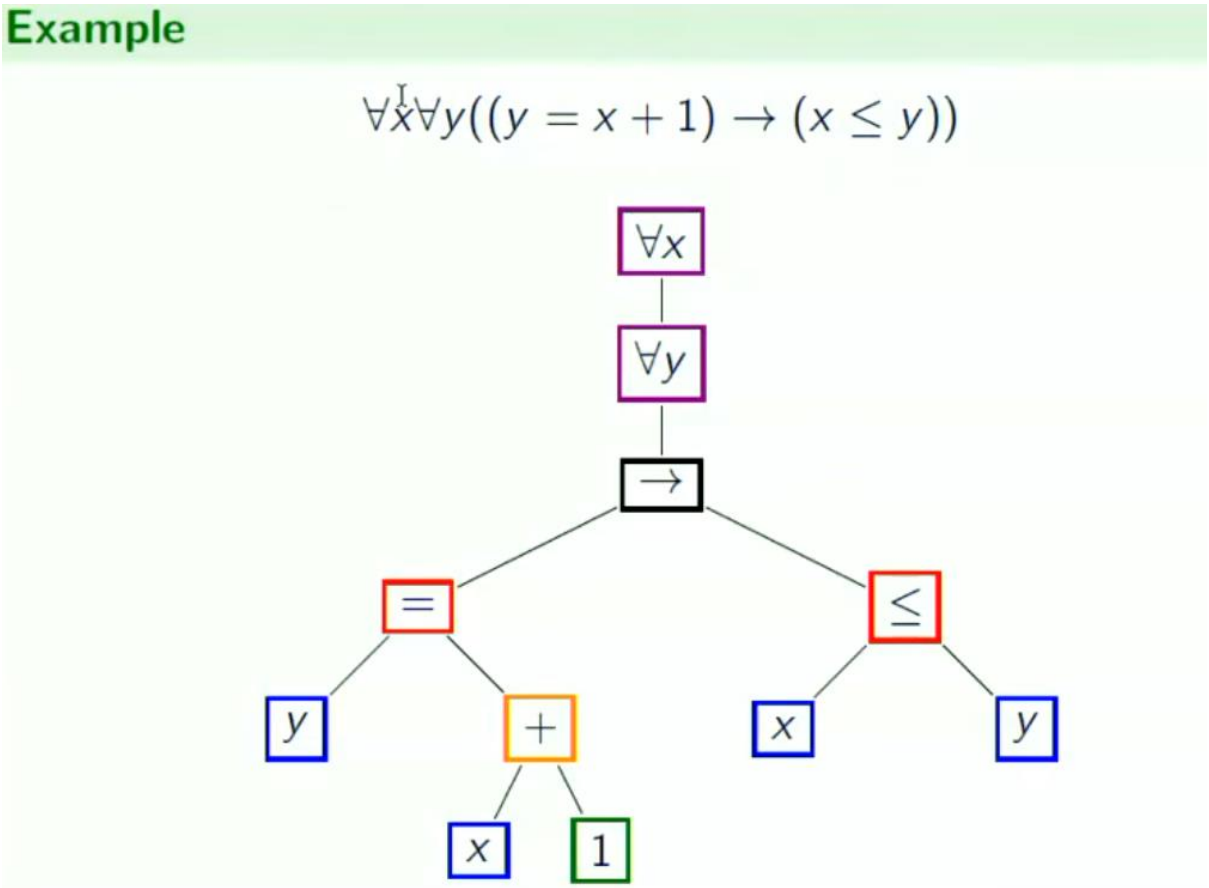
- If P is a predicate symbol with arity k, and t1 … tk are terms, then P (t1, t2 … tk) is a formula
- If t1 and t2 are terms then (t1 = t2) is a formula for equality
- If x and y are formulas then following are formulas (RECURSIVE CASE)
    - ~x
    - (x ^ y)
    - (x v y)
    - (x → y)
    - (x ←→ y)
    - For all variables in x
    - There exists a variable in x
- The base cases are known as atomic formulas, and they play a similar role in the parse tree as propositional variables

Example

y = x + 1

y = +(x,1)

<= (y, x)

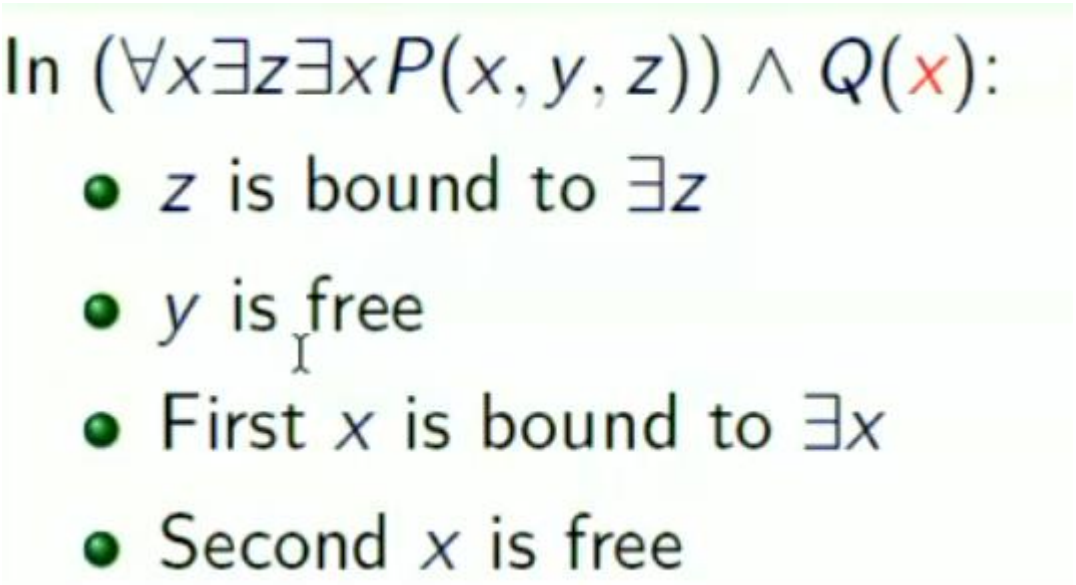


In the parse tree, red nodes are atomic formulas (propositional variables)

Free and Bound variables

A variable is bound to the closest matching quantifier that lies above it in the parse tree. A variable that is not bound is considered free

Example



A formula with no free variables is a sentence, equated to either tree or false

Formally

We can define the set of free variables recursively on the structure of a formula:

FV(x) = {x} for all variables x

FV(c) = EMPTYSET for all constants c

FV (f (t1 … tk)) = FV(t1) U FV(t2) U FV(t3) …. U FV (tk) for all k-ary functions f

FV (P (t1 … tk)) = FV(t1) U FV(t2) U FV(t3) …. U FV (tk) for all k-ary predicates P

FV (t1 = t2) = FV(t1) U FV(t2)

FV (~x) = FV (x)

FV (x ^ y) = FV (x v y) = FV (x → y) = FV (x ←→y) = FV (x) U FV (y)

Lecture 8

An example of a vocabulary could be a database schema

For example, a database schema could have different tables of following

Person

- String name
- String surname
- String address

Employee

- Int id
- String surname

A database table relates a number of attributes in the following

DB = {Person, Employee}

Where Person is a ternary predicate symbol and employee is a binary predicate symbol. And isString and isInteger are unary predicate symbols.

DB = {Person, Employee, isString, isInteger} is our first order vocabulary

Example of formulas in databases

In relational databases, formulas correspond to select queries

For the vocabulary

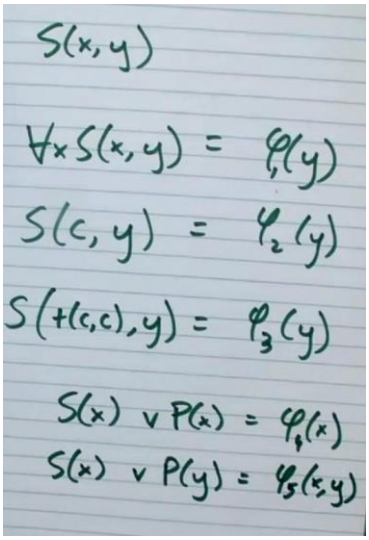
DB = {Person, Employee, isString, isInteger}

Select * from personTable corresponds to find Person (x, y, z) predicate person, and variables x, y, z

Select * from personTable where person.name= “bob” corresponds to find Person (x, y, z) ^ x = “Bob”

Formulas as predicates

Formulas can be viewed as complex predicates; predicates that are built from other predicates by

- Combining them using the Boolean operators
  - 

- "Simplifying" using quantification and term substitutions (projection)

The free variables represent the arity of the predicate, hence the notation f (x1, x2, …, xn)

Note that the variable names matter f(x) and f(y) are different formulas however they will be interpreted as the same predicate.

Example

From binary predicates P and Q; and constant c we can build complex predicates like:

- $\alpha(w, x, y, z) = (P(x, w) \vee Q(y, z)) \wedge (w = y) \wedge (z = c)$
- $\beta(x, y, z) = (P(x, y) \vee Q(y, z)) \wedge (z = c)$
- $\gamma(x, y) = P(x, y) \vee Q(y, c)$
- $\delta(x) = \exists y P(x, y) \vee \exists y Q(y, c)$

Substitution

If t is a term, φ a formula, and $x \notin FV(\varphi)$, then the **substitution of t for x in φ** (denoted $\varphi[t/x]$) is the formula obtained by replacing every free occurrence of x with t.

Alternatively, if the free variables are listed, substituting t for x in PSI(x) can be written as PSI(t).

Models

Predicate formulas are interpreted in Models

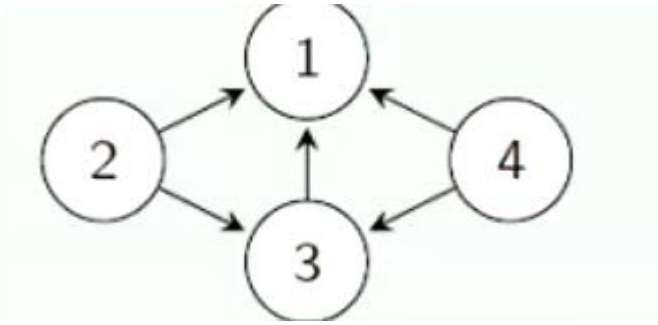
Given a vocabulary V a model M defines

- A (non-empty) domain $D = \text{Dom}(\mathcal{M})$
- For every predicate symbol $P \in V$ with arity k: a k-ary relation $P^{\mathcal{M}}$ on D
- For every function symbol $f \in V$ with arity k: a function $f^{\mathcal{M}} : D^k \to D$
- For every constant symbol $c \in V$: an element, $c^{\mathcal{M}}$ of D

Remember, predicates are syntactic and have more to do with formulas, and relations are semantic and have more to do with models!

Example

For the vocabulary V = {<=, +, 1} the following are models

- Natural numbers with the standard definitions of <=, + and 1
- {0, 1, 2, 3, 4} with the standard definitions of <=, 1 and m + n (mod 5)
- The directed graph G = (V, E) shown below with <= as E; and v + w defined to be w.
  - 

- A model simply just gives us an interpretation of our symbols in our vocabulary and define our domain of discourse

In the example of our database, viewed as a model isString and isInteger are defined by what values are permitted in each of the columns, it is an input sanitizer

Person		
Name	**Surname**	**Address**
Arya	Stark	Winterfell
Jon	Snow	Winterfell
Cersei	Lannister	King's Landing

Employee	
ID	**Surname**
31415	Tyrell
27182	Lannister
16180	Targaryen

Environments

Given a model M, an environment for M (or lookup table) is a function from the set of variables to Dom(M). This essentially handles all our variables for our models.

Given an environment e, we denote e [x |→c] the environment that agrees with e everywhere except possibly at x where it has value c. it is similar to mapping elements to domain.

Interpretations

An interpretation is a pair (M, e) where M is a model and e are an environment. It assigns to every formula a truth value. It maps terms to elements of Dom(M) recursively in the following

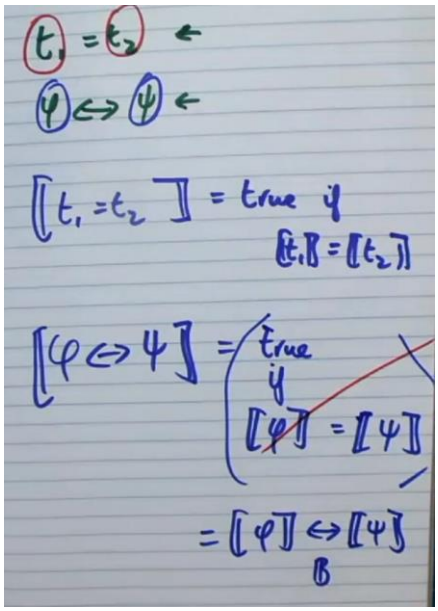
- $[\![x]\!]^{\eta}_{\mathcal{M}} = \eta(x)$
- $[\![c]\!]^{\eta}_{\mathcal{M}} = c^{\mathcal{M}}$
- $[\![f(t_1, \ldots, t_k)]\!]^{\eta}_{\mathcal{M}} = f^{\mathcal{M}}([\![t_1]\!]^{\eta}_{\mathcal{M}}, \ldots, [\![t_k]\!]^{\eta}_{\mathcal{M}})$

our variables are defined in the environment, and our constants are defined in our model, and we recursively use these definitions to evaluate functions in our model and environment. In the third dot point, we interpret all the terms in our model/environment, and then we apply our function implementation.

Example

An interpretation $(\mathcal{M}, \eta)$ maps formulas to $\mathbb{B}$ recursively as follows:

- $[\![P(t_1, \ldots, t_k)]\!]^{\eta}_{\mathcal{M}} = \texttt{true}$ if $P^{\mathcal{M}}([\![t_1]\!]^{\eta}_{\mathcal{M}}, \ldots, [\![t_k]\!]^{\eta}_{\mathcal{M}})$ holds.
- $[\![t_1 = t_2]\!]^{\eta}_{\mathcal{M}} = \texttt{true}$ if $[\![t_1]\!]^{\eta}_{\mathcal{M}} = [\![t_2]\!]^{\eta}_{\mathcal{M}}$



We can't use equality to compare formulas, we need to use Boolean operators for that. Equality is only used to compare terms.

- $[\![\forall x \varphi]\!]^{\eta}_{\mathcal{M}} = \texttt{true}$ if $[\![\varphi]\!]^{\eta[x \mapsto c]}_{\mathcal{M}} = \texttt{true}$ for all $c \in \mathrm{Dom}(\mathcal{M})$
- $[\![\exists x \varphi]\!]^{\eta}_{\mathcal{M}} = \texttt{true}$ if $[\![\varphi]\!]^{\eta[x \mapsto c]}_{\mathcal{M}} = \texttt{true}$ for some $c \in \mathrm{Dom}(\mathcal{M})$
- $[\![\varphi]\!]^{\eta}_{\mathcal{M}}$ defined in the same way as Propositional Logic for all other formulas φ. For example $[\![\varphi \wedge \psi]\!]^{\eta}_{\mathcal{M}} = [\![\varphi]\!]^{\eta}_{\mathcal{M}} \&\& [\![\psi]\!]^{\eta}_{\mathcal{M}}$

Notation

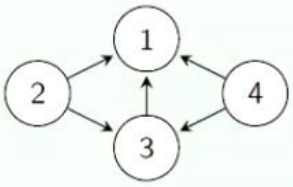
We write $\mathcal{M}, \eta \models \varphi$ if $[\![\varphi]\!]^{\eta}_{\mathcal{M}} = \texttt{true}$

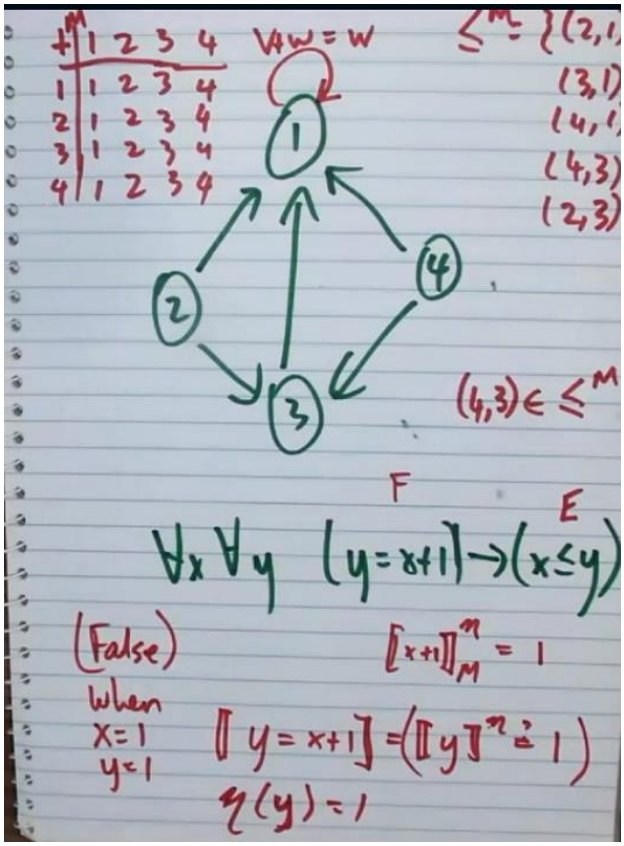
More concrete example

Example

$$\forall x \forall y((y = x + 1) \rightarrow (x \leq y))$$

- $\mathbb{N}$ with the standard definitions of $\leq$, $+$, and 1: `true`
- $\{0, 1, 2, 3, 4\}$ with the standard definition of $\leq$ and 1, and $m + n$ defined as $m + n \pmod 5$: `false`
- The directed graph $G = (V, E)$ shown below with $\leq = E$; and $v + w$ defined to be w.





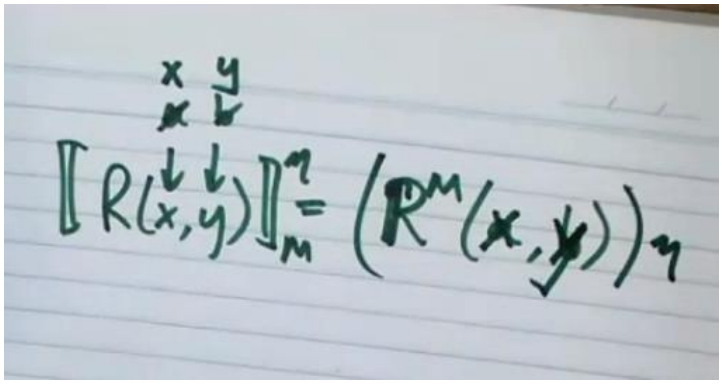
The model interprets everything but the variables, and the environment is used to interpret variables. We use our environment to process our quantifiers.

In the definition of $[\![\varphi]\!]^{\eta}_{\mathcal{M}}$, η is only used to define values for the free variables. In particular, if φ is a sentence then $[\![\varphi]\!]^{\eta}_{\mathcal{M}}$ is independent of η.

Define $[\![\cdot]\!]_{\mathcal{M}}$ by "delaying" the assigning of values to free variables and propagating them out. That is, define:

$$[\![\varphi(x_1, x_2, \ldots, x_n)]\!]_{\mathcal{M}} = [\![\varphi]\!]_{\mathcal{M}}(x_1, x_2, \ldots, x_n)$$

where $[\![\varphi]\!]_{\mathcal{M}} : \mathrm{Dom}(\mathcal{M})^n \to \mathbb{B}$; that is, $[\![\varphi]\!]_{\mathcal{M}}$ is an n-ary relation on $\mathrm{Dom}(\mathcal{M})$.



This matches the perspective of formulas as complex predicates:

$$\varphi(x_1, x_2, \ldots, x_n) \text{ an } n\text{-ary predicate}$$
$$\Downarrow$$
$$[\![\varphi]\!]_{\mathcal{M}} \text{ an } n\text{-ary relation on } \mathrm{Dom}(\mathcal{M})$$

So, in turn, our model is a relation which maps the function on the domain of the model, and the environment will pick tuples from those results and give a Boolean result whether the tuple is true/false.

Database example

Vocabulary: schema for database

Formulas: queries (PSI)

Model: databases (D)

Interpretation: [[PSI]] D is a relation on the Dom(D), i.e. a derived table in D

Environment: "looks up" an entry in a derived table and returns whether the lookup was successful.

- $[\![\varphi]\!]^{\eta}_{\mathcal{D}}$: Success/fail outcome of looking up a specific entry in a query result on $\mathcal{D}$.

Satisfiability, truth, validity

A formula x of predicate logic is:

- Satisfiable if there is some model M and some environment e such that M, e |= x. that there is some interpretation (M, e) that satisfies x
- True in a model M if for all environments n we have M, e |= x
- A logical validity if it is true in all models.
- For sentences the first two definitions are the same, since there is no environment for sentences.

Satisfiability is very easy to prove as we just have to give one case, however validity is much harder as a vocabulary can have an almost infinite set of models so we need to use a different system. Rather than brute force checking each model.

Example

The sentence $\forall x \forall y((y = x + 1) \rightarrow (x \leq y))$ is satisfiable but not a logical validity.

Entailment, Logical equivalence

- A theory T entails a formula x, T |= x, if x is satisfied by any interpretation hat satisfies all formulas in T
- X is logically equivalent to y, x == y, if [[x]] em = [[y]] em for all interpretations (M, e)

Theorem

- $\varphi_1, \ldots, \varphi_n \models \psi$ *if, and only if,* $(\varphi_1 \wedge \cdots \wedge \varphi_n) \rightarrow \psi$ *is a logical validity.*
- $\varphi \equiv \psi$ *if, and only if,* $\varphi \leftrightarrow \psi$ *is a logical validity.*

Natural deduction for predicate logic

Demonstrating a formula is satisfiable or invalid is very easy, we just need to provide an example or counter example. Finding an example is very random, there is no algorithm to generate one we kind of have to make very educated guesses, however showing a formula/entailment is valid is a much more difficult task. We do this with a sound proof system (natural deduction)

We have new inference rules for propositional logic and seven rules for quantifiers and equality

Operator	Introduction	Elimination
$\forall$	$\forall$-I	$\forall$-E
$\exists$	$\exists$-I	$\exists$-E
$=$	$=$-I	$=$-E1 $=$-E2

Arbitrary variables

Formulas of predicate logic involve variables. The new inference rules involve variable manipulation.

A variable is arbitrary if it does not occur (as a free variable) in any undischarged assumption.

An arbitrary variable can be assigned any element of the domain and the formula will still hold.

For all elimination

$\forall$-elimination:

$$\frac{\forall x A(x)}{A(c)}\ (\forall\text{-E})$$

For all introduction

$\forall$-introduction:

$$\frac{\begin{array}{ll} A(c) & (c \text{ is arbitrary}) \\ & (x \text{ not free in } A(c)) \\ & (c \text{ not free in } A(x)) \end{array}}{\forall x A(x)} \ (\forall\text{-I})$$

Simple example

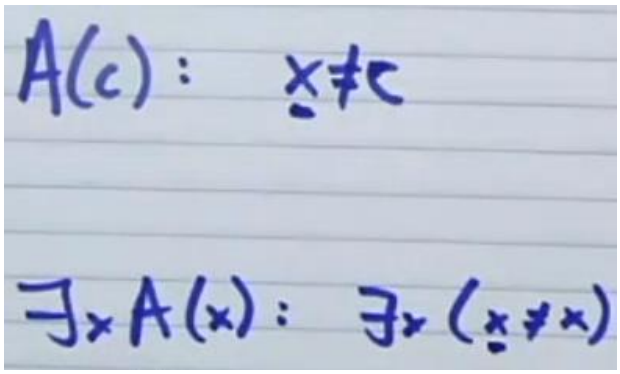
Prove: $\forall x \forall y\, P(x,y) \vdash \forall y \forall x\, P(x,y)$

1	$\forall x \forall y\ P(x,y)$	
2	$\forall y\ P(a,y)$	$\forall$-E 1
3	$P(a,b)$	$\forall$-E 2
4	$\forall x\ P(x,b)$	$\forall$-I 3
	$\forall y \forall x\ P(x,y)$	$\forall$-I 4

Exists introduction

$\exists$-introduction:

$$\frac{A(c) \qquad (x \text{ not free in } A)}{\exists x A(x)} \ (\exists\text{-I})$$



Exists elimination

$\exists$-elimination:

$$\dfrac{\exists x A(x) \qquad \begin{array}{c}[A(c)] \\ \vdots \\ B\end{array} \qquad \begin{array}{l}(x \text{ not free in } A(c)) \\ (c \text{ is arbitrary}) \\ (c \text{ not free in } B)\end{array}}{B}\ (\exists\text{-E})$$

Example

Prove: $\exists x \exists y\, P(x, y) \vdash \exists y \exists x\, P(x, y)$

1. $\exists x \exists y\, P(x, y)$
   2. $\exists y\, P(a, y)$
      3. $P(a, b)$
         4. $\exists x\, P(x, b)$ — $\exists$-I: 3
         5. $\exists y \exists x\, P(x, y)$ — $\exists$-I: 4
      6. $\exists y \exists x\, P(x, y)$ — $\exists$-E: 2, 3–5
7. $\exists y \exists x\, P(x, y)$ — $\exists$-E: 1, 2–6

= elimination and introduction

=-introduction: $\dfrac{}{a = a}\ (=\text{-I})$

=-elimination (1): $\dfrac{a = b \qquad A(a)}{A(b)}\ (=\text{-E1})$

=-elimination (2): $\dfrac{a = b \qquad A(b)}{A(a)}\ (=\text{-E1})$

Introduction rule has no requirements and can be introduced anywhere

Prove: $\vdash \forall x \forall y\, (x = y) \rightarrow (y = x)$

1. $a = b$	
2. $a = a$	=-I
3. $b = a$	=-E1: 1,2
4. $(a = b) \rightarrow (b = a)$	→-I: 1–3
5. $\forall y\, (a = y) \rightarrow (y = a)$	∀-I: 4
6. $\forall x \forall y\, (x = y) \rightarrow (y = x)$	∀-I: 5

Natural deduction is sound and complete for predicate logic:

T |- x if, and only if T |= x

We use proofs to show validity.

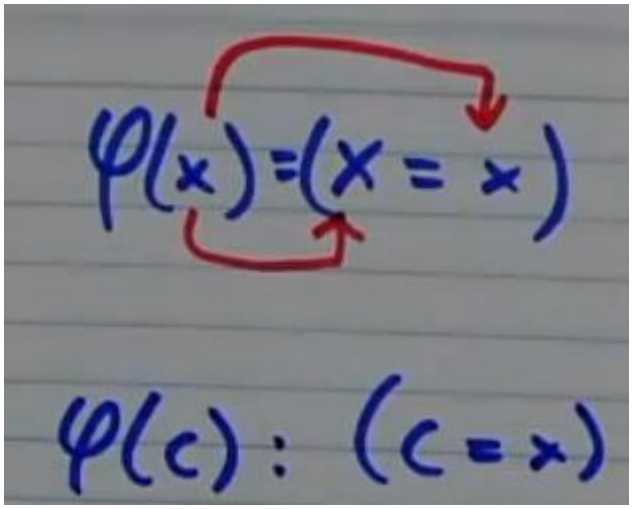
We can also use counter models to show unprovability

Lecture 9

Syntax of Predicate (First-Order) Logic

Given a vocabulary

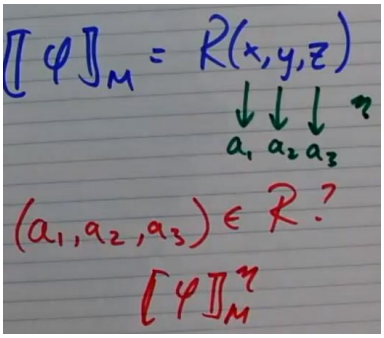
- Terms are defined recursively over a set of variables
- Formulas defined recursively as
    - Atomic formulas: built from predicates and equality, and terms
    - Other formulas: built recursively using Boolean connectives and quantifiers
- Parentheses usage relaxed to aid readability
- Free variables "captured" syntactically with PSI(x) notation



Semantics of Predicate (First-Order) Logic

Given a vocabulary

- A model interprets all the symbols of the vocabulary over some domain
    - Models give meaning to our vocabulary so for example an interface in java needs an implementation, a model can be seen any implementation of an interface.
- [[x]] M is a relation on DOM(M)
- An environment assigns variables to elements of Dom(M)

- 

- [[x]] eM "looks up" the tuple defined by e in the relation [[x]] M and returns an element of B depending on its presence
    - $[\![\varphi]\!]^{\eta}_{\mathcal{M}}$ "looks-up" the tuple defined by η in the relation $[\![\varphi]\!]_{\mathcal{M}}$ and returns an element of $\mathbb{B}$ depending on its presence

Satisfiability and Validity

A formula x is

- Satisfiable if there is some model and environment (interpretation) such that [[x]] eM = true i.e. (M, e |= x)
- True in a model M if for all environments e, M, e |= x
- A logical validity if it is true in all models
- A logical consequence of a set of formulas T (written T |= x) if M, e |= x for all interpretations which satisfy every element of T.
- Logically equivalent to a formula y if [[x]] eM = [[y]] eM for all interpretations.

Theorems

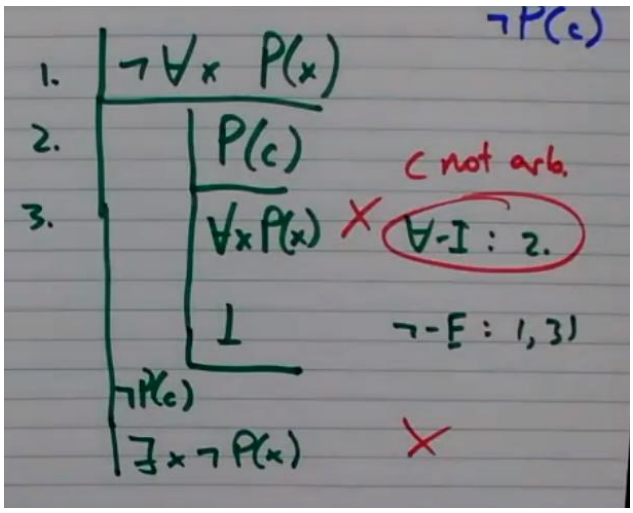
- EMPTYSET |= x if and only if x is a logical validity
- Y |= x if and only if y → x is a logical validity
- Y == x if and only if y ←→ x is a logical validity

$\dfrac{}{a = a}$ (=-I)	$\dfrac{a = b \qquad A(a)}{A(b)}$ (=-E1)	$\dfrac{a = b \qquad A(b)}{A(a)}$ (=-E1)
$\dfrac{A(c) \qquad (1,2,3)}{\forall x A(x)}$ ($\forall$-I)	$\dfrac{A(c) \qquad (2)}{\exists x A(x)}$ ($\exists$-I)	$\dfrac{\forall x A(x)}{A(c)}$ ($\forall$-E)
$\dfrac{\exists x A(x) \qquad \begin{matrix}[A(c)] \\ \vdots \\ B\end{matrix} \qquad (1,2,4)}{B}$ ($\exists$-E)		(1): c is arbitrary (2): x is not free in $A(c)$ (3): c is not free in $A(x)$ (4): c is not free in B

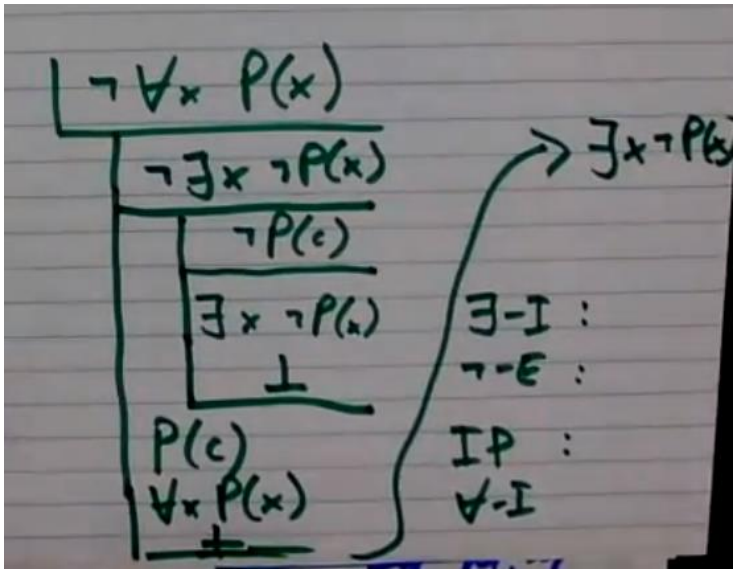
Example proof

$$\neg\forall x P(x) \quad \vdash \quad \exists x \neg P(x)$$

Wrong way to do it, since C is not arbitrary in our undischarged assumption!



Correct way



Need to know how to

- Translate requirements into propositional or predicate logic
- Syntax and semantics definitions
- How to do proofs in natural deduction

Easy problems are to verify a proof or checking satisfiability of model, the hard problems are for finding proofs or finding a satisfying model. It's a lot easier to check than generate.

Theorem provers

Proof assistants

- Interactive/directed theorem proving
    - Give it a goal and it will go out and try to prove it
- Some have automated theorem proving
- Minimal default behaviour
    - Need to provide rules and axioms
    - Can prove correctness from foundations
- Some can extract code from specifications, for example Coq.
- Proofs are only as correct as the proof assistant itself, so if the proof assistant is incorrect, we know the proofs are incorrect.
- If the proof assistant is correct at its core it can provide correct proof systems.

Examples

COQ used to prove 4-colour theorem

Isabelle proved functional correctness of seL4 microkernel

HOL (built on top of Isabelle) proved Kepler conjecture

Natural deduction prover

SMT solvers

Satisfaction Modulo Theories

- SAT solvers for fragments of predicate logic
- Theories force certain interpretations (e.g. arithmetic)
- Two main approaches
    - Convert everything into SAT (eager solvers)
    - Combination of SAT solver and Theory specific solvers (lazy solvers)

Advantages

- They tend to be very fast
- Good at handling domain specific logic (e.g. arithmetic)

Disadvantages

- Restricted to quantifier-free fragments
- Unwieldly statements
- You don't get all of predicate logics, generally can't do for all or there exists
    - Trade off for being fast, it only covers smaller pieces

Examples

Z3

- Wide range of built in theories
- Backend for verification tools such as Dafny (used in SENG2011)

OpenSMT

CVC4

Knowledge Based Systems

Build up “knowledge” from base facts and inference rules

Advantages

- Works with unrestricted predicate logic
- Simpler statements

Disadvantages

- We need more fine-tuned constraints
- Slower

Hoare Logic

Created by Sir Tony Hoare

- Pioneer of formal verification
- Invented quicksort
- Invented the null reference
- Invented CSP (formal specification language)
- Invented Hoare Logic

It is a procedure for reasoning about programs

L a simple imperative programming language

Consider the vocabulary of this basic arithmetic

- Constant symbols: $0, 1, 2, \ldots$
- Function symbols: $+, *, \ldots$
- Predicate symbols: $<, \leq, \geq, |, \ldots$
- An **(arithmetic) expression** is a term over this vocabulary.
- A **boolean expression** is a predicate formula over this vocabulary.

See examples below to understand last 2 dot points

The language L is a simple imperative programming language made up of four statements

Assignment:

X := e (where x is a variable and e is an arithmetic expression

Sequencing:

P; Q

Combines two smaller statements into one bigger statement

Conditional:

if b then P else Q fi

Where b is a Boolean expression

While:

While b do P od

Where b is a Boolean expression

Example program in L factorial

```
F := 1;
K := 0;
While k < n do
        K := k + 1;
        F := F * k
Od
```

In this program n is a free variable which is number factorial we are working out.

Hoare triples (SYNTAX)

$$\{\varphi\}\ P\ \{\psi\}$$

This statement basically has, pre-condition to the left in PSI and post condition in trident to the right

Intuition:

- PSI
  - The pre-condition which is an assertion about the state prior to the execution of the code fragment
- P
  - The code fragment in our model and machine
- Trident
  - The post-condition which is an assertion about the state after the execution of the code fragments IF IT TERMINATES

Example

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

$$\{(x = 0)\}\ x := 1\ \{(x = 500)\}$$

$$\{(x > 0)\}\ y := 0 - x\ \{(y < 0) \wedge (x \neq y)\}$$

Example with factorial in L

Example

$$\{n \geq 0\}$$
$$f := 1;$$
$$k := 0;$$
$$\textbf{while } k < n \textbf{ do}$$
$$\quad k := k + 1;$$
$$\quad f := f * k$$
$$\textbf{od}$$
$$\{f = n!\}$$

We can provide the code fragments, pre and post conditions as proof that our program performs its behaviour.

Hoare Logic

We know what we want informally, we want to establish when a Hoare triple is valid

To do this we develop a semantics or we can derive the triple in a syntactic manner (proof)

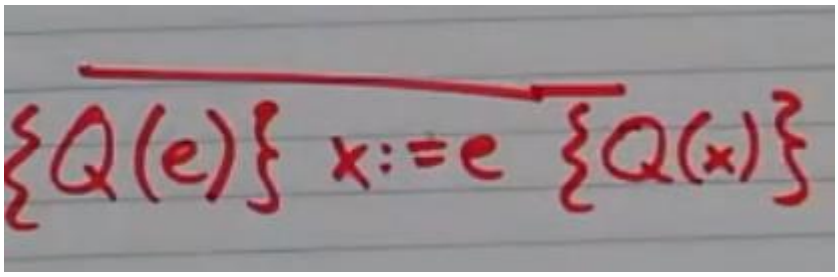
Hoare Logic consists of one axiom and four inference rules for deriving Hoare triples. And we can use this to prove properties of programs.

Inference rules

Assignment axiom

$$\frac{}{\{\varphi[e/x]\}\, x := e\, \{\varphi\}} \quad (\text{ass})$$

If x has property Q after executing the assignment, then e must have property Q before executing the assignment.



Some examples

Example

$$\{(y = 0)\}\, x := y\, \{(x = 0)\}$$

$$\{(y = y)\}\, x := y\, \{(x = y)\}$$

$$\{(1 < 2)\}\, x := 1\, \{(x < 2)\}$$

$$\{(y = 3)\}\, x := y\, \{(x > 2)\} \qquad \textit{Problem!}$$

Sequence axiom

$$\frac{\{\varphi\}\ P\ \{\psi\} \qquad \{\psi\}\ Q\ \{\rho\}}{\{\varphi\}\ P; Q\ \{\rho\}}\ (\text{seq})$$

If the post condition of P matches the pre-condition of Q, we can sequentially combine the two program fragments! So, if x = 3 is our post condition and our pre condition is x = 3, we can essentially combine the two together and sequence it.

Example

Example

$$\frac{\{(0=0)\}\ x := 0\ \{(x=0)\} \qquad \{(x=0)\}\ y := 0\ \{(x=y)\}}{\{(0=0)\}\ x := 0; y := 0\ \{(x=y)\}}\ (\text{seq})$$

Conditional axiom

$$\frac{\{\varphi \wedge g\}\ P\ \{\psi\} \qquad \{\varphi \wedge \neg g\}\ Q\ \{\psi\}}{\{\varphi\}\ \textbf{if } g \textbf{ then } P \textbf{ else } Q \textbf{ fi}\ \{\psi\}}\ (\text{if})$$

- When a conditional is executed either P or Q will be executed
- If trident is a post condition of the conditional then it must be a post condition of both branches
- Likewise, if PSI is a precondition of the conditional then it must be a precondition of both branches
- Which branch gets executed depends on g, so we can assume g to be a precondition of P and ~g to be a precondition of Q (strengthen the preconditions).

While axiom

$$\frac{\{\varphi \wedge g\}\, P\, \{\varphi\}}{\{\varphi\}\ \textbf{while}\ g\ \textbf{do}\ P\ \textbf{od}\ \{\varphi \wedge \neg g\}}\ (\text{loop})$$

- PSI is a loop invariant, it must be both a pre and post condition of P so that sequences of Ps can be run together.
- A loop invariant holds at every iteration of our loop.
- G is our loop condition.
- If the while loop terminates, then g cannot hold.

Precondition strengthening and Postcondition weakening

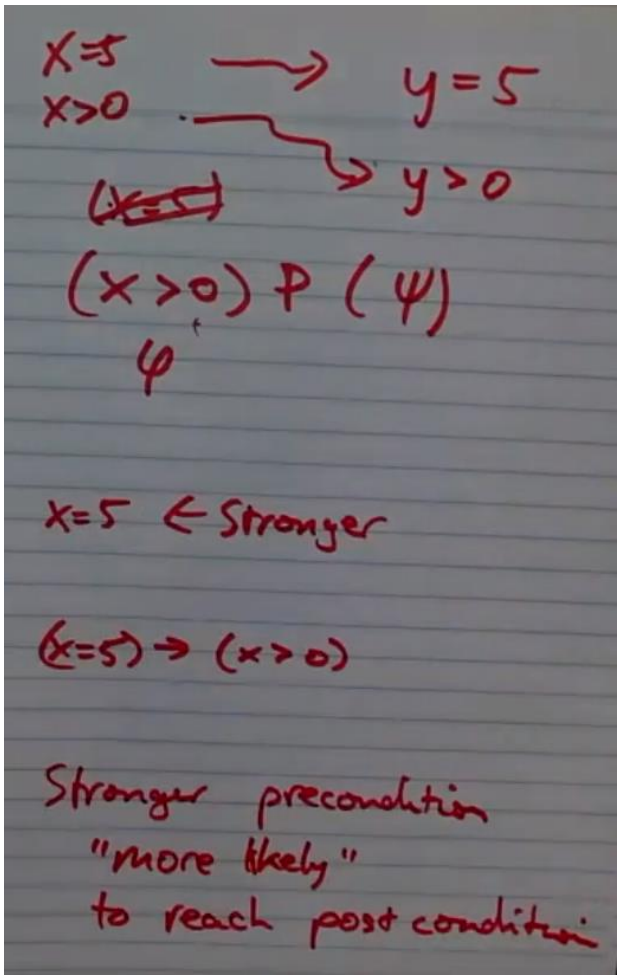
$$\frac{\varphi' \Rightarrow \varphi \qquad \{\varphi\}\, P\, \{\psi\} \qquad \psi \rightarrow \psi'}{\{\varphi'\}\, P\, \{\psi'\}}\ (\text{cons})$$

- adding assertions to the precondition makes it more likely the post condition will be reached
- removing assertions to the postconditions make it more likely the postcondition will be reached
- if you can reach the postcondition initially, then you can reach it in the more likely scenario.

We can either strengthen pre-conditions or weaken post conditions.

For example, let's say we have instead our precondition (x > 0) we have precondition (x = 5). X = 5 is a stronger precondition as (x = 5) → (x > 0) this works one way, because we know 5 > 0, but if we had it the other way around x > 0 does not imply x = 5, x can be 1, 2, 3 or any number greater than 0.

For strengthening pre-condition



Likewise, if we weaken our post-conditions it is more likely to reach post conditions.

Lecture 10

Hoare logic consists of one axiom and four rules to derive Hoare triples.

$$\frac{}{\{\varphi(e)\}\, x := e\, \{\varphi(x)\}} \quad \text{(ass)}$$

Precondition strengthening and Postcondition weakening

$$\frac{\varphi' \to \varphi \qquad \{\varphi\}\, P\, \{\psi\} \qquad \psi \to \psi'}{\{\varphi'\}\, P\, \{\psi'\}} \quad \text{(cons)}$$

Intuition:

- $\varphi' \to \varphi$: φ' is **stronger** than φ
  - Stronger conditions impose more restrictions
  - $\Rightarrow$ States which satisfy φ' are a subset of states which satisfy φ
  - $\Rightarrow$ States reached after executing P are a subset
  - $\Rightarrow$ The postcondition will hold in the smaller set of terminal states
- $\psi \to \psi'$: ψ' is **weaker** than ψ
  - Weaker conditions impose fewer restrictions
  - $\Rightarrow$ States which satisfy ψ are a subset of states which satisfy ψ'
  - $\Rightarrow$ States reached after executing P are a subset of those which satisfy ψ'

Full example of Hoare Logic factorial

Example

$$
\begin{array}{l}
f := 1; \\
k := 0; \\
\textbf{while } \neg(k = n) \textbf{ do} \\
\quad k := k + 1; \\
\quad f := f * k \\
\textbf{od}
\end{array}
$$

We know one of our conditions for exiting the loop is that k = n, it is one of our post conditions. We want to prove that this can be satisfied with the following

Example

$$
\begin{array}{l}
\{\textsc{True}\} \\
f := 1; \\
k := 0; \\
\textbf{while } \neg(k = n) \textbf{ do} \\
\quad k := k + 1; \\
\quad f := f * k \\
\textbf{od} \\
\{f = n!\}
\end{array}
$$

So, from every state {TRUE} we know that after executing the code {f = n!}.

Example

1.	$\{1 = 0!\}\, f := 1\, \{f = 0!\}$	(ass)
2.	$\{f = 0!\}\, k := 0\, \{f = k!\}$	(ass)
3.	$\{1 = 0!\}\, f := 1; k := 0\, \{f = k!\}$	(seq) : 1, 2
4.	$\{f(k+1) = (k+1)!\}\, k := k+1\, \{fk = k!\}$	(ass)
5.	$\{fk = k!\}\, f := f * k\, \{f = k!\}$	(ass)
6.	$\{f(k+1) = (k+1)!\}\, \textsc{Loop}\, \{f = k!\}$	(seq) : 4, 5
7.	$(f = k!) \wedge \neg(k = n) \rightarrow f(k+1) = (k+1)!$	math
8.	$\{(f = k!) \wedge \neg(k = n)\}\, \textsc{Loop}\, \{f = k!\}$	(cons): 6,7
9.	$\{f = k!\}$ **while** ... **od** $\{(f = k!) \wedge (k = n)\}$	(loop): 8
10.	$\{1 = 0!\}\, \textsc{Factorial}\, \{(f = k!) \wedge (k = n)\}$	(seq)
11.	$\textsc{True} \rightarrow (1 = 0!)$	math
12.	$((f = k!) \wedge (k = n)) \rightarrow f = n!$	math
13.	$\{\textsc{True}\}\, \textsc{Factorial}\, \{f = n!\}$	(cons): 10,11,12

OR WE CAN WRITE IT BESIDE OUR CODE

Example

$$
\begin{array}{lr}
 & \{\text{TRUE}\} \\
 & \{1 = 0!\} \\
f := 1; & \{f = 0!\} \\
k := 0; & \{f = k!\} \\
\textbf{while } \neg(k = n) \textbf{ do} & \{(f = k!) \wedge \neg(k = n)\} \\
 & \{f(k+1) = (k+1)!\} \\
\quad k := k + 1; & \{fk = k!\} \\
\quad f := f * k & \{f = k!\} \\
\textbf{od} & \{(f = k!) \wedge (k = n)\}
\end{array}
$$

Semantics of Hoare Logic

If R and S are binary relations, then the **relational composition** of R and S, $R;S$ is the relation:

$$R;S := \{(a,c) \ : \ \exists b \text{ such that } (a,b) \in R \text{ and } (b,c) \in S\}$$

If $R \subseteq A \times B$ is a relation, and $X \subseteq A$, then the **image of** X **under** R, $R(X)$ is the subset of B defined as:

$$R(X) := \{b \in B \ : \ \exists a \ in X \text{ such that } (a,b) \in R\}.$$

Informal semantics

Hoare logic gives a proof of {x} P {y}, that is |- {x} P {y} (proof from no assumptions) this is an axiomatic semantic.

But how do we determine {x} P {y} is **valid**, that is |= {x} P {y}?

If x holds in a state of some computational model, then y holds in the state reached after a successful execution of P (pre and post conditions).

To do this we need to define a program

A program is a relation between system states to system states. The guard for our program states is also decided during runtime, it has an element of non-determinism. We don't know from runtime which state our program will end up in, our program simply relates initial states to final states. There are two views on what a state is

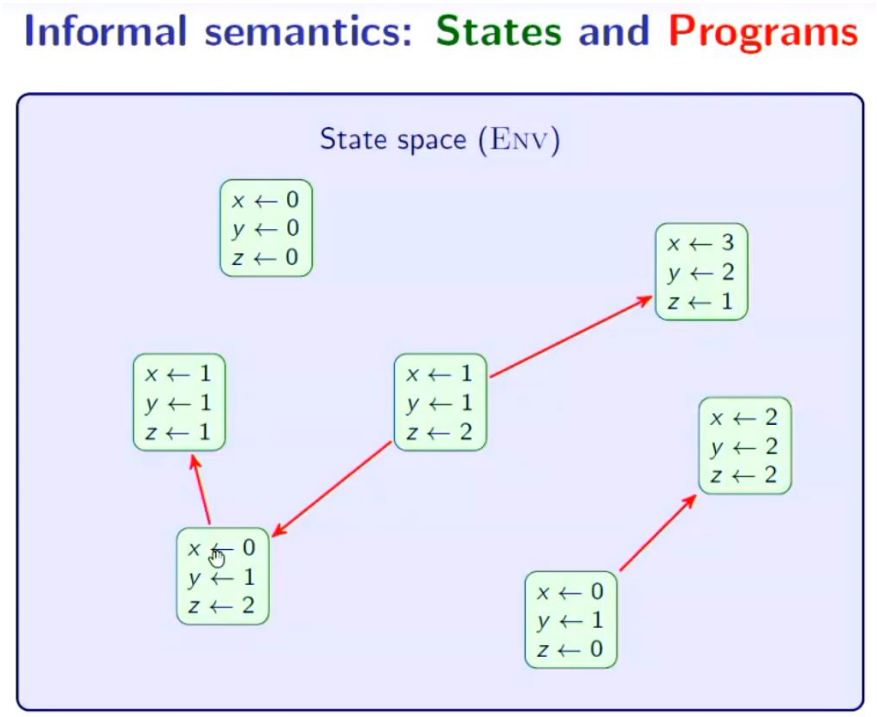
Concrete view

From a physical perspective

- states are memory configurations, register contents, etc. (where is mouse pointing at)
- store of variables and the values associated with them

Abstract view

- the pre/post condition predicates hold in a state
- states are logical interpretations (Model and environment) an interpretation
- There is only one model of interest: standard interpretations of arithmetical symbols.
- States are fully determined by environments
- States are functions that map variables to values.



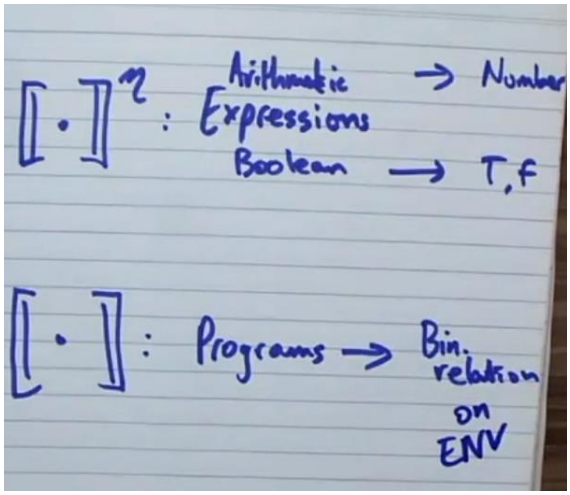
The arrows are the jump between states based on program execution.

An environment or state is a function from variables to numeric values. We denote by ENV the set of all environments.

NB

An environment, η, assigns a numeric value $[\![e]\!]^\eta$ to all expressions e, and a boolean value $[\![b]\!]^\eta$ to all boolean expressions b.

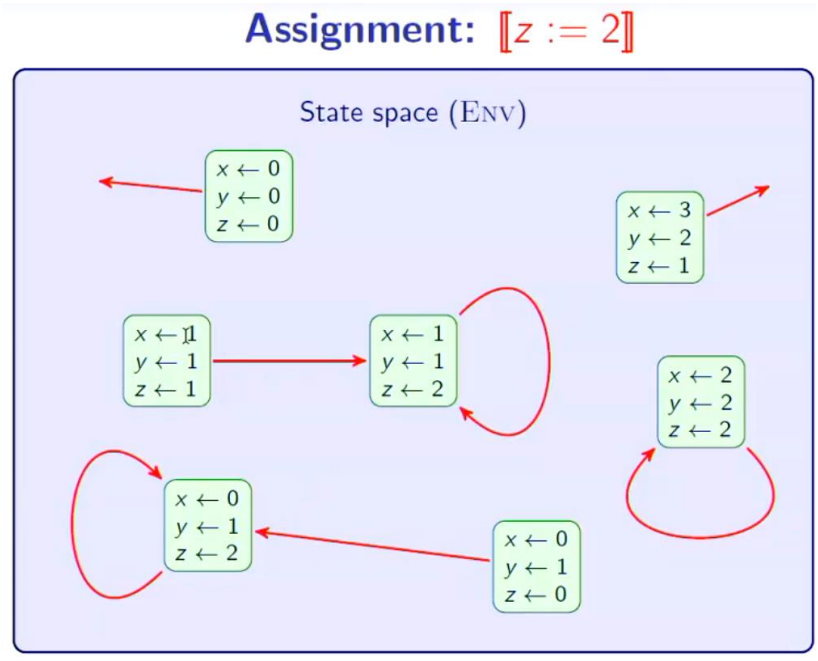
Given a program P of L, we define [[P]] to be a binary relation on the set of all environments ENV in a recursive manner. A program is defined recursively.



Now we will define our rules inductively for each of our program cases (assignment, loop, sequence etc.).

Assignment

$$(\eta, \eta') \in [\![x := e]\!] \quad \text{if, and only if} \quad \eta' = \eta[x \mapsto [\![e]\!]^\eta]$$



Basically, if there is another state where the value, we are checking matches then we take those states. Essentially, we send the current state to the next available state that has all other assignments the same, and the new assignment mapped as seen in the picture above, so the arrows are looking for matching x, y before it assigns z which is our assignment, since our previous values must be maintained. This is our base case.

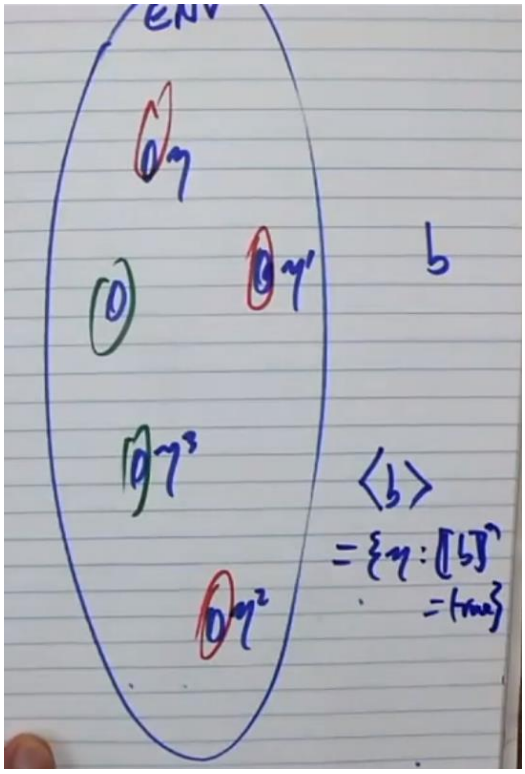
Sequencing

[[P; Q]] = [[P]]; [[Q]] note; is a relational composition outside on the RHS whereas inside on LHS it represents program composition. For sequencing we don't have to talk about environments, its built solely on the structure of P.

Conditional

$$[\![\textbf{if } b \textbf{ then } P \textbf{ else } Q \textbf{ fi}]\!] = \begin{cases} [\![P]\!] & \text{if } [\![b]\!]^{\eta} = \texttt{true} \\ [\![Q]\!] & \text{otherwise.} \end{cases}$$

Predicates can be seen as programs. A Boolean expression b defines a subset of ENV



We can think of the read states as where our Boolean expression b holds, and the green states where it doesn't. As can be seen, b produces a subset of all our possible states where it is true (in this case the red states).

A boolean expression b defines a subset (or unary relation) of ENV:

$$\langle b \rangle = \{\eta \ : \ [\![b]\!]^\eta = \texttt{true}\}$$

This can be extended to a binary relation (i.e. a program):

$$[\![b]\!] = \{(\eta, \eta) \ : \ \eta \in \langle b \rangle\}$$

We can extend our red unary relations by relating them to themselves. (n, n). So, in turn, b corresponds to the program

If b then skip else BOTTOM fi

Using this we can refine out conditional semantics.

[[if b then P else Q fi]] = [[b; P]] U [[~b; Q]]

While

While b do P od

Do 0 or more executions of P while b holds, terminate when b does not hold.

This can be seen as, do 0 or more executions of (b; P), terminate with execution of ~b. but now we need to define how to do "0 or more executions" of (b; P). to do this we will use transitive closure.

$$[\![\textbf{while } b \textbf{ do } P \textbf{ od}]\!] = [\![b; P]\!]^*; [\![\neg b]\!]$$

Transitive Closure

Given a binary relation $R \subseteq E \times E$, the *transitive closure of* R, R^* is defined to be the limit of the sequence

$$R^0 \cup R^1 \cup R^2 \cdots$$

where

- $R^0 = \Delta$, the diagonal relation
- $R^{n+1} = R^n; R$

NB

- R^* *is the smallest transitive relation which contains* R
- *Related to the Kleene star operation seen in languages:* Σ^*

Technically R* is the least-fixed point of f(X) = X U X; R

Validity

A Hoare triple is valid written |= {x} P {y} if

[[P]] (<x>) $\subseteq \langle\psi\rangle$.

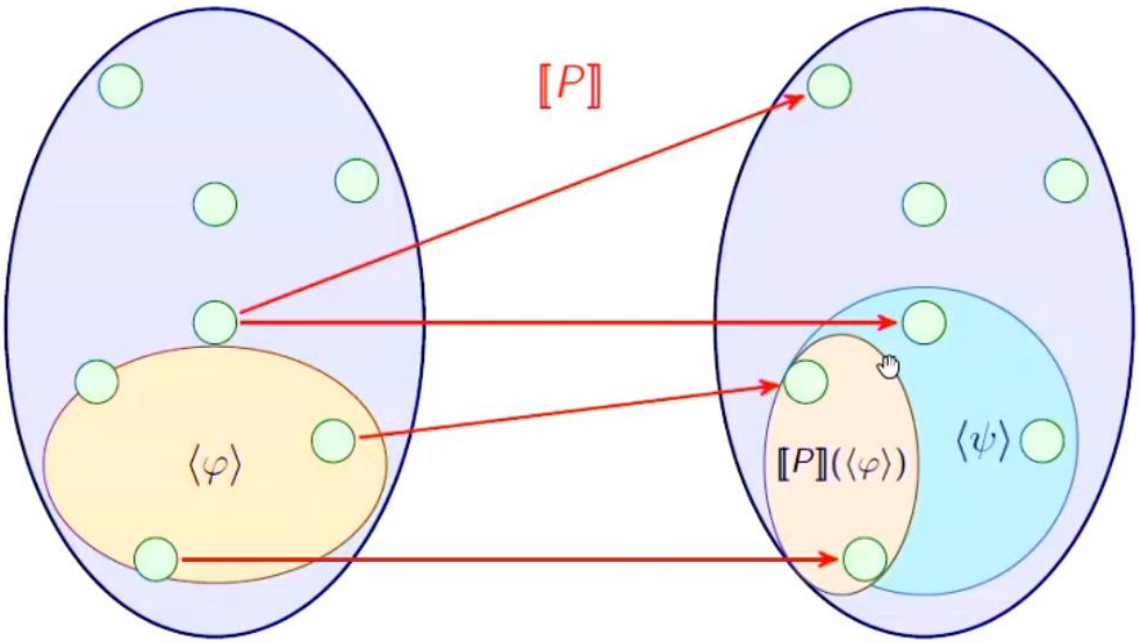
So, the relational image under P of all states where x hold, is contained in the set of states where y holds.

A Hoare triple is **valid**, written $\models \{\varphi\}\, P\, \{\psi\}$ if

$$[\![P]\!](\langle\varphi\rangle) \subseteq \langle\psi\rangle$$

That is, the relational image under $[\![P]\!]$ of the set of states where φ holds is contained in the set of states where ψ holds.

A picture form note both ovals are the environments (or states)



Lecture 11

While loop in Hoare logic can be seen as the transitive closure between the loop condition and the code inside the loop sequenced with the negation of the loop condition. The transitive closure can be seen as the union for the sequence of all iterations of the loop

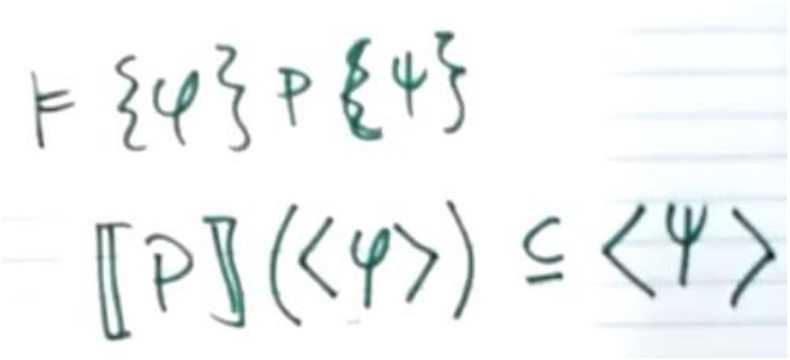
- $R^{i+1} = R^i; R$ for $i \geq 0$.

The **transitive closure**, R^* is then defined to be:

$$R^* := \bigcup_{i=0}^{\infty} R^i$$

We can see its from iteration 0 to however many there are until termination where we sequence all the steps of our loop.

To check if a Hoare triplet is valid, we need to take the relation image of the preconditions and make sure they are a subset of the post condition, in other words pre conditions imply post conditions



Hoare logic is sound when the following holds

If |- {x} P {y} then |= {x} P {y}

This is basically saying if we have a derivation of a Hoare logic then we have a valid Hoare triplet.

Try proving all three of these

Some results on relational images

Lemma

For any binary relations $R, S \subseteq X \times Y$ and subsets $A, B \subseteq X$:

(a) *If $A \subseteq B$ then $R(A) \subseteq R(B)$*

(b) $R(A) \cup S(A) = (R \cup S)(A)$

(c) $R(S(A)) = (S; R)(A)$

As a consequence of these rules we have

Corollary

If $R(A) \subseteq A$ then $R^*(A) \subseteq A$

Since R is a subset of R*(a)

Theorem

If $\vdash \{\varphi\}\, P\, \{\psi\}$ *then* $\models \{\varphi\}\, P\, \{\psi\}$

Proof:
By induction on the structure of the proof.

If we have a derivation of a Hoare triple, then that Hoare triple is valid, we can prove this by structural induction.

To do this we have to prove it for all base cases and all inductive cases

First, we will prove it on the rule of

Substitution

Base case: Assignment rule

$$\frac{}{\{\varphi[e/x]\}\, x := e\, \{\varphi\}}\ (\text{ass})$$

to prove this Hoare triple is valid we need to show that under the relational image our program that is phi is replacing x with e, we have x: =e that we have our post conditions always hold

$$[\![x := e]\!](\langle\varphi[e/x]\rangle) \subseteq \langle\varphi\rangle.$$

We can observe that this is the same as taking the post conditions in the environment where x is replaced with the evaluation of e in the environment of our pre conditions

Observation: $[\![\varphi[e/x]]\!]^{\eta} = [\![\varphi]\!]^{\eta'}$ where $\eta' = \eta[x \mapsto [\![e]\!]^{\eta}]$

So if $\eta \in \langle\varphi[e/x]\rangle$ then $\eta' \in \langle\varphi\rangle$

Recall: $(\eta, \eta'') \in [\![x := e]\!]$ if and only if $\eta'' = \eta[x \mapsto [\![e]\!]^{\eta}]$,

So $[\![x := e]\!](\eta) \in \langle\varphi\rangle$ for all $\eta \in \langle\varphi[e/x]\rangle$

So $[\![x := e]\!](\langle\varphi[e/x]\rangle) \subseteq \langle\varphi\rangle$

Now we can prove all other inductive cases

Sequence rule

Assuming that {x} P {y} and {y} Q {z} are valid, then we need to show that {x} P; Q{z} is valid.

[[P; Q]] = [[P]]; [[Q]]

So that means that [[P; Q]] (<x>) == [[Q]] ([[P]] (<x>)) (do Q on the result of P)

From our inductive hypothesis we get [[P]] (<x>) is a subset of <y> and [[Q]] (<y) is a subset of <z>

So, we get [[Q]] ([[P]] (<x>)) is a subset of [[Q]] (<y>) which is a subset of <z> which is our result!

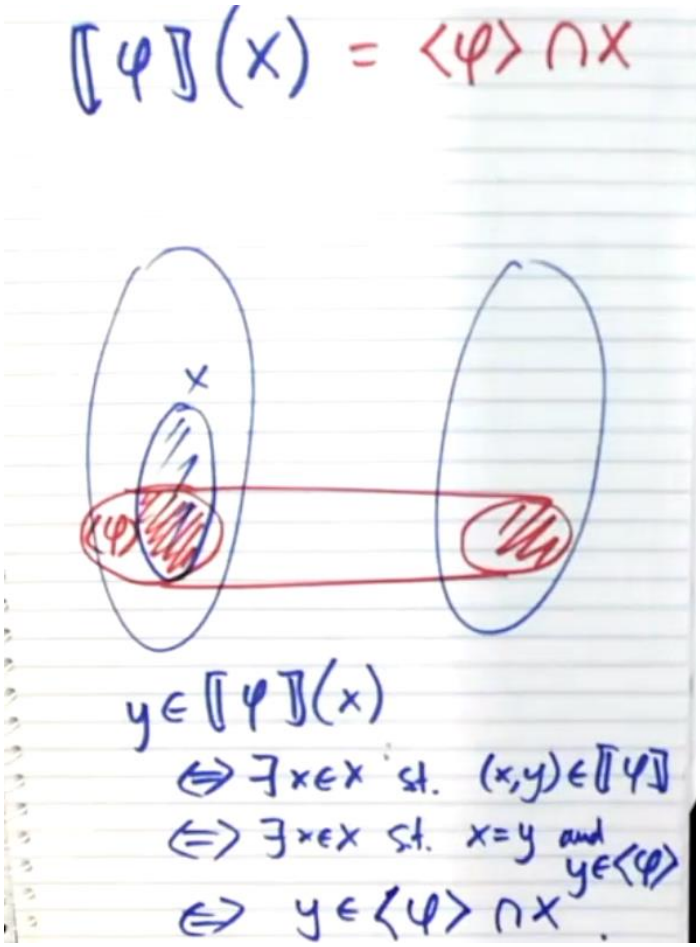
For the next two rules we need predicate logic.

For $R \subseteq \text{ENV} \times \text{ENV}$, predicates φ and ψ, and $X \subseteq \text{ENV}$:

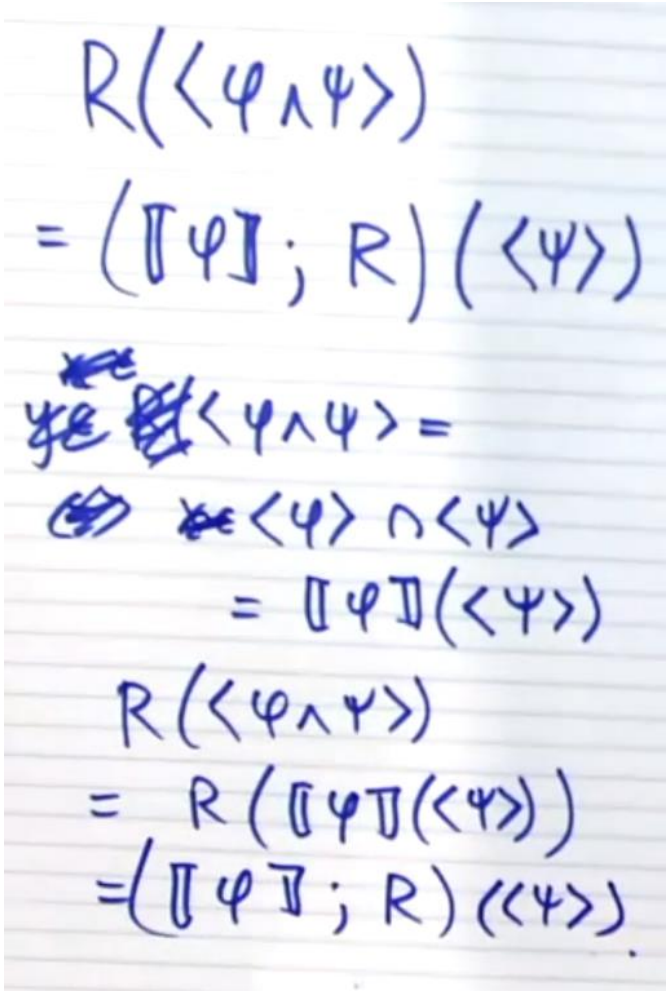
(a) $[\![\varphi]\!](X) = \langle\varphi\rangle \cap X$

(b) $R(\langle\varphi \wedge \psi\rangle) = ([\![\varphi]\!]; R)(\langle\psi\rangle)$

Proof for part 1 which says that performing the relational image of X on PHI we only perform it on the intersection of those two sets see below



For part b



Conditional rule

$$\frac{\{\varphi \wedge g\}\, P\, \{\psi\} \qquad \{\varphi \wedge \neg g\}\, Q\, \{\psi\}}{\{\varphi\}\ \textbf{if}\ g\ \textbf{then}\ P\ \textbf{else}\ Q\ \textbf{fi}\ \{\psi\}}\ \text{(if)}$$

Assume $\{\varphi \wedge g\}\, P\, \{\psi\}$ and $\{\varphi \wedge \neg g\}\, Q\, \{\psi\}$ are valid. Need to show that $\{\varphi\}\ \textbf{if}\ g\ \textbf{then}\ P\ \textbf{else}\ Q\ \textbf{fi}\ \{\psi\}$ is valid.

if g then P else Q fi is the same as [[g; P]] U [[~g; Q]]

Applying the relational image of x, we get [[g; P]] (<x>) U [[~g; Q]] (<x>)

And this gives us [[P]] (<g ^ x>) U [[Q]] (<~g ^ x>) which is a subset of <y>

While rule

Given that {x ^ g} P {y} is valid, we need to show that {x} while g do P od {y ^ ~g} is valid.

We know that [[while g do P od]] is the same thing as [[g; P]] *; [[~g]]

[[g; P]] (<x>) = [[P]](<g ^ x>) which is a subset of x, so by using our corollary for transitive closure, since the P is a subset of the transitive closure of P we also have [[g; P]]*(<x>) as a subset of X. by using sequence we can now introduce ~g so

[[g; P]] *; [[~g]] (<x>) = [[~g]] ([[g; P]] *(<x>)) which is a subset of [[~g]] (<x>)

Which is = <~g ^ x>

Consequence rule

Assume {x} P {y} is valid and X' → x and y → y'. we now need to show that {x'} P {y'} is valid.

If x' → x then x' is a subset of x (if a state satisfies x' then it will also satisfy x meaning it is a subset)

[[P]] (<x'>) is a subset of [[P]] (<x>), which is a subset of <y> which is also a subset of <y'>

Given all these proofs btw cheat sheet them for exam!

Given all these proofs we can now show that any deduction from Hoare logic using the rules of substitution, conditional, loop, consequence is a valid. All deduced Hoare triples are valid triples. So, Hoare logic is sound.

Incompleteness of Hoare Logic

Theorem Godels Incompleteness theorem

- There is no proof system that can prove every first order sentence about arithmetic over the natural numbers. There are things that hold that we cannot prove in any proof system, like how do we prove 1 + 1 = 2. Our axioms are unprovable; however, they hold, we know 1 + 1 is 2 from rules of arithmetic but we cannot prove it. This is because our model is is the natural numbers.
- Natural deduction is complete because there is no specification it is simply propositions there are no axioms.
- This is also partially the reason why the last bit was so confusing, because there are consequences that result from valid reasoning that is unprovable consequences, for example x → x' is valid, we cannot prove it, however without it we cannot complete the proof without it.
- There are valid triples which rely on a truth statement that has no proof.

Relative completeness of Hoare Logic

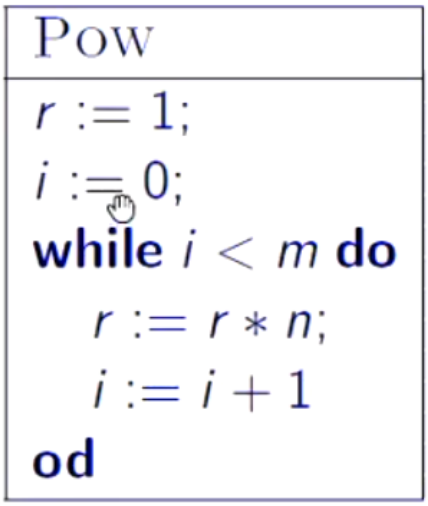
If we did for instance have some otherworldly being that can say yes this is valid and can determine validity of predicates then we have sound proof and that means all valid proofs will have a deduction i.e.

If |= {x} P {y} then |- {x} P {y}

Lecture 12

This stuff won't be deeply assessed it is only a taste of what can come in future courses!

Weakest precondition reasoning



We would like to show $\{\varphi\}\ \textsc{Pow}\ \{r = n^m\}$.

- What should φ be?
- What should the intermediate assertions be?

Well we can have the pre-condition that m ~=0 and n ~= 0 to make sure we get the right values, if n is 0, we can have some issues maybe we a change it just to n ~= 0

What we want out of Hoare logic is that

Given a post condition, we want to find the weakest possible pre-condition to satisfy the post condition after pogrom execution as this gives the most general solution that can apply to more cases.

Given a program P and a post condition y, the weakest precondition of P with respect to y, wp (P, y) is a predicate x such that

{x} P {y} AND if {x'} P {y} then x' → x

So, x is the precondition that satisfies y and if any other precondition satisfies the post condition then that pre condition implies x, i.e. it is a subset of x. Note this is a partial function so it doesn't have to exist.

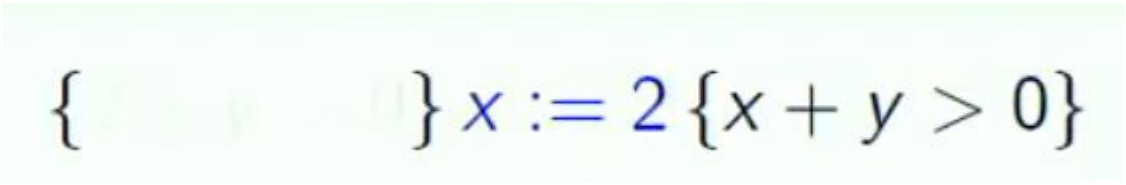
Assignment

We can attempt to compute wp based on the structure of P. for example

$$wp(x := e, \psi) = \psi[e/x]$$

This is very simple however. It is an axiom for substitution

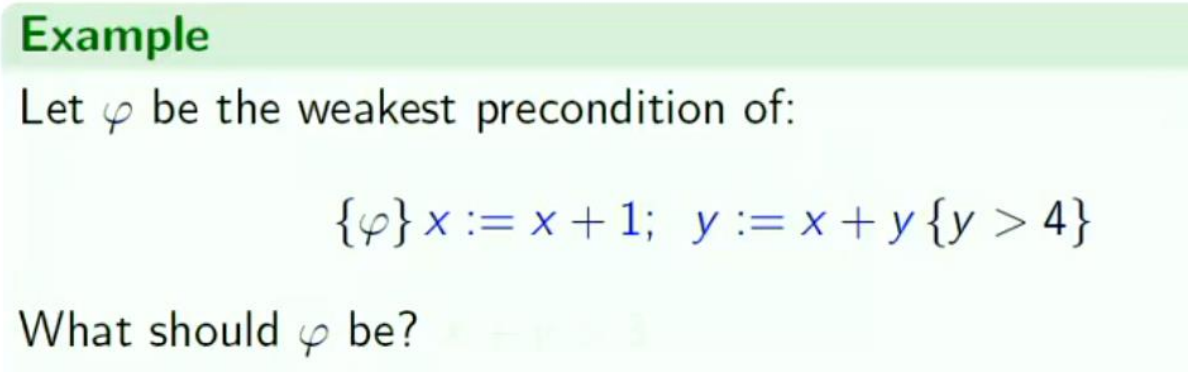
However, if we have



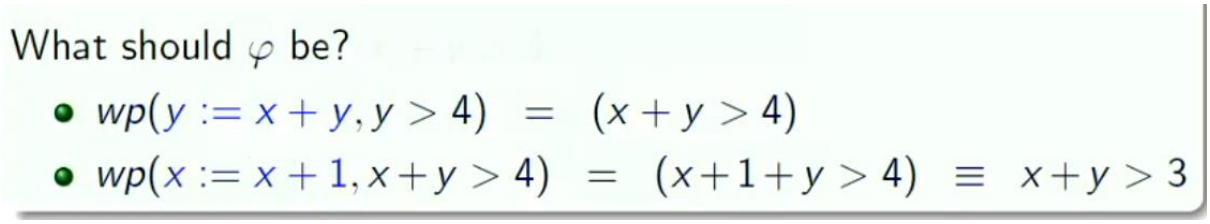
Then our most general precondition would be 2 + y > 0 since 2 would satisfy ALL solutions. This is very easy for assignment.

Sequence

For a sequence wp (P; S; y) = wp (P, wp (s, y))



So first we find the weakest precondition of the final piece of our sequence so x + y > 4 would be the precondition of wp (s, y), and given that to find the overall weakest precondition x + 1 + y > 4 would be the weakest precondition, i.e. x + y > 3



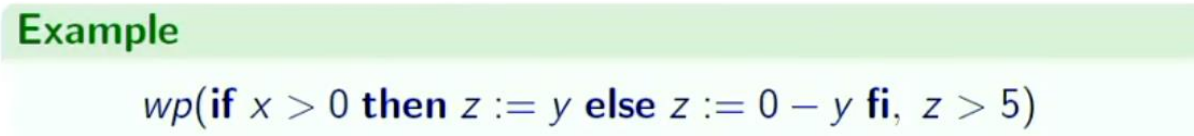
Im smort lol.

Conditional

Wp (if b then P else Q fi, y) = (b → wp (P, y)) && (~b → wp (Q, y))

This is also equivalent to

(b && wp (P, y)) || (~b && wp (Q, y))



(x > 0, y > 5) || (z =< 0, y < -5)

Loops

Wp (while b do P od, y) =???

Road block. While loops execute an undeterminable amount of times, we can't always tell exactly how many executions for our loop.

Wp calculates a triple for a single program statement block, but loops consist of a block executed repeatedly. Weakest precondition for 1 loop may be different weakest precondition for 100 loops.

So instead we will look for a loop invariant **I** such that

If we have {x} while b do P od {y}

- X → **I** loop invariant holds before execution
- **{**I^b} P{**I}** loop invariant is maintained throughout loop
- **I** ^ ~b → y conclusion

Finding good loop invariants is a research level giga hard problem. For simple programs easy, for hard programs very hard, this is a non-deterministic problem. This is hard because post conditions have to be specific, but we want pre conditions to be very general, so we need to find the balance between being too specific and too general.

Back to the example

Pow

$\{\text{init: } (m \geq 0) \wedge (n > 0)\}$

$r := 1;$
$i := 0;$

while $i < m$ **do**

$r := r * n;$
$i := i + 1$
od

$\{r = n^m\}$

What would be a good invariant?

Inv:

A good loop invariant for this problem would be that r = n^I but we need to add extra to force this to give r = n ^m as in current state when we exit loop we have I >= m so we end up with r = n^I where the I >= m so we can add I <= m to narrow our invariant to

Invariant = R = n^I && I <= m however we can do better

By passing in the pre-condition as well

Invariant → r = n^I && I <= m and m >= 0 && n > 0

We need to make sure this invariant holds on each execution of the loop.

A good heuristic for loop invariants is to look at our post condition and rephrase it in terms of the loop variables.

Pow

$\{\text{init: } (m \geq 0) \wedge (n > 0)\}$
$\{(1 = n^0) \wedge (0 \leq m) \wedge \text{init}\}$
$r := 1;$ $\{(r = n^0) \wedge (0 \leq m) \wedge \text{init}\}$
$i := 0;$

$\{\text{Inv}\}$
while $i < m$ **do** $\{\text{Inv} \wedge (i < m)\}$
$\{(r * n = n^{i+1}) \wedge (i + 1 \leq m) \wedge \text{init}\}$
$r := r * n;$ $\{(r = n^{i+1}) \wedge (i + 1 \leq m) \wedge \text{init}\}$
$i := i + 1$ $\{\text{Inv}\}$
od $\{\text{Inv} \wedge (i \geq m)\}$
$\{r = n^m\}$

What would be a good invariant?

$$\text{Inv:} \quad r = n^i \ \wedge \ i \leq m \ \wedge \ \text{init}$$

However, our obligation is to prove

$\{\text{Inv} \wedge (i < m)\}$
$\{(r * n = n^{i+1}) \wedge (i + 1 \leq m) \wedge \text{init}\}$

AND

$\{\text{init: } (m \geq 0) \wedge (n > 0)\}$
$\{(1 = n^0) \wedge (0 \leq m) \wedge \text{init}\}$

To complete the prove

Handling termination

Hoare triples for partial correctness states

{x} P {y}

Asserts y holds IF P terminates and x holds

What if we wanted to make a stronger statement that y holds and P terminates?

Hoare triples for total correctness states

[x] P [y]

If x holds at a starting state and P is executed; then P will terminate and y will hold in the resulting state.

Guess what termination is also hard.

- Algorithmic limitations (halting problem)
  - No algorithm which will tell u the code will terminate
- Mathematical limitations
  - Example

COLLATZ

while $n > 1$ **do**
  if $n\%2 = 0$
  then
    $n := n/2$
  else
    $n := 3 * n + 1$
  fi
od

  - For some n it will halt, but we don't know for all n

How can we show [(m >= 0) && (n > 0)] POW [r = n^m]

We use Hoare logic for total correctness

We are only modifying our loop rule.

{x} while b do P od {y}

Partial correctness:
Find an invariant I such that:

- $\varphi \rightarrow I$ (establish)
- $[I \wedge b]\, P\, [I]$ (maintain)
- $(I \wedge \neg b) \rightarrow \psi$ (conclude)

Show termination:
Find a **variant** v such that:

- $(I \wedge b) \rightarrow v > 0$ (positivity)
- $[I \wedge b \wedge v = N]\, P\, [v < N]$ (progress)

We need to find a variant v which is greater than 0, so as we enter the loop our variant is positive, and our progress condition shows that the variant keeps decreasing each iteration till it reaches 0 or negative.

$$\frac{[\varphi \wedge g \wedge (v = N)]\ P\ [\varphi \wedge (v < N)] \qquad (\varphi \wedge g) \rightarrow (v > 0)}{[\varphi]\ \textbf{while}\ g\ \textbf{do}\ P\ \textbf{od}\ [\varphi \wedge \neg g]} \quad \text{(loop)}$$

Using the power function as an example to show it terminates

Pow	
	$\{\text{init: } (m \geq 0) \wedge (n > 0)\}$
	$\{(1 = n^0) \wedge (0 \leq m) \wedge \text{init}\}$
$r := 1;$	$\{(r = n^0) \wedge (0 \leq m) \wedge \text{init}\}$
$i := 0;$	
	$\{\text{Inv}\}$
while $i < m$ **do**	$\{\text{Inv} \wedge (i < m) \wedge (v = N)\}$
	$\{(r * n = n^{i+1}) \wedge (i + 1 \leq m) \wedge \text{init} \wedge (v \vDash N)\}$
$r := r * n;$	$\{(r = n^{i+1}) \wedge (i + 1 \leq m) \wedge \text{init} \wedge (v = N)\}$
$i := i + 1$	$\{\text{Inv} \wedge (v < N)\}$
od	$\{\text{Inv} \wedge (i \geq m)\}$
	$\{r = n^m\}$

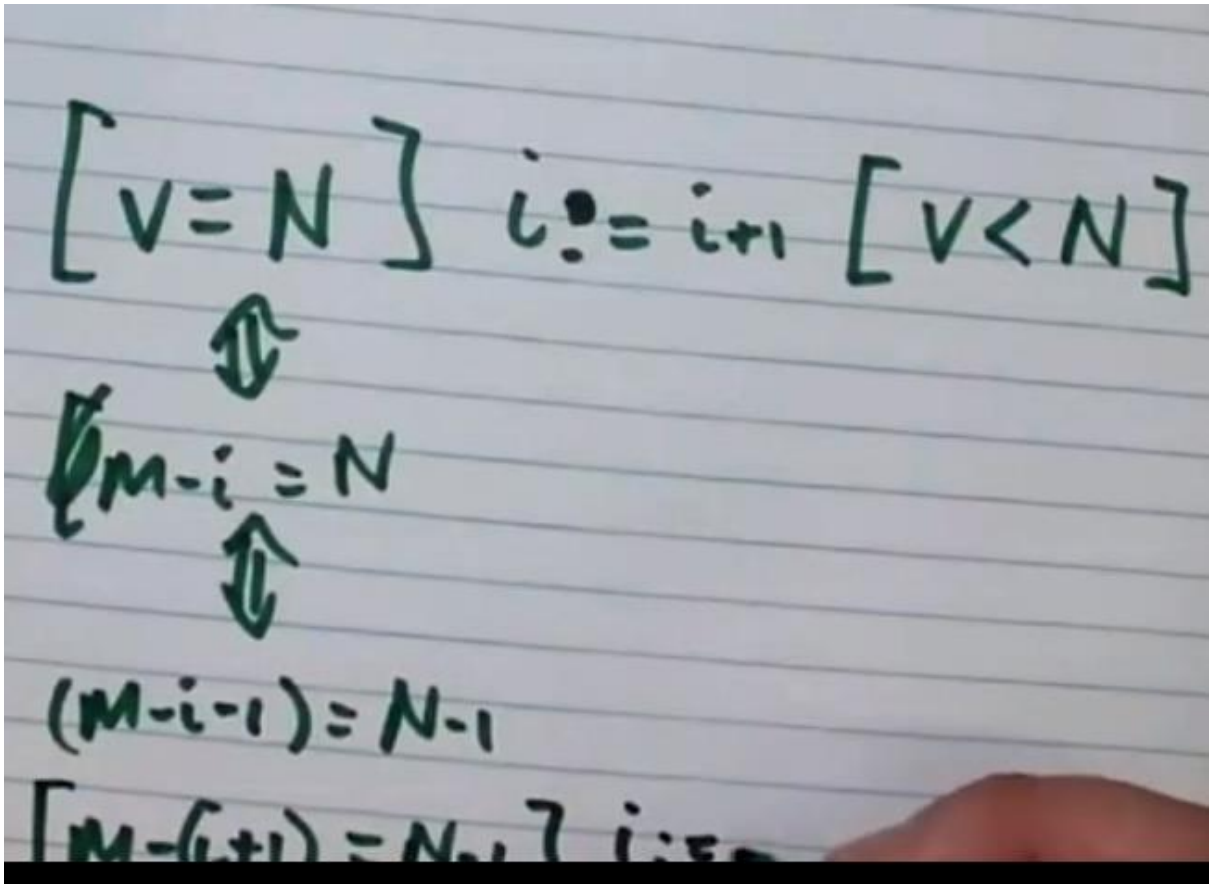
What is a suitable variant? $v := (m - i)$

A suitable variant would be the value v:= (m – I) decreasing each iteration

We have a proof obligation however to show that (m – i) > 0 before entering the loop and we also need to show on each iteration m – I decreases

- $\text{Inv} \wedge (i < m) \rightarrow (v > 0)$
- $[v = N]\ i := i + 1\ [v < N]$

This is very easy to see as v decreases when u add 1 to I on iteration



From last line [m – (I + 1) = N – 1] I:= I + 1 [m – I = N – 1] EXPAND THE NEGATIVE ON THE PRECONDITION

Implying

M – I < N by post condition weakening we can get v < N

Operational semantics

We have covered Hoare Logic in a denotational semantic:

- Programs given an abstract mathematical denotation
- Validity of Hoare triples defined in terms of this denotation

Operational semantics is an alternative approach

- Define/construct a reduction relation between programs, start and end states
- Validity defined in terms of the reduction relation.

The **Operational semantics of Hoare logic** involves defining a relation $\Downarrow \subseteq \textsc{Programs} \times \textsc{Env} \times \textsc{Env}$ recursively (on the structure of a program).

Intuitively $(P, \eta, \eta') \in \Downarrow$, written $[P, \eta] \Downarrow \eta'$, means that the program P reduces to the state η' when executed from state η.

In operational semantics we look at a program mapping states to states kind of makes more sense intuitively

Adding non-determinism

Non-determinism involves the computational model branching into one of several directions, branching

- Behaviour is unspecified: any branch can happen (decision is made at run time)
- Purely theoretical concept
- "Dual" of parallelism (one of many branches vs all of many branches): not quantum either.

Adding non-determinism is more general than deterministic behaviour, determinism is non-determinism with no branch points. In many computational models mon-determinism represents "magic" behaviour:

- Always choosing the best branch leading to faster computation e.g. P vs NP
  - P deterministic polynomial time
  - NP non deterministic polynomial time
- Error/exception handling

It is useful for abstraction, abstracted code is much easier to reason about, since we have a lot of additions which is not mathematically nice. It abstracts the language to a general setting.

Mathematically easier to deal with.

L+ a simple language with non-determinism

We relax the conditional and loop commands in L to give us non-deterministic behaviour

The programs of L+ are defined as:

- Assign
  - X := e, where x is a variable and e is an expression
- Predicate
  - PHI is a predicate
- Sequence
  - P; Q, where P and Q are programs
- Choice
  - P + Q, where P and Q are programs; intuitively, make a non-deterministic choice between P and Q
  - Cheating lmao
- Loop
  - P*, where P is a program; intuitively loop for a non-deterministic number of iterations

This abstracts the language much easier for example

$$P ::= (x := e) \mid \varphi \mid P_1; P_2 \mid P_1 + P_2 \mid P_1^*$$

L can be determined within L+

- **if** b **then** P **else** Q **fi** $= (b; P) + (\neg b; Q)$
- **while** b **do** P **od** $= (b; P)^*; \neg b$

Lecture 13

$$\frac{[\varphi \wedge g \wedge (v = N)]\, P\, [\varphi \wedge (v < N)] \qquad (\varphi \wedge g) \rightarrow (v > 0)}{[\varphi]\ \textbf{while}\ g\ \textbf{do}\ P\ \textbf{od}\ [\varphi \wedge \neg g]} \quad \text{(loop}$$

State machines

State machine model step-by-step process

- Set of states
- For each state, a set of actions defining how to transition to another state

For example, a program in L

- States
  - Functions from variables to numerical values
- Transitions
  - Program code itself each line.

Chess solver

- States
  - Board state
- Transitions
  - Legal moves

Path finding algorithms!

- States
  - Location on path heuristically defined
- Transitions
  - Moves (moving)

Example

Die Hard jug problem: Given jugs of 3L and 5L, measure out exactly 4L.

- States: Defined by amount of water in each jug
- Start state: No water in both jugs
- Transitions: Pouring water (in, out, jug-to-jug)

Try do this in code!

There is a design pattern on states check it out!

A transition system is a pair of States and transitions (S, →) where:

- S is a set of states
- → is a subset of S x S (transition) relation

If (s, s') is a transition → we write s → s'

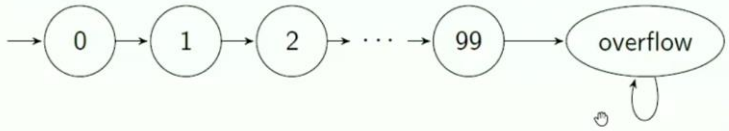
We can also add things to S, such as a designated start and final state, s0 and F respectively. We can also label transitions by elements from a set A, so for example

- → is a subset of S x A x S
- (s, a, s') is a transition → we write s -a→ s'

If our transition is a function, we say the system is deterministic otherwise it is non deterministic. In every state we have only 1 state we can transition to in a deterministic transition system. This implies in deterministic transition system there is one path we follow.

Example

A bounded counter that counts from 0 to 99 and overflows at 100:

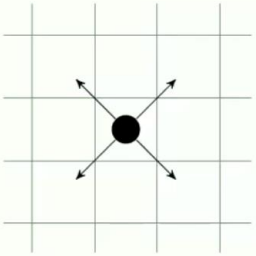


- $S = \{0, 1, \ldots, 99, \text{overflow}\}$
- $\rightarrow = \{(i, i+1) : 0 \leq i < 99\} \cup \{(99, \text{overflow})\} \cup \{(\text{overflow}, \text{overflow})\}$
- $s_0 = 0$
- Deterministic

Note the arrow going into 0 shows start state.

An example with labelling

Example



$S = \mathbb{Z} \times \mathbb{Z}$

$\Lambda = \{NW, NE, SW, SE\}$

$(x, y) \xrightarrow{NW} (x-1, y+1)$

$(x, y) \xrightarrow{NE} (x+1, y+1)$

$(x, y) \xrightarrow{SW} (x-1, y-1)$

$(x, y) \xrightarrow{SE} (x+1, y-1)$

Deterministic

Finally, a much more complicated example

Example

Given jugs of 3L and 5L, measure out exactly 4L.

- $S = \{(i, j) \in \mathbb{N} \times \mathbb{N} : 0 \leq i \leq 5 \text{ and } 0 \leq j \leq 3\}$
- $s_0 = (0, 0)$
- $\rightarrow$ given by
  - $(i, j) \rightarrow (0, j)$ [empty 5L jug]
  - $(i, j) \rightarrow (i, 0)$ [empty 3L jug]
  - $(i, j) \rightarrow (5, j)$ [fill 5L jug]
  - $(i, j) \rightarrow (i, 3)$ [fill 3L jug]
  - $(i, j) \rightarrow (i + j, 0)$ if $i + j \leq 5$ [empty 3L jug into 5L jug]
  - $(i, j) \rightarrow (0, i + j)$ if $i + j \leq 3$ [empty 5L jug into 3L jug]
  - $(i, j) \rightarrow (5, j - 5 + i))$ if $i + j \geq 5$ [fill 5L jug from 3L jug]
  - $(i, j) \rightarrow (i - 3 + j, 3)$ if $i + j \geq 3$ [fill 3L jug from 5L jug]

A run from s is a (possibly infinite) sequence s1, s2, … F such that s = s1 and si → si+1 for all I >= 1.

So, we follow a path a run can be seen as path from start to end state

We say s' is reachable from s when s →* s', if (s, s') is in the transitive closure of →. If there is a path from s to s'.

Safety and Liveness

A common problem is safety, will a transition system always avoid a particular state or states? For example, a state where a plane crashes in a plane flight is a state we want to permanently avoid. We want to guarantee we avoid a particular set of states.

The dual of safety is liveness, will a transition system always reach a particular state or set of states? For example, a state where program succeeds in a program. We want to guarantee we reach a particular set of states.

Reachability for the jug problem

Given jugs of 3l and 5l measure out exactly 4l

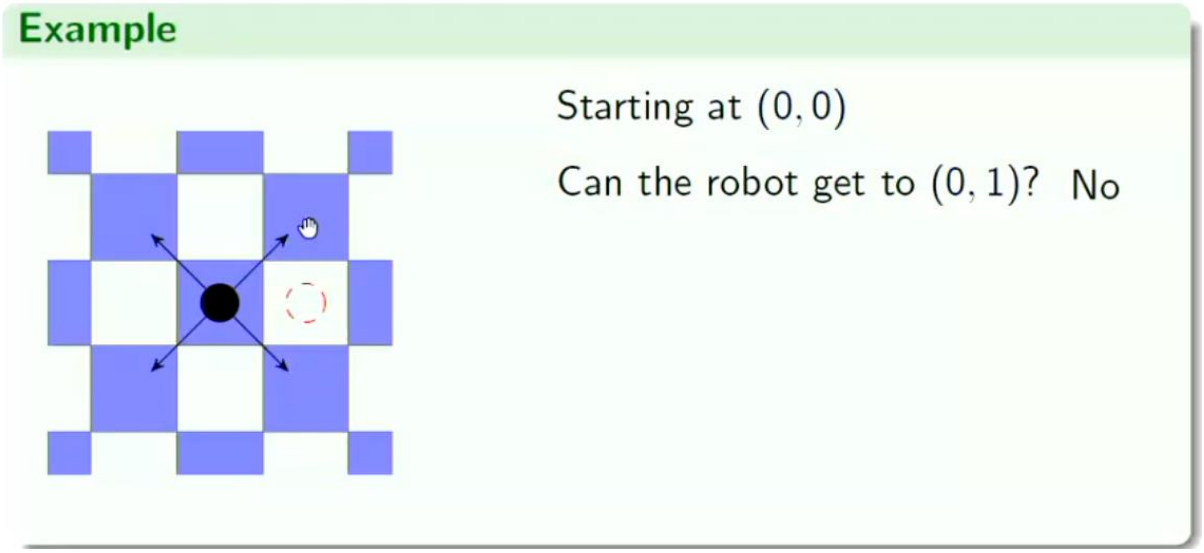
Is (4, 0) reachable from (0, 0) using our transition rules in the screenshot above.

Yes:

(0, 0) → (0, 3) → (3, 0) → (3, 3) → (5, 1) → (0, 1) → (1, 0) → (1, 3) → (4, 0)

This is one of many possible runs to solve this problem.

Safety for robot diagonally moving



Since we can only move diagonally, we always avoid a set of states particularly every 2nd right and left.

If x + y is even we get a valid tile this is a way to formalise valid diagonal movements.

Isblue ((m, n)): = 2 | (m + n)

If isBlue(s) and s → s'

Then isBlue(s')

All transitions from blue tile imply the tile we move to is blue. This can be seen as an invariant. isBlue(s) is an invariant of our system.

IsBlue ((0,0)) and ~isBlue ((0, 1))

The Invariant Principle

A preserved invariant of a transition system is a unary predicate x on states such that if x(s) holds and s → s' then x(s') holds.

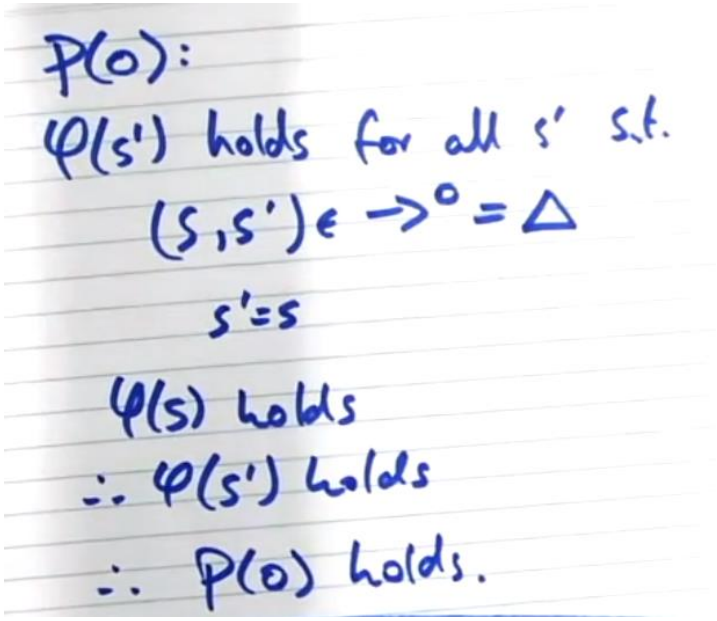
If a preserved invariant holds at a state s then it holds for all states reachable from s. This can be shown with a proof by induction.

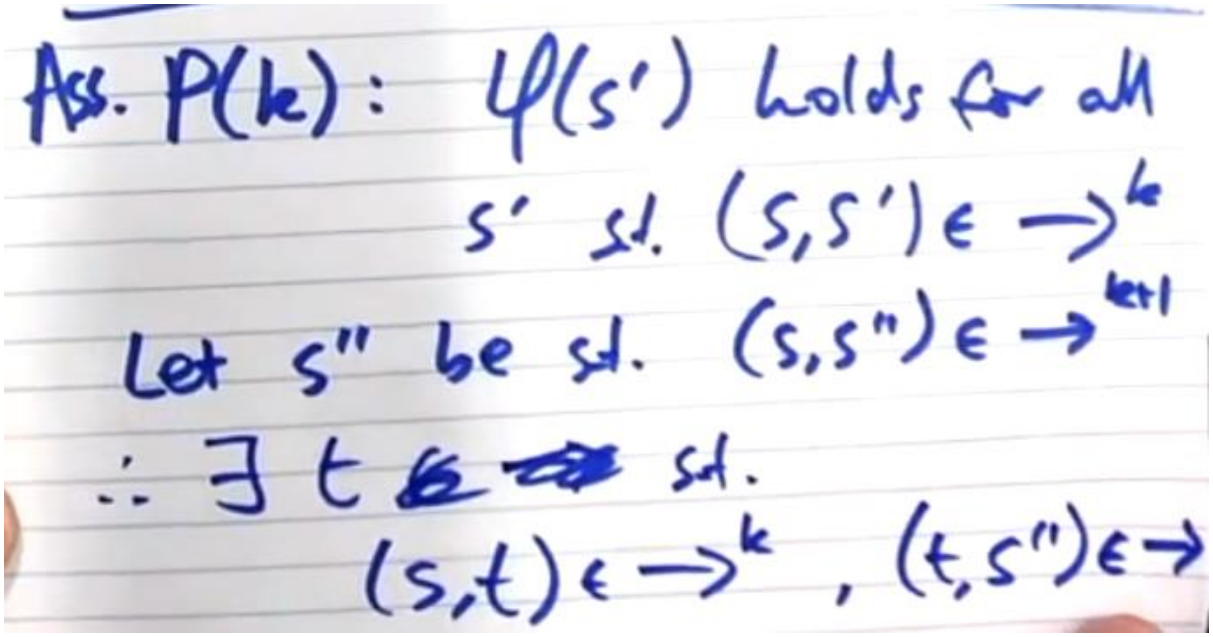
X(s), s → s' then x(s')

Base case: x(s) and x(s')

IH: for all s' such that (s, s') exist in the transitive closure of our transitions there exists a transition I such that (s, s') →I.

Inductive step: Let P(n) be the proposition that x(s') holds for all s' such that (s, s') → n





Last bit of image says we go from s to t in k iterations, and the next one says we go from t s+1 state in a single iteration (k → k + 1 is a single iteration).

Looking at a modified jug problem

Given a jug of 3L and 6L measure out exactly 4L

Is (4, 0) reachable from (0, 0)? Impossible since we can't get rid of our multiple of 3 since 6 | 3 and 3 | 3 in other words x ((I, j)) = (3 | I) && (3 | j). We can show this with an invariant. x ((I, j)) becomes our invariant. and using the principle of invariance we can show this is impossible to reach (4, 0).

Partial correctness and termination

We use invariance to show safety conditions, for example in above we showed that (4, 0) is impossible to reach. We however can't use this to show liveness. We show liveness properties with termination.

Let (s, →, s0, F) be a transition system with start state s0 and final state F and x be a unary predicate on S. we say the system is partially correct for x if x(s') holds for all final states s' that are reachable from s0. Note partial correctness does not guarantee a transition system will reach a final state.

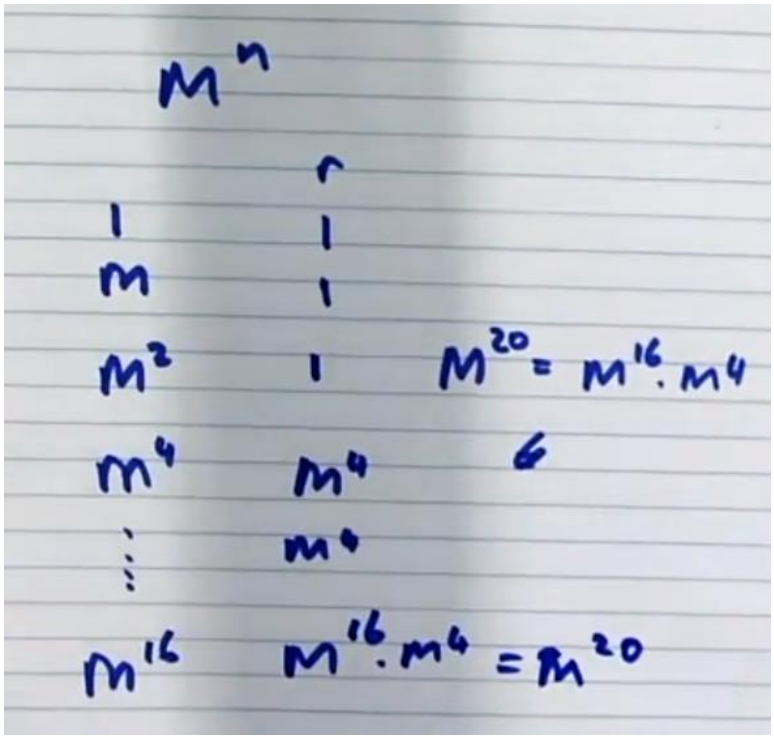
Example fast exponentiation in program L

Example

Consider the following program in $\mathcal{L}$:

```
x := m;
y := n;
r := 1;
while y > 0 do
  if 2|y then
    y := y/2
  else
    y := (y − 1)/2;
    r := r ∗ x
  fi;
  x := x ∗ x
od
```

This can be shown as this as well



It uses exponents represented in binary. To compute exponents.

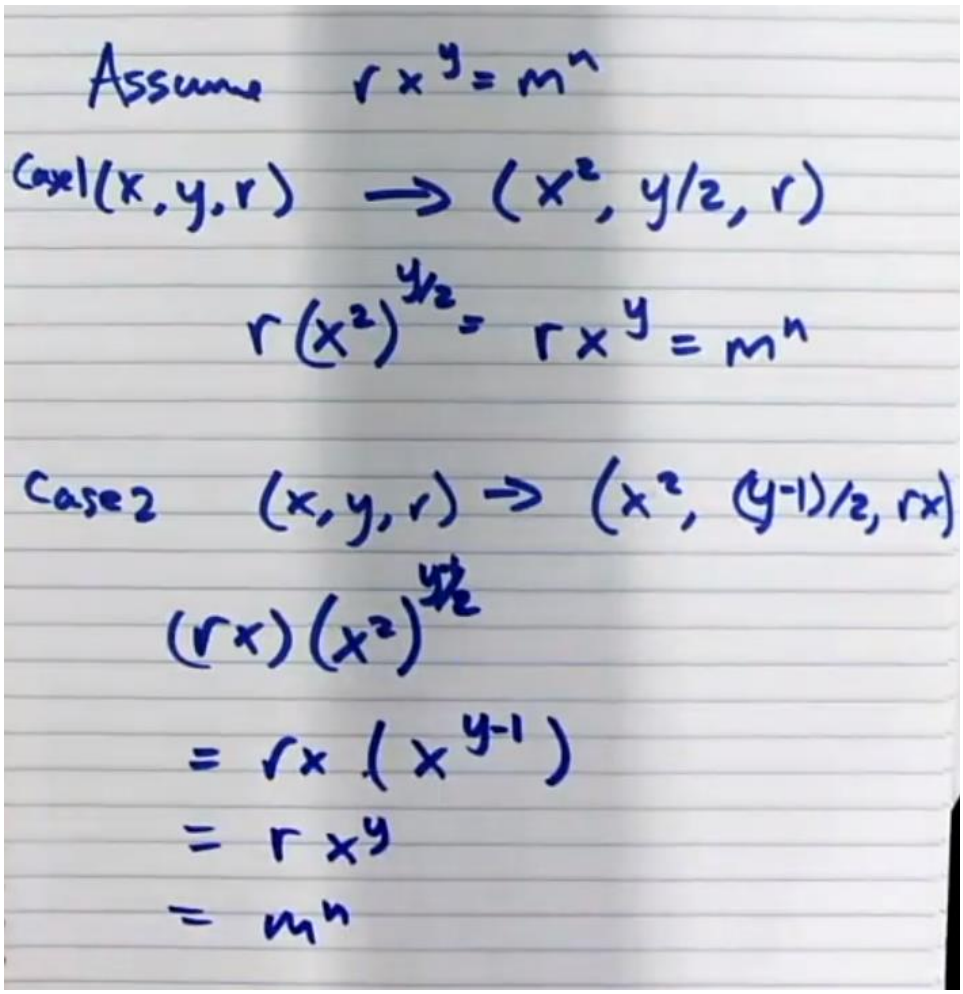
We get r = m^n.

We can show the state machine in the following

- States
  - Functions from {m, n, x, y, r} to Natural numbers
- Transitions
  - Effect of each iteration of while loop.
  - (x, y, r) → (x^2, y/2, r) if y is even
  - (x, y, r) → (x^2, (y-1)/2, rx) if y is odd
- State states: (m, n, 1)
- Final states: (x, 0, r): x and r are natural numbers)

Our goal is to show partial correctness x ((x, y, r)): = (r = m^n) since some states can never be reaches e.g. M = 0, n = 0.

Show $\psi((x, y, r)) := (rx^y = m^n)$ is a preserved invariant...



Establishing our preserved invariant, so if the program terminates then r = m^n.

How can we show total correctness? Variant.

A transition system (S, →) terminates from a state s if there is a number N such that all runs from s have length at most N, there is a bound which bound all run.

A transition system is totally correct for a unary predicate x, if it terminates from s0 and x holds in the last state of every run.

Derived variable

In a transition system (S, →) a derived variable is a function f: S → Real numbers.

A derived variable is strictly decreasing if s → s' implies f(s) < f(s').

Theorem

If f is an $\mathbb{N}$-valued, strictly decreasing derived variable, then the length of any run from s is at most $f(s)$.

In other words, f(s) gives us the bound of termination.

Example with fast exponentiation

derived variable: f ((x, y, r)) = y since y is either being halved or y-1 and halved. This guarantees the program will terminate.

Input and Output

Interaction with the environment

We can model the system interacting with an external entity via inputs (SIGMA) and outputs (GAMMA) by using labelled transitions:

$$\rightarrow \subseteq S \times \Lambda \times S \text{ where } \Lambda = \Sigma \times \Gamma$$

Input output pair

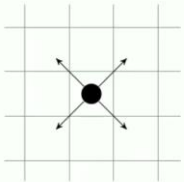
Two main categories of input/output transition systems:

Acceptors: Accept/reject a sequence of inputs (Relations).

Transducers: Take a sequence of inputs and produce a sequence of outputs (Functions).

Example

Example



$S = \mathbb{Z} \times \mathbb{Z}$

$s_0 = (0, 0)$

$(x, y) \xrightarrow{NW} (x - 1, y + 1)$

$(x, y) \xrightarrow{NE} (x + 1, y + 1)$

$(x, y) \xrightarrow{SW} (x - 1, y - 1)$

$(x, y) \xrightarrow{SE} (x + 1, y - 1)$

Accept if $(2, 2)$ reached

Accepted sequences:

NE, NE

NE, SE, NE, NW ...

NE, NE, NE, SW ...

We can also create a transducer with outputs x-coordinate of the movements

Output of the input Ne, Se, Ne, Nw:

1, 0, 1, 2

Another example with the jug problem

Example

- $S = \{(i,j) \in \mathbb{N} \times \mathbb{N} : 0 \leq i \leq 5 \text{ and } 0 \leq j \leq 3\}$
- $s_0 = (0,0)$
- $\rightarrow$ given by
  - $(i,j) \xrightarrow{E5} (0,j)$ [empty 5L jug]
  - $(i,j) \xrightarrow{E3} (i,0)$ [empty 3L jug]
  - $(i,j) \xrightarrow{F5} (5,j)$ [fill 5L jug]
  - $(i,j) \xrightarrow{F3} (i,3)$ [fill 3L jug]
  - $(i,j) \xrightarrow{E35} (i+j,0)$ if $i+j \leq 5$ [empty 3L jug into 5L jug]
  - $(i,j) \xrightarrow{E53} (0,i+j)$ if $i+j \leq 3$ [empty 5L jug into 3L jug]
  - $(i,j) \xrightarrow{F53} (5,j-5+i))$ if $i+j \geq 5$ [fill 5L jug from 3L jug]
  - $(i,j) \xrightarrow{F35} (i-3+j,3)$ if $i+j \geq 3$ [fill 3L jug from 5L jug]
- Accept if $(4,0)$ is reached:

Give me an algorithm which reaches goal state is what acceptor asks for.

- Accept if $(4,0)$ is reached: e.g. F3, E35, F3, F53, E5, E35, F3, E35

Epsilon-transitions

It can be useful to allow the system to transition without taking input or producing output. We use the special symbol epsilon to denote those transitions. Epsilon can be seen as empty word.

Formal definitions

An **acceptor** is a $\Sigma \cup \{\epsilon\}$-labelled transition system $A = (S, \rightarrow, \Sigma, s_0, F)$ with a start state $s_0 \in S$ and a set of final states $F \subseteq S$.

A **transducer** is a $(\Sigma \cup \{\epsilon\}) \times (\Gamma \cup \{\epsilon\})$-labelled transition system $T = (S, \rightarrow, \Sigma, s_0, F)$ with a start state $s_0 \in S$ and a set of final states $F \subseteq S$.

Finite Automata

State transitions with a finite set of states are the useful ones in Computer Science.

Acceptors: **Finite state automata**

Transducers: **Mealy machines**

Everything is built upon a finite state transition system.

Lecture 14

Finite automata

In general transition systems we don't necessarily want a finite set of states, sometimes we want to work with an infinite set of states e.g. Natural numbers.

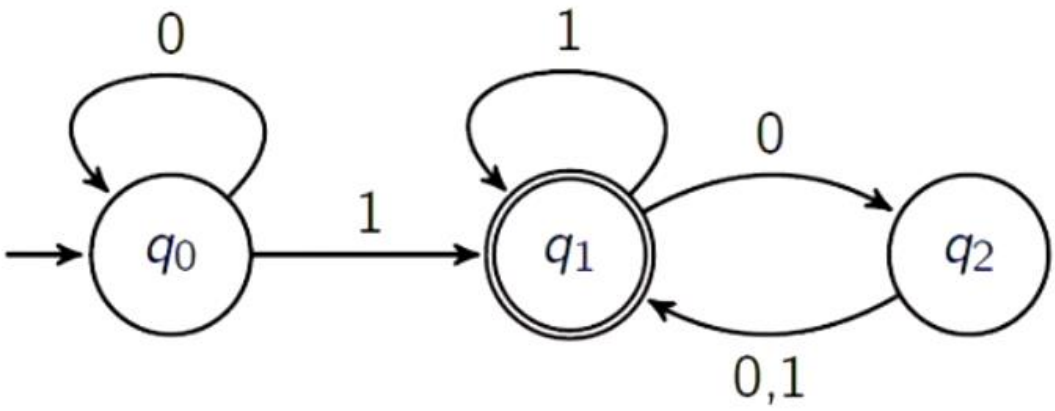
S is not always finite in our transition systems.

NB

In a non-deterministic transition system there may be many (including none) runs from a state. In an unlabelled deterministic transition system there is exactly one run from every state.

A deterministic Finite Automata is a deterministic finite state acceptor. DFAs represent computation with finite memory and form the backbone of most computational models.

Deterministic Finite Automata



Formally a deterministic finite automaton (DFA) is a 5-tuple of

Formally, a **deterministic finite automaton (DFA)** is a tuple $(Q, \Sigma, \delta, q_0, F)$ where

- Q is a finite set of states: $Q = \{q_0, q_1, q_2\}$
- Σ is the input alphabet: $\Sigma = \{0, 1\}$
- $\delta : Q \times \Sigma \to Q$ is the transition function
- $q_0 \in Q$ is the start state
- $F \subseteq Q$ is the set of final/accepting states: $F = \{q_1\}$

Note the final state is the one with a double circle, in this case F = {q1}

Our transition function takes input and current state, and shows where we transition to

$$\delta(q_0, 0) = q_0$$
$$\delta(q_0, 1) = q_1$$
$$\delta(q_1, 0) = q_2$$
$$\delta(q_1, 1) = q_1$$
$$\delta(q_2, 0) = q_1$$
$$\delta(q_2, 1) = q_1$$

A DFA accepts or rejects a sequence of symbols from our alphabet. A word will define a run in the DFA, and where the run ends determines if the word is rejected.

For example, w = 1001

Start in q0, input 1 → q1

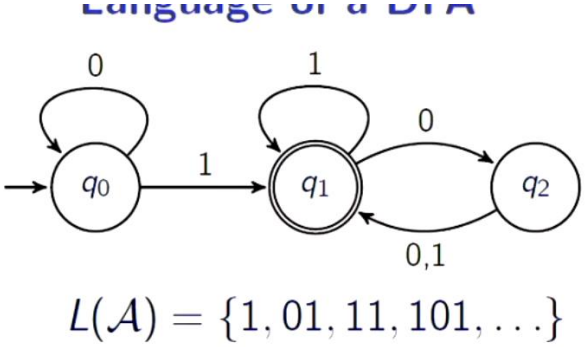
Q1 input 0 → q2

Q2 input 0 → q1

Q1 input 1 → q1 so w 1001 is accepted

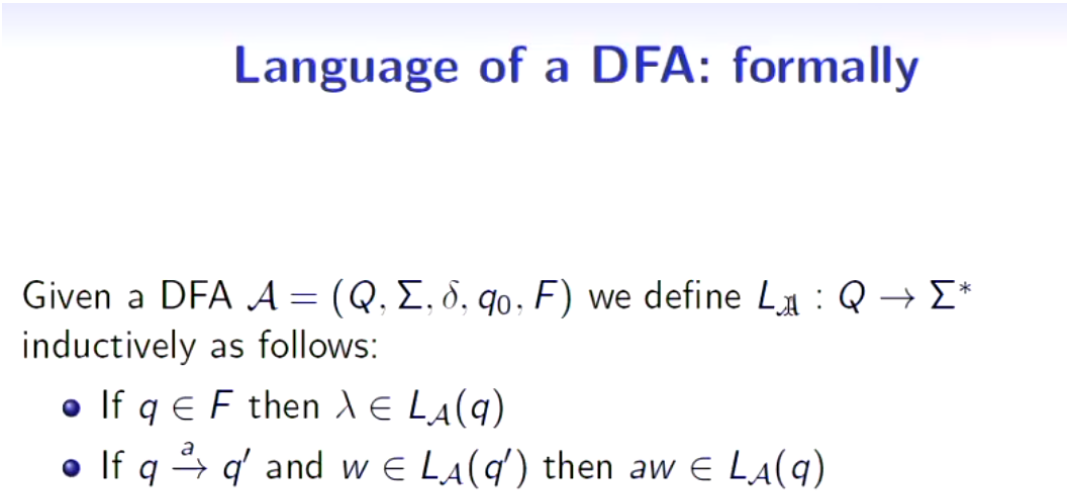
If a run ends in any state in the set of final states, then the word is accepted, otherwise it is rejected.

For a DFA the language of our DFA L(DFA) is the set of words from our alphabet which are accepted by our DFS



A language L is regular if there is some DFA such that L = L(A)

Formal definition of language of a DFA

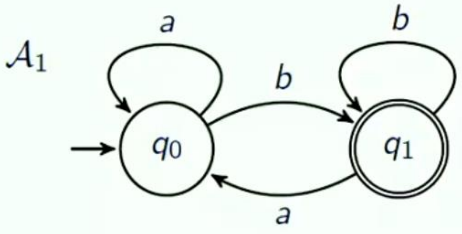


If my state is a final state, then the empty word is our language, otherwise the language is the transition required from our current state to the final state.

We then define L(A) = La(qo)

Example

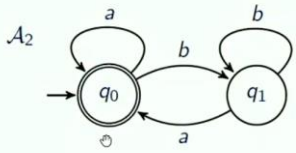
Example



$L(\mathcal{A}_1) = ?$

Language for this automaton is words that end with b

Example

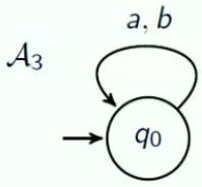


$L(\mathcal{A}_2) = ?$

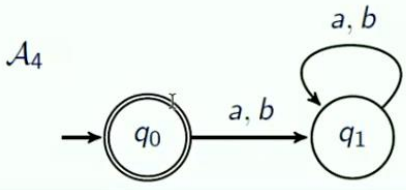
Language for this automaton is words that end with a AND the empty word because we initially start at the final state.

Example

Find $\mathcal{A}_3$ such that $L(\mathcal{A}_3) = \emptyset$



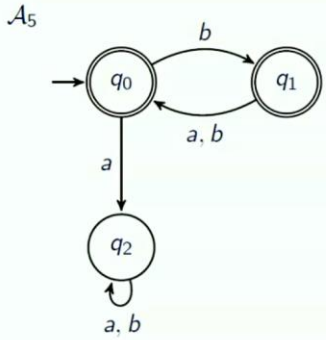
Find $\mathcal{A}_4$ such that $L(\mathcal{A}_4) = \{\lambda\}$



Example

Example

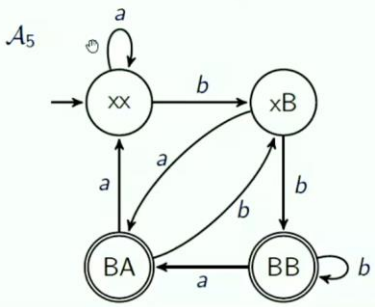
Find $\mathcal{A}_5$ such that $L(\mathcal{A}_5) = \{w \in \{a, b\}^* : \text{every odd symbol is } b\}$



HARD EXAMPLE 2nd last symbol is B

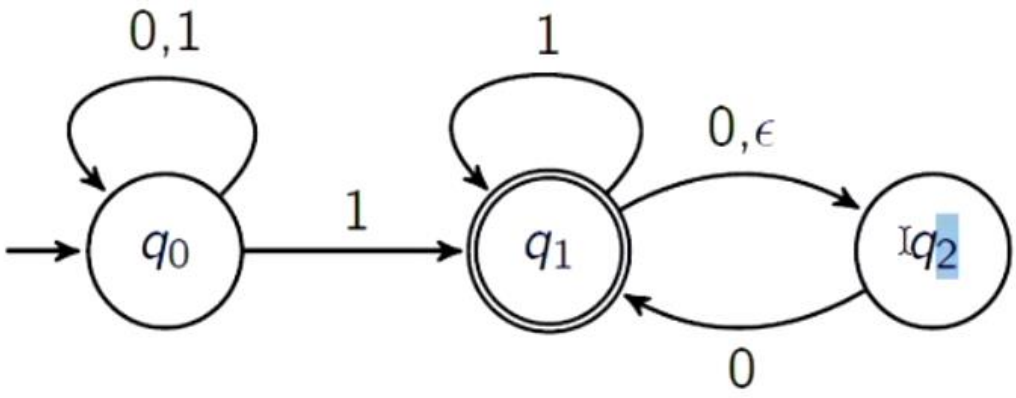
Example

Find $\mathcal{A}_6$ such that
$L(\mathcal{A}_6) = \{w \in \{a, b\}^* : \text{second-last symbol is } b\}$



Non-deterministic Finite Automata

A non-deterministic finite automaton NFA is a non-deterministic finite state acceptor, and it is more general than DFAs. A DFA is an NFA, but an NFA is not a DFA



Formally it is exactly the same, however instead of a transition function we have a transition relation

- $\delta \subseteq Q \times (\Sigma \cup \{\epsilon\}) \times Q$ is the transition relation

δ	ϵ	0	1
q_0	$\emptyset$	$\{q_0\}$	$\{q_0, q_1\}$
q_1	$\{q_2\}$	$\{q_2\}$	$\{q_1\}$
q_2	$\emptyset$	$\{q_1\}$	$\emptyset$

Language of an NFA

Non-determinism is interested in there existing a run, determinism only focuses on the one run. With a DFA a word defines a single run, whereas an NFA defines multiple runs, and in an NFA we accept words if there is at least one run, and a word is rejected if no runs from the word end in a final state. In an NFA we branch down multiple paths and one input can have different transition systems, so we only care for if one run can satisfy end state. An NFA will always choose wisely.

An NFA accepts a sequence of symbols from Σ – i.e. elements of Σ^*

Informally: A word defines several runs in the NFA and the word is accepted if **at least one run** ends in a final state.

Note 1: Runs can end prematurely (these don't count)

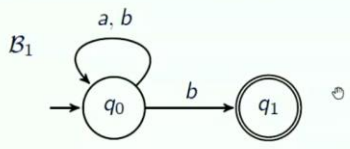
Note 2: An NFA will always "choose wisely"

Epsilon transitions don't consume input so we essentially make the move and we are at the same spot in our input for processing.

The language for an NFA is the same as the language of our DFAs

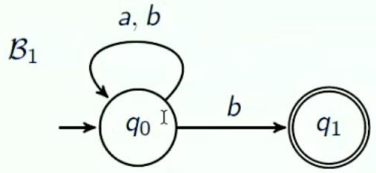
Examples for NFAs

Example



$L(\mathcal{B}_1) = \{w \in \{a, b\}^* : w \text{ ends with } b\}$

Example

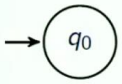


$$L(\mathcal{B}_1) = \{w \in \{a, b\}^* \ : \ w \text{ ends with } b\}$$

Example

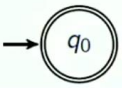
Find $\mathcal{B}_3$ such that $L(\mathcal{B}_3) = \emptyset$

$\mathcal{B}_3$



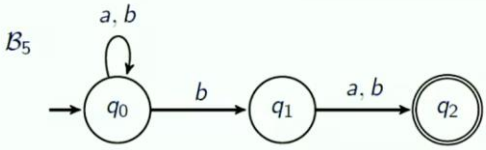
Find $\mathcal{B}_4$ such that $L(\mathcal{B}_4) = \{\lambda\}$

$\mathcal{B}_4$



Example

Find $\mathcal{B}_5$ such that $L(\mathcal{B}_5) = \{w \in \{a, b\}^* : \text{second-last symbol is } b\}$



Clearly for any DFA there is an NFA there is a language where L(DFA) = L(NFA)

And the converse is true

Clearly for any DFA $\mathcal{A}$ there is an NFA $\mathcal{B}$ such that $L(\mathcal{A}) = L(\mathcal{B})$.

Theorem

For any NFA $\mathcal{B}$ there is a DFA $\mathcal{A}$ such that $L(\mathcal{A}) = L(\mathcal{B})$.

NFA vs DFA

Theorem

- *For any NFA with n states there exists a DFA with at most 2^n states that accepts the same language*
- *There exist NFAs with n states such that the smallest DFA that accepts the same language has at least 2^n states.*

Regular Languages

A language L over an alphabet is regular if there is some DFA such that L = L(DFA)

Equivalently, there is some NFA such that L = L(B)

Regular languages

A language $L \subseteq \Sigma^*$ is **regular** if there is some DFA $\mathcal{A}$ such that $L = L(\mathcal{A})$

Equivalently, there is some NFA $\mathcal{B}$ such that $L = L(\mathcal{B})$

Non-Regular Languages

There are languages which are not regular, "Simple" counting argument: there are uncountably many languages, and only countably many DFAs

An example of a non-regular language: $\{0^n 1^n : n \in \mathbb{N}\}$
Intuitively: need arbitrary large memory to "remember" the number of 0's

Regular languages are mathematically nice

Theorem

If L is a regular language then $L^c = \Sigma^* \setminus L$ is a regular language.

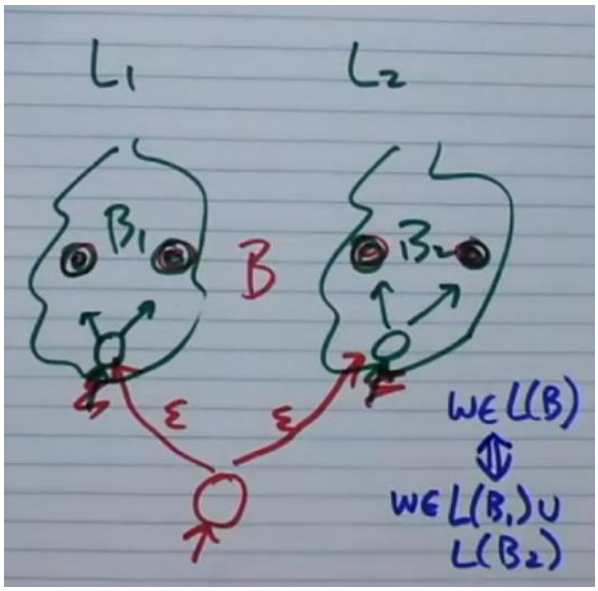
Proof:

- Let $\mathcal{A} = (Q, \Sigma, \delta, q_0, F)$ be a DFA such that $L(\mathcal{A}) = L$
- Consider $\mathcal{A}' = (Q, \Sigma, \delta, q_0, Q \setminus F)$
- For any word $w \in \Sigma^*$, the corresponding run in $\mathcal{A}$ is unique, so:
  - If $w \in L(\mathcal{A})$ then $w \notin L(\mathcal{A}')$, and
  - If $w \notin L(\mathcal{A})$ then $w \in L(\mathcal{A}')$,
- Therefore $L(\mathcal{A}') = \Sigma^* \setminus L(\mathcal{A}) = L^c$

THIS DOES NOT WORK FOR NFA's IT RELYS ON THE FACT THERE IS ONLY ONE SOLUTION!

If L1 and L2 are regular languages, then L1 U L2 is also regular

Proof



Theorem

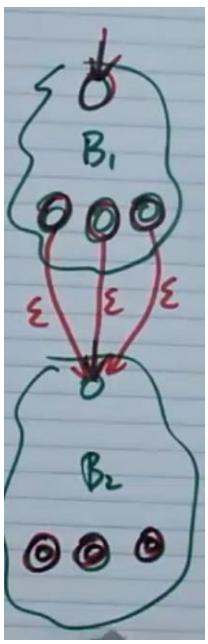
If L_1 and L_2 are regular languages, then $L_1 \cap L_2$ is regular.

Proof:

$$L_1 \cap L_2 = (L_1^c \cup L_2^c)^c$$

Theorem

If L_1 and L_2 are regular languages, then $L_1 \cdot L_2$ is regular.



If L is regular then L* is regular.

Lecture 15

If the transition system is a relation, then it is considered non-deterministic, but when a transition system is a function (1 input 1 output) it is deterministic.

Safety is dual of liveliness, (flipped).

Example

- States: $\mathbb{Z} \times \mathbb{Z} \times \mathbb{Z}$
- Transition:
  - $(x, y, r) \to (x^2, \frac{y}{2}, r)$ if y is even
  - $(x, y, r) \to (x^2, \frac{y-1}{2}, rx)$ if y is odd
- Preserved invariant: rx^y is a constant
- $\Rightarrow$ All states reachable from $(m, n, 1)$ will satisfy $rx^y = m^n$
- $\Rightarrow$ if $(x, 0, r)$ is reachable from $(m, n, 1)$ then $r = m^n$.

A transition system $(S, \to)$ **terminates** from a state s if there is an N such that all runs from s have length at most N.

A **derived variable** is a function $f : S \to \mathbb{R}$.

A derived variable is **strictly decreasing** if $s \to s'$ implies $f(s) > f(s')$.

Theorem

If f is an $\mathbb{N}$-valued, strictly decreasing derived variable, then the length of any run from s is at most $f(s)$.

Properties of regular languages

The class of regular languages is closed under:

- Complementation
- Intersection
- Union
- Concatenation
- Kleene Star
- Homomorphisms (symbol rewriting)
- Inverse homomorphisms

If any of these operations are performed on regular languages then the result is also a regular language.

When we write code and compile, it is fed into a finite automaton that will check if your code is syntactically correct. This is one application of finite automaton.

Need to know for this course

Transition systems:

- Definitions
- Invariant principle
- Termination proofs

DFAs/NFAs:

- Definitions
- Language accepted by a DFA/NFA
- Closure properties of regular languages

Regular Expressions (REGEX)

Regular expressions are a way of describing patterns (literally regex)

E.g.

- Second-last letter is b
- Every odd symbol is b

This has a lot of applications such as lexical analysis, grep, awk, text editors, databases, any form of REGEX.

What is a regular expression?

Given a finite set of symbols, a regular expression over our symbols is defined recursively as the following

- Base case
  - Empty symbol is a regular expression
  - Epsilon (skip input) is a regular expression
  - Any character in our set of symbols is a regular expression
- Inductive case
  - If E1 and E2 are regular expressions, then E1E2 is also a regular expression
  - If E1 and E2 are regular expressions then E1 + E2 is also a regular expression
  - If E is a regular expression then E* is a regular expression

Epsilon can be thought of as all the things between our pattern when we regex match (the rest of the line).

We use parentheses to disambiguate Regular expressions, however * binds tighter than concat, and concat binds tighter than +.

Example

Example

The following are regular expressions over $\Sigma = \{0, 1\}$:

- $\emptyset$
- $101 + 010$
- $(\epsilon + 10)^*01$

A regular expression defines a language over an alphabet which matches expressions.

- Concatenation = sequence of expressions
- Union = choice of expressions
- Star = 0 or more occurrences of an expression

Example

The following words match $(000 + 10)^*01$:

- 01
- 101001
- 000101000001

Formal definition of the language of a regular expression

Formally given a regular expression E over an alphabet we define the language of our regular expression $L(E) \subseteq \Sigma^*$ recursively as follows

- Base case
  - If $E = \emptyset$ then $L(E) = \emptyset$
  - If $E = \epsilon$ then $L(E) = \{\lambda\}$
  - If $E = a$ where $a \in \Sigma$ then $L(E) = \{a\}$
- Inductive case
  - If $E = E_1 E_2$, then $L(E) = L(E_1) \cdot L(E_2)$
  - If $E = E_1 + E_2$, then $L(E) = L(E_1) \cup L(E_2)$
  - If $E = E_1^*$ then $L(E) = (L(E_1))^*$

For example

- L (010 + 101) = L (010) U L (101) = {010, 101}
- $L((\epsilon + 10)^*01) = \{01, 1001, 101001, \ldots\}$

Adding a star turns finite language into infinite languages 0 … many relation.

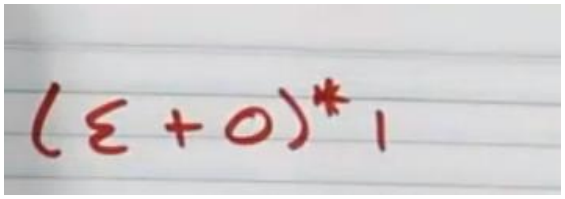
Kleene's Theorem

Theorem (Kleene's theorem)

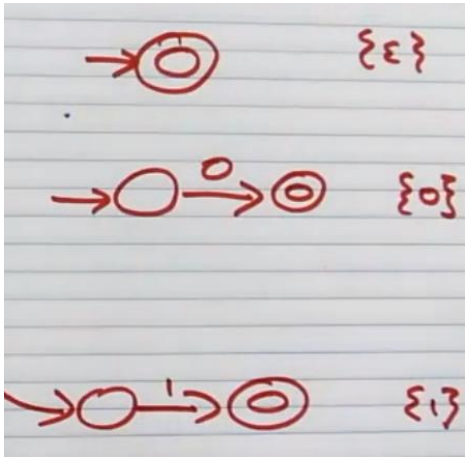
- *For any regular expression E, $L(E)$ is a regular language.*
- *For any regular language L, there is a regular expression E such that $L = L(E)$*

Some examples

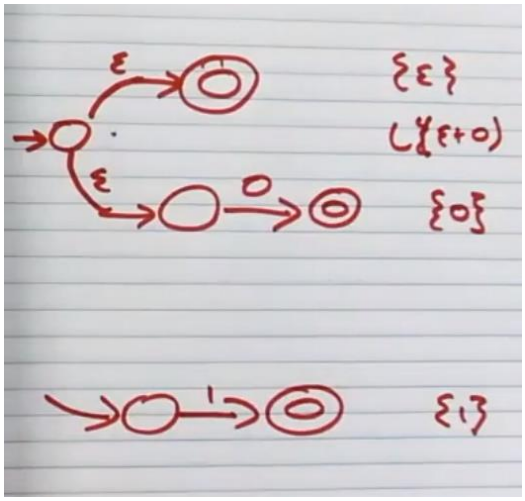
Find NFA for



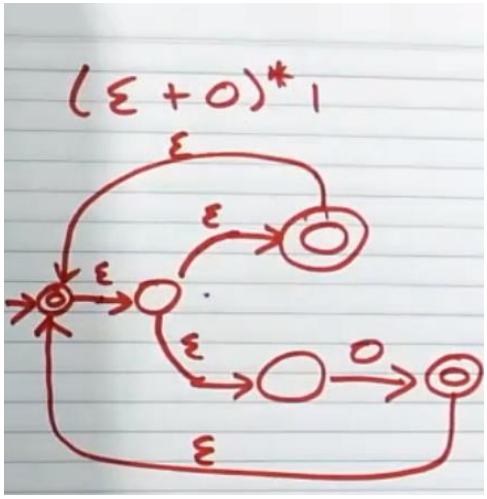
Step 1 we need our base cases for each symbol



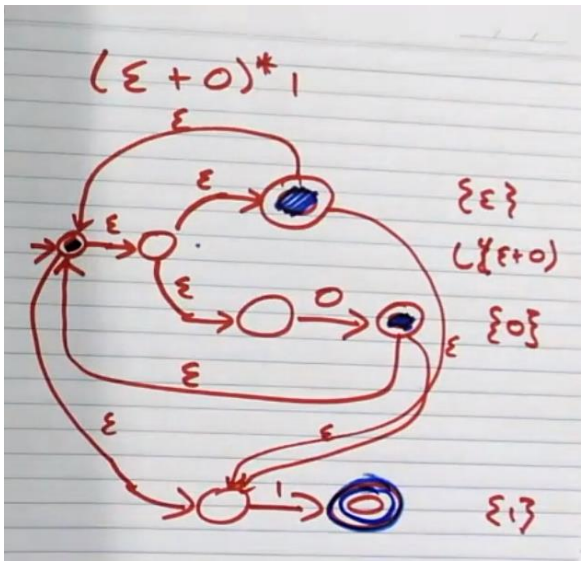
Now we construct (epsilon + 0)



Now to add star operation we need to make a way to loop back to the beginning from (epsilon + 0)



Finally, to concatenate our (epsilon + 0) with 1 we replace the final states for (e+0) with just regular state, and then perform epsilon transitions to 1

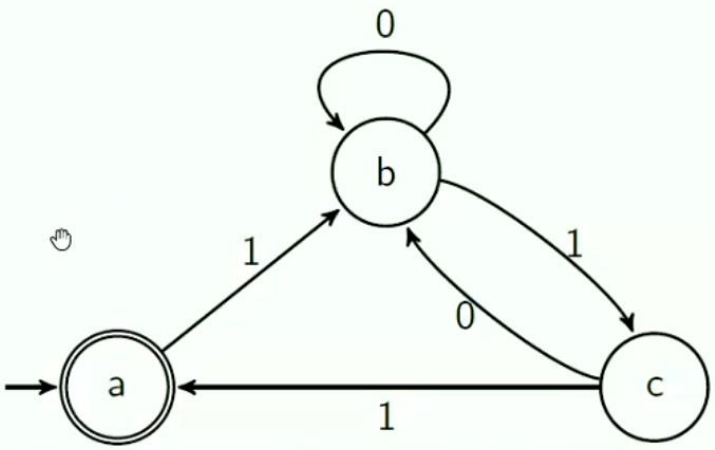


We keep building it up from base cases, however it is much simpler to perform this.

(0 or epsilon self-loop).

Example

Find a regular expression for this NFA:



Myhill-Nerode theorem

L-indistinguishability

Let $x, y \in \Sigma^*$ and let $L \subseteq \Sigma^*$.
We say that x and y are L-**indistinguishable**, written $x \equiv_L y$, if for every $z \in \Sigma^*$,

$$xz \in L \quad \text{if and only if} \quad yz \in L.$$

Two words are indistinguishable if we append a word z onto the end of x and y and the resulting words are both in the language or not in the language.

Essentially, whenever xz is in the language, yz is in the language and whenever xz isn't in the language, then yz is not in the language.

Fact

$\equiv_L$ *is an equivalence relation.*

We define the **index** of L to be the number of equivalence classes of $\equiv_L$.

NB

The index of L may be finite or infinite.

Examples

Let's say our alphabet is {0,1}

$$L_1 = \{w : w \text{ has even length}\}.$$

Then we say two words are indistinguishable if the following

$$u \equiv_{L_1} v \text{ iff } \text{length}(u) \equiv \text{length}(v) (\text{mod } 2).$$

Now $\equiv_{L_1}$ has two equivalence classes:
$[\epsilon] = [00] = [10] = \cdots = \{w : \text{length}(w) \text{ even}\}$ and
$[0] = [1] = [010] = [110] = \cdots = \{w : \text{length}(w) \text{ odd}\}.$

Another example

Example

Take $\Sigma = \{0, 1\}$

$$L_2 = \{w : w \text{ has equal numbers of 0s and 1s}\}.$$

For any $i, j \geq 0$, if $i \neq j$ then $0^i \not\equiv_{L_2} 0^j$ (because $0^i 1^i \in L_2$ but $0^j 1^i \notin L_2$).

Therefore the index of L_2 is infinite.

The theorem itself

L is regular if and only if L has finite index.

The index is the size of the smallest DFA accepting L (smallest number of states).

Example

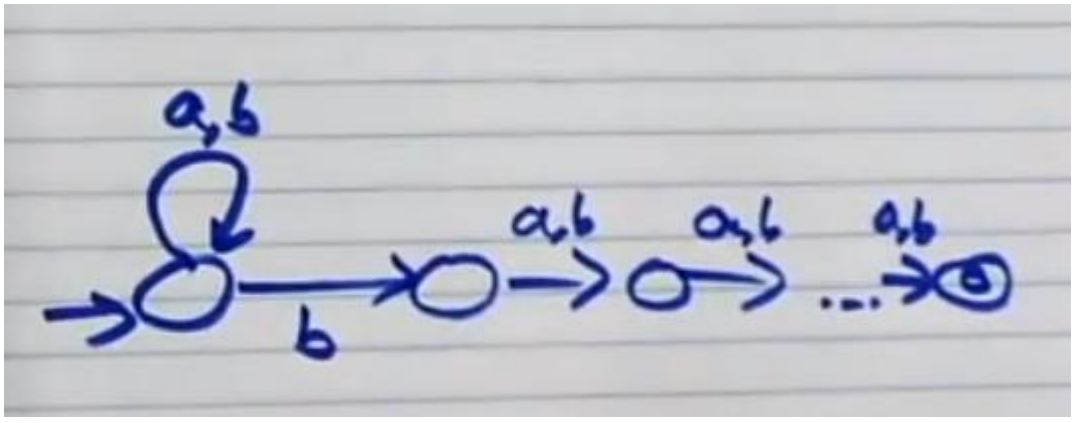
Take $\Sigma = \{a, b\}$.

$$L_n = \{w : \text{ the } n\text{-th last symbol of } w \text{ is } b\}$$

What is the index of L_n?

How do we find size of the DFA needed?

This can be accepted by an NFA with n states



We can make a DFA for this with 2^n states to represent this NFA.

So, the index at most is 2^n.

What is the index of L_n?

Take $w, v \in \Sigma^n$ with $w \neq v$. Suppose w and v differ in the i-th symbol, $0 \leq i \leq n-1$. Let $z = a^{n-i}$.

- Then only one of wz, vz is in L_n
- So w and v are L_n-distinguishable ($w \not\equiv_{L_n} v$)
- So the index is at least 2^n

This also tells us the blow up from NFA → DFA is at least 2^n.

Context-free grammars

Regular languages can be specified in terms of finite automata that accept or reject strings, equivalently, in terms of regular expressions, which strings are to match.

Grammars are a generative means of specifying sets of strings.

To construct a CFG we need

- Variables
- Terminals
- Productions (rules)
- Start symbol

The start symbol is a special variable. A CFG generates strings over the alphabet $\Sigma = \{\text{terminals}\}$.

Example

$G = (\{A, B\}, \{0, 1\}, \mathcal{R}, A)$ where $\mathcal{R}$ consists of three rules:

$$\begin{cases} A \rightarrow 0\,A\,1 \\ A \rightarrow B \\ B \rightarrow \epsilon \end{cases}$$

How to generate strings using a CFG

1. Set w to be the start symbol.
2. Choose an occurrence of a variable X in w if any, otherwise STOP.
3. Pick a production whose lhs is X, replace the chosen occurrence of X in w by the rhs.
4. GOTO 2.

Example

$G = (\{A, B\}, \{0, 1\}, \{A \rightarrow 0\,A\,1 \mid B, \quad B \rightarrow \epsilon\}, A)$ generates $\{0^i\,1^i : i \geq 0\}$.

$$\begin{aligned} A &\Rightarrow 0\,A\,1 \\ &\Rightarrow 0\,0\,A\,1\,1 \\ &\Rightarrow 0\,0\,B\,1\,1 \\ &\Rightarrow 0\,0\,\epsilon\,1\,1 = 0^2\,1^2 \end{aligned}$$

Such sequences are called **derivations**.

Lecture 16

Context-free languages

Grammars are a way of generating languages! Machines consume words as acceptors, regular expressions match things inside the language. Grammars work the other way around and produce words inside the language.

CFGS composition

To make CFG we need

- A set of non-terminal variables
- A set of terminals
- A set of productions (rule set) rewrite rules
- A set of starting symbols

Example

$G = (\{A, B\}, \{0, 1\}, \mathcal{R}, A)$ where $\mathcal{R}$ consists of three rules:

$$\begin{cases} A \rightarrow 0A1 \\ A \rightarrow B \\ B \rightarrow \epsilon \end{cases}$$

What makes this a Context-free grammar is that on the left-hand side of the rule, we only have one thing. If we had multiple things on the right-hand side of the rule, we would get unrestricted grammars.

Essentially what CFG rule works like is that, we start at A and we can choose one of our three re-write rules. E.g. Rewrite A → 0A1 or B, and if we see a B, we can rewrite it to epsilon. And we can keep doing this as many times as we want to, since we have no terminal symbols on the left-hand side.

Another example.

How to generate strings using a CFG

1. Set w to be the start symbol.
2. Choose an occurrence of a variable X in w if any, otherwise STOP.
3. Pick a production whose lhs is X, replace the chosen occurrence of X in w by the rhs.
4. GOTO 2.

Example

$G = (\{A, B\}, \{0, 1\}, \{A \rightarrow 0A1 \mid B, \quad B \rightarrow \epsilon\}, A)$ generates $\{0^i 1^i : i \geq 0\}$.

$$\begin{aligned} A &\Rightarrow 0A1 \\ &\Rightarrow 00A11 \\ &\Rightarrow 00B11 \\ &\Rightarrow 00\epsilon11 = 0^2 1^2 \end{aligned}$$

Such sequences are called **derivations**.

The set of words we can derive is the language we generate.

Formal definitions

A CFG is a 4-tuple G = (variables, terminals, rules, starts) where

- Variables are a finite set of non-terminal variables
- Terminals are a finite set of ends
- Rules are a finite set of productions
  - element of $V \times (V \cup \Sigma)^*$, written $A \to w$.
- S is in the set of variables and is the start symbol

We define a binary relation $\Rightarrow$ over $(\{V \cup \Sigma\})^*$ by: for each $u, v \in (\{V \cup \Sigma\})^*$, for each $A \to w$ in $\mathcal{R}$

$$u\,A\,v \Rightarrow u\,w\,v$$

The **language generated by the grammar**, $L(G)$, is $\{w \in \Sigma^* : S \Rightarrow^* w\}$.

A language is **context-free** if it can be generated by a CFG.

We can non-deterministically pick rules for our grammar to give us derivations.

Being context free we can rewrite ANY variable occurrence with our rule.

An example of another CFG to generate a set of parentheses would be

Example

Well-balanced parentheses: generated by $(\{S\}, \{(,)\}, \mathcal{R}, S)$ where $\mathcal{R}$ consists of

$$S \to (\,S\,) \mid S\,S \mid \epsilon$$

E.g. (( ) (( ))) ( )

Examples

Inductively defined syntax:

- Well-formed formulas
- L
- Regular expressions
- Code specifications

WFFs: Generated by $(\{\varphi\}, \Sigma, \mathcal{R}, \varphi)$ where $\Sigma = \text{PROP} \cup \{\top, \bot, (,), \neg, \wedge, \vee, \to, \leftrightarrow\}$ and $\mathcal{R}$ consists of

$$\varphi \to \top \mid \bot \mid P \mid \neg\varphi \mid (\varphi \wedge \varphi) \mid (\varphi \vee \varphi) \mid (\varphi \to \varphi) \mid (\varphi \leftrightarrow \varphi)$$

Example 2

English language

Example

A small English language

⟨SENTENCE⟩ → ⟨NOUN-PHRASE⟩ ⟨VERB-PHRASE⟩
⟨NOUN-PHRASE⟩ → ⟨CMPLX-NOUN⟩ | ⟨CMPLX-NOUN⟩ ⟨PREP-PHRASE⟩
⟨VERB-PHRASE⟩ → ⟨CMPLX-VERB⟩ | ⟨CMPLX-VERB⟩ ⟨PREP-PHRASE⟩
⟨PREP-PHRASE⟩ → ⟨PREP⟩ ⟨CMPLX-NOUN⟩
⟨CMPLX-NOUN⟩ → ⟨ARTICLE⟩ ⟨NOUN⟩
⟨CMPLX-VERB⟩ → ⟨VERB⟩ | ⟨VERB⟩ ⟨NOUN-PHRASE⟩
⟨ARTICLE⟩ → a | the
⟨NOUN⟩ → boy | girl | flower
⟨VERB⟩ → touches | like | see
⟨PREP⟩ → with

Here is an example on how to generate a sentence starting with “a”

⟨SENTENCE⟩ ⇒ ⟨NOUN-PHRASE⟩ ⟨VERB-PHRASE⟩
⇒ ⟨CMPLX-NOUN⟩ ⟨PREP-PHRASE⟩ ⟨VERB-PHRASE⟩
⇒ ⟨ARTICLE⟩ ⟨NOUN⟩ ⟨PREP-PHRASE⟩ ⟨VERB-PHRASE⟩
⇒ a girl ⟨PREP⟩ ⟨CMPLX-NOUN⟩ ⟨VERB-PHRASE⟩
⇒ a girl with ⟨CMPLX-NOUN⟩ ⟨VERB-PHRASE⟩
⇒ a girl with ⟨ARTICLE⟩ ⟨NOUN⟩ ⟨VERB-PHRASE⟩
⇒ a girl with a flower ⟨VERB-PHRASE⟩
⇒ a girl with a flower ⟨CMPLX-VERB⟩
⇒ a girl with a flower ⟨VERB⟩ ⟨NOUN-PHRASE⟩
⇒ a girl with a flower likes ⟨CMPLX-NOUN⟩
⇒ a girl with a flower likes ⟨ARTICLE⟩ ⟨NOUN⟩
⇒ a girl with a flower likes the boy

A CFG is right-linear if every rule is either in the form of R → wT or of the form R → w, where w ranges over strings of terminals and R and T over variables

A CFG is **right-linear** if every rule is either of the form $R \to wT$ or of the form $R \to w$ where w ranges over strings of terminals, and R and T over variables.

Theorem

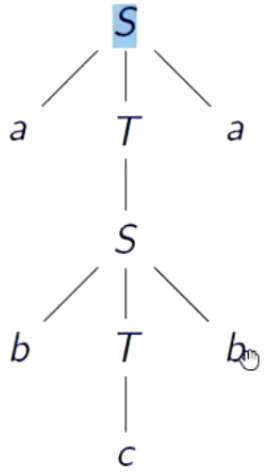
A language is regular if and only if it is generated by a right-linear CFG.

Regular Languages are a subset of Context-Free Languages. In our right-linear CFG, variables go to either variables, or variables and terminals

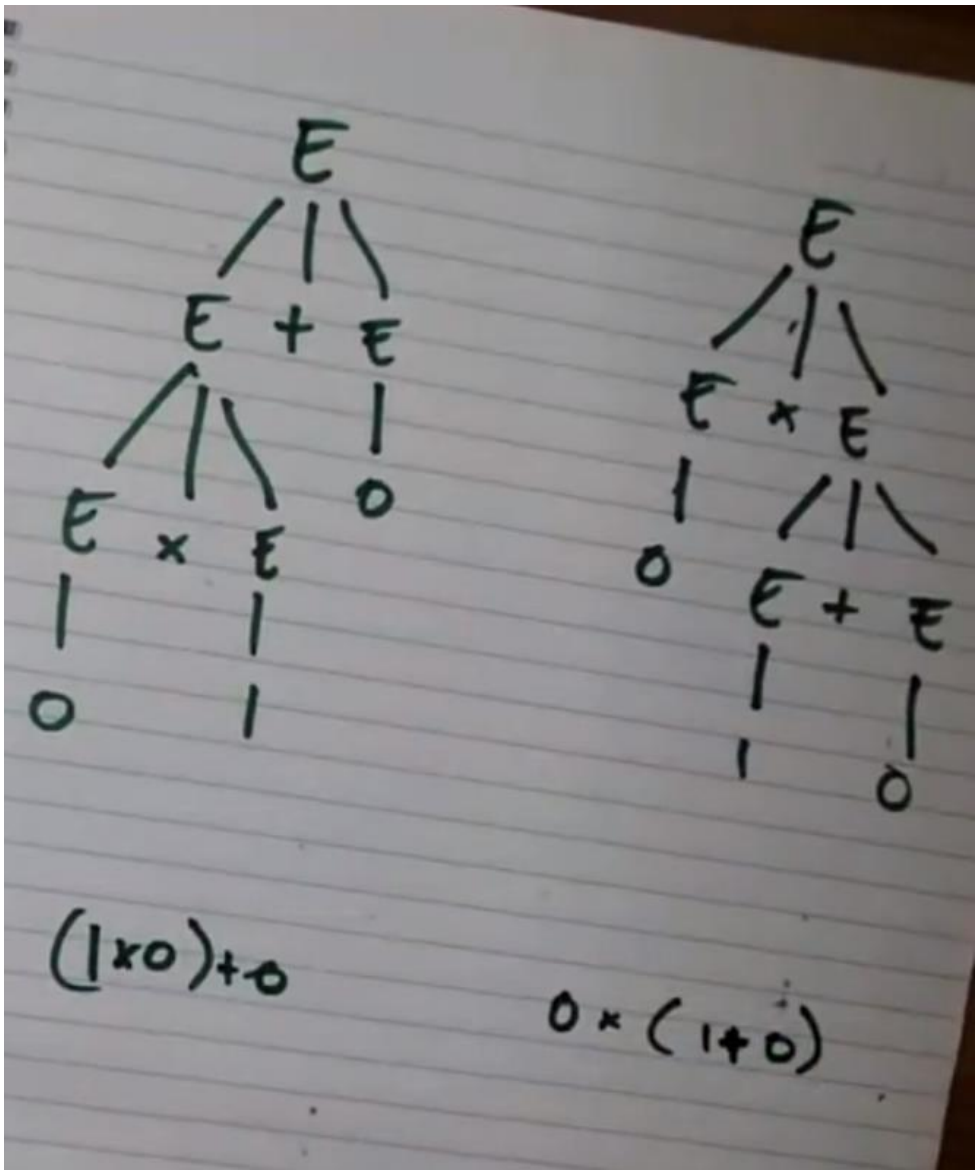
Parse trees

Each derivation determines a **parse tree**.
Parse trees are *ordered* trees: the children at each node are ordered.
The parse tree of a derivation abstracts away from the order in which variables are replaced in the sequence.

$$
\begin{aligned}
S &\Rightarrow a\,T\,a \\
&\Rightarrow a\,S\,a \\
&\Rightarrow a\,b\,T\,b\,a \\
&\Rightarrow a\,b\,c\,b\,a
\end{aligned}
$$



Example



Context Free Languages are

- Closed under union
- Not closed under complement or intersection

Mealy Machines

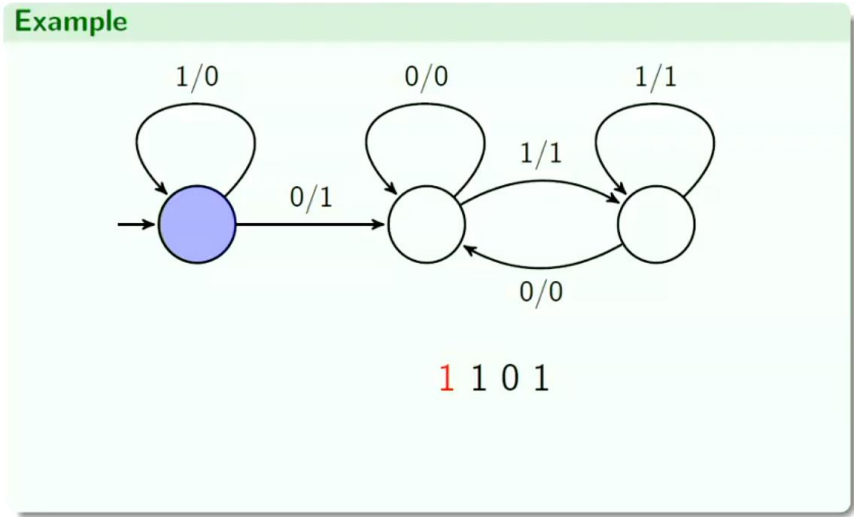
A Mealy machine is a finite state deterministic transducer which has

- A finite set of states
- Input alphabet
- Output alphabet
- Transition function not relation! 1-1
- Start states

A Mealy machine deals with inputs and outputs at the same time, in a finite state transition way (deterministic transducer).

DFAs accept languages, Mealy machines compute length-preserving functions.

In mealy machines, input on left hand side of /, right hand side is output



0011 is output if u follow I/O transitions.

Mealy machines model I/O systems.

Moore Machines outputs at states rather than transitions which is better for synchronicity.

Models of computation

Now that we have our specifications, we want to add requirements to achieve this model.

We would like to reason about them:

- Every request is eventually responded to
- The traffic light is not always red
- The brakes are applied until the pedal is released
- If φ holds in a state, then ψ will hold in the successor state

First Order Logic Pros and Cons

Pros

- Very expressive specification language
- Satisfiability is decidable

Cons

- “readability” can be an issue
- Satisfiability checking is of non-elementary complexity

The way we check satisfiability is, we convert to automaton and check if there is a run to satisfy.

LTL

LTL is an extension of Propositional Logic with the ability to work with state transitions

LTL is a restriction of Predicate Logic with limitations on various constructs such as quantifiers

LTL pros and cons

Pros:

- Logical language close to English
- Satisfiability checking is (relatively) efficient
- Quite expressive (same as FO on paths)

Cons:

- Does not take "branching" into account (see CTL)
- Satifiability is still computationally difficult

LTL Syntax

The **formulas of LTL** are defined recursively as follows:

- $\top$, $\bot$, p ($p \in \text{PROP}$) are all formulas;
- If φ, ψ are formulas then so are:
  - $\neg\varphi$
  - $(\varphi \wedge \psi), (\varphi \vee \psi), (\varphi \rightarrow \psi), (\varphi \leftrightarrow \psi)$
  - $\mathbf{X}\varphi$
  - $\varphi \mathbf{U} \psi$

NB

X and U are known as **temporal operators**.

Xphi means Phi holds in the state after X

U operator means until, so Phi holds until PSI

Example

Some formulas:

- $\top \mathbf{U}(p \wedge q)$
- $\mathbf{X}(p \vee (q \mathbf{U} \neg p))$

Derived operators are

Three additional common operators:

- $\mathbf{F}\varphi$: eventually (in the **F**uture) φ, for $\top\mathbf{U}\varphi$
- $\mathbf{G}\varphi$: always (**G**lobally) φ, for $\neg(\top\mathbf{U}(\neg\varphi))$
- $\varphi\mathbf{W}\psi$: φ **W**eakly until ψ, for $(\mathbf{G}\varphi) \vee (\varphi\mathbf{U}\psi)$
- $\varphi\mathbf{R}\psi$: φ **R**eleases ψ, for $\neg(\neg\varphi\mathbf{U}\neg\psi)$

Example

- Every request is eventually responded to: $\mathbf{G}(\text{req} \rightarrow \mathbf{F}\text{resp})$
- The traffic light is not always red: $\neg\mathbf{G}\text{red}$
- The brakes are applied until the pedal is released: $\text{brake}\mathbf{U}(\neg\text{pedal})$
- If φ holds in a state, then ψ will hold in the successor state: $\varphi \rightarrow \mathbf{X}\psi$

Lecture 19

Fundamentals

- Sets
- Languages
- Relations and Functions

Need to know for this course:

- Formal language definitions
- Relation/function definitions
- Equivalence relations
- Partial orders

Example

Common relations and their properties

	(R)	(AR)	(S)	(AS)	(T)
$=$	✓		✓	✓	✓
$\leq$	✓			✓	✓
$<$		✓		✓	✓
$\emptyset$		✓	✓	✓	✓
$\mathcal{U}$	✓		✓		✓
$\mid$	✓			✓	✓

Relation/Function definitions

- Reflexive, anti-reflexive
- Symmetric, anti-symmetric
- Transitive
- Composition, converse, inverse
- Injective, surjective, bijective

Set Theory and Boolean Algebras

- Sets
- Boolean Algebras

Need to know for this course:

- Proofs using the Laws of Set Operations
- Proofs using the Laws of Boolean Algebras
- Principle of duality

Conjunctive/Disjunctive Normal Forms

CNFs and DNFs are *syntactic* forms:

Literal: A propositional variable or the negation of a propositional variable

Clause: A CNF-clause is a disjunction ($\vee$) of literals. A DNF-clause is a conjunction ($\wedge$) of literals

CNF/DNF: A formula is in CNF (DNF) if it is a conjunction (disjunction) of CNF-clauses (DNF-clauses).

Theorem

Every propositional formula is logically equivalent to one in CNF and one in DNF.