P. Madhusudan, Mahesh Viswanathan

Logic in Computer Science

Rough course notes

October 11, 2021

Contents

1 Propositional Logic 1
- 1.1 Syntax 3
- 1.2 Semantics 6
- 1.3 Satisfiability and Validity 9
- 1.4 Compactness Theorem 16
  - 1.4.1 Compactness using König's Lemma 16
  - 1.4.2 Compactness using Henkin Models 19
  - 1.4.3 An Application of Compactness: Coloring Infinite Planar Graphs 20

2 Proof Systems 23
- 2.1 A Frege-style Proof System 24
  - 2.1.1 Completeness Theorem 26
- 2.2 Resolution 27
  - 2.2.1 Proving Tautologies with Resolution 30
  - 2.2.2 Completeness of Resolution 32
- 2.3 Craig's Interpolation Theorem and Proof Complexity 34
  - 2.3.1 Craig's Interpolation Theorem 34
  - 2.3.2 Size of Interpolants 35
  - 2.3.3 Interpolants from Refutations 38
  - 2.3.4 Lower bounds on Resolution Refutations 44

3 First Order Logic 49
- 3.1 Syntax 49
- 3.2 Semantics 51
  - 3.2.1 Satisfiability, Validity, and First order theories 55
- 3.3 Overview 56

4 Quantifier Elimination and Decidability 59
- 4.1 Dense Linear Orders without Endpoints 61
- 4.2 Linear Arithmetic 64

4.2.1 Fourier-Motzkin 65
4.2.2 Ferrante-Rackoff 68
4.3 Other theories that admit quantifier elimination 70

5 Lower Bounds for the Validity Problem 71
5.1 Number Lines 72
5.2 Church-Turing Theorem 74
5.3 Trakhtenbrot's Theorem 78

A Computability and Complexity Theory 81
A.1 Turing Machines 82
A.2 Church-Turing Thesis 85
A.3 Recursive and Recursively Enumerable Languages 87
A.4 Reductions 89
A.5 Complexity Classes 92
A.6 Relationship between Complexity Classes 95
A.7 P and NP 97
A.7.1 Alternate characterization of NP 98
A.7.2 Reductions, Hardness and Completeness 99

Chapter 1
Propositional Logic

Modern logic is a formal, symbolic system that tries to capture the principles of correct reasoning and truth. To describe any formal language precisely, we need three pieces of information — the *alphabet* describes the symbols used to write down the sentences in the language; the *syntax* describes the rules that must be followed for describing "grammatically correct" sentences in the language; and finally, the *semantics* gives "meaning" to the sentences in our formal language. You may have encountered other contexts where formal languages were introduced in such a manner. Here are some illustrative examples.

Example 1.1 Binomial coefficients are written using natural numbers and parentheses. However, not every way to put together parenthesis and natural numbers is a binomial coefficient. For example, (1, (1), or $\binom{2}{}$ are examples of things that are no binomial coefficients. Correctly formed binomial coefficients are of the form $\binom{i+j}{i}$, where i and j are natural numbers. We could define the meaning of $\binom{i+j}{i}$ to be the natural number $\frac{(i+j)!}{i!j!}$. On the other hand, we could define the meaning of $\binom{i+j}{i}$ to be the number of ways of choosing i elements from a set of $i + j$ elements. Though both these ways of interpreting binomial coefficients are the same, they have a very different presentation. In general, one could define semantics in different ways, or even very different semantics to the same syntactic objects.

Example 1.2 Precise definitions of programming languages often involve characterizing its syntax and semantics. Turtle is an extremely simple programming language for drawing pictures. Programs in this language are written using **F**, +, and −. Any sequence formed by such symbols is a syntactically correct program in this language. We will interpret such a sequence of symbols as instructions to draw a picture — **F** is an instruction to draw a line by moving forward 1 unit; + is an instruction to turn the heading direction 60° to the left; − is an instruction to turn the heading direction 60° to the right. Figure 1.1 shows example programs and the pictures they draw based on this interpretation.

Even though Turtle is a very simple programming language, some very interesting curves can be approximated. Consider the following iterative procedure that produces

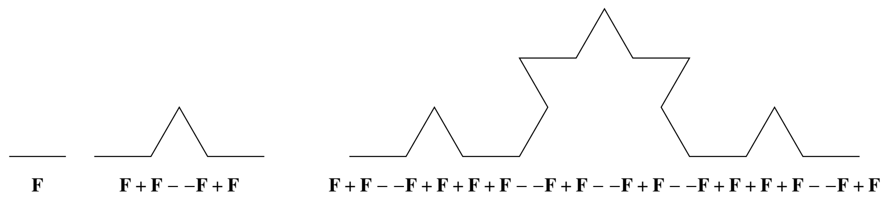
F
F + F − −F + F
F + F − −F + F + F + F − −F + F − −F + F − −F + F + F + F − −F + F

Fig. 1.1 Example Turtle programs and the pictures they draw.

a sequence of programs. Start with the program **F**. In each iteration, if P is a program at the start of the iteration, then construct the program P' obtained by replacing each **F** in P by **F + F − −F + F**. So at the beginning we have program **F**, in the next iteration the program is **F + F − −F + F**, and in the iteration after that it will be **F + F − −F + F + F + F − −F + F − −F + F − −F + F + F + F − −F + F**, and so on. The programs in this sequence draw pictures that in the limit approach the Koch curve.

Example 1.3 Regular expressions define special collections of strings called regular languages. Regular expressions over an alphabet Σ are built up using Σ, parentheses, $\emptyset$, ε, $\cdot$, $+$, and *. Inductively, they are defined as the smallest set that satisfy the following rules.

- $\emptyset$ and ε are regular expressions.
- For any $a \in \Sigma$, a is a regular expression.
- If r_1, r_2 are regular expressions then so are $(r_1 \cdot r_2)$, $(r_1 + r_2)$, and (r_1^*).

Each regular expression r, semantically defines a subset of Σ^* [1] that we will denote by $[\![r]\!]$. The semantics of regular expressions is defined inductively as follows.

- $[\![\emptyset]\!] = \emptyset$, and $[\![\varepsilon]\!] = \{\varepsilon\}$.
- For $a \in \Sigma$, $[\![a]\!] = \{a\}$.
- Inductively, $[\![(r_1 + r_2)]\!] = [\![r_1]\!] \cup [\![r_2]\!]$, $[\![(r_1 \cdot r_2)]\!] = [\![r_1]\!] \cdot [\![r_2]\!]$ and $[\![(r_1^*)]\!] = [\![r_1]\!]^*$, where $\cdot$ (on the right hand side) denotes the concatenation of two languages, and * denotes the Kleene closure of a language.

We will now define one of the simplest logics encountered in an introductory discrete mathematics class called *propositional* or *sentential* logic. Propositional logic is a symbolic language to reason about *propositions*. Propositions are declarative sentences that are either true or false. Examples of such include "Springfield is the capital of Illinois", "1+1 = 2", "2+2 = 0". Notice that propositions don't need to be true facts and their truth may depend on the context. For example, "2+2 = 0" is not true under the standard interpretation of + as integer addition, but is true if + denotes addition modulo 4. Non-examples of propositions include questions (like "What is it?"), commands (like "Read this!"), and things like "Location of robot".

[1] For a finite set Σ, Σ^* denotes the collection of (finite) sequences/strings/words over Σ. For $n \in \mathbb{N}$, we use Σ^n to denote the set of sequences/strings/words over Σ of length exactly n.

The logic itself will be symbolic and abstract away from English sentences like the ones above. We will introduce a precise definition of this logic, much in the same way as Example 1.3, defining the syntax and semantics inductively.

1.1 Syntax

We will assume a (countably infinite) set of *propositions* $\mathsf{Prop} = \{p_i \mid i \in \mathbb{N}\}$. The formulas of propositional logic will be strings over the alphabet $\mathsf{Prop} \cup \{(,), \neg, \vee\}$. Here $\neg$ is *negation*, and $\vee$ is *disjunction*.

Definition 1.4 The set of *well formed formulas* (wff) in propositional logic (over the set Prop) is the smallest set satisfying the following properties.

1. Any proposition $p_i \in \mathsf{Prop}$ is a wff.
2. If φ is a wff then $(\neg\varphi)$ is a wff.
3. If φ and ψ are wffs then $(\varphi \vee \psi)$ is a wff.

Examples of wffs include p_1, $(\neg p_1)$, $(p_1 \vee p_2)$, $((\neg(p_1 \vee p_3)) \vee (p_1 \vee p_4))$. On the other hand the following strings are not wffs: $(p_1\neg)$, $(\vee p_1)$, $(p_1\vee)$.

Inductive definitions of the kind in Example 1.3 or Definition 1.4 are quite common when defining the syntax of formulas in a logic or of programming languages. Therefore, in computer science, one often uses a "grammar-like" presentation for the syntax. For example, wffs φ in propositional logic are given by the following *BNF grammar*.

$$\varphi ::= p \mid (\neg\varphi) \mid (\varphi \vee \varphi) \tag{1.1}$$

where p is an element of Prop. Reading such grammars takes some getting used to. For example, the rule $(\varphi \vee \varphi)$ doesn't mean that disjunctions can only be used when the two arguments are the same. Instead it says that if we take two elements that belong to the syntactic entity φ (i.e., wffs), put $\vee$ between them with surrounding parenthesis, then we get another element belonging to the same syntactic entity φ. We will sometimes use such a grammar representation to describe syntax in a succinct manner.

Inductive Definitions

What is the set identified by inductive definitions like Definition 1.4 and (1.1) ? Is there a unique single minimal set that satisfies the conditions in Definition 1.4? After all sets can be incomparable with respect to the $\subseteq$ relation. And what does the grammar given in (1.1) mean? Finally, do Definition 1.4 and (1.1) identify the same set?

Let us begin by first defining the set described by (1.1) . Equation (1.1) defines the set $S = \cup_{i\in\mathbb{N}} S_i$, where the sets S_i (for $i \in \mathbb{N}$) are given as follows.

$$\begin{aligned} S_0 &= \mathsf{Prop} \\ S_{i+1} &= S_i \cup \{(\neg\varphi) \mid \varphi \in S_i\} \cup \{(\psi \vee \varphi) \mid \psi, \varphi \in S_i\} \qquad \text{for } i \geq 0 \end{aligned}$$

Note that $S = \cup_{i \in \mathbb{N}} S_i$ is an infinite union. You can think of S as the limit of the increasing sequence of sets $S_0 \cup S_1 \cup \cdots \cup S_n$. Another way to think of S is as the set of all φ that belong to *some* S_i. That is,

$$S = \{\varphi \mid \varphi \in S_i, \text{ for some } i\}.$$

Another way to interpret the sets S and S_i is as follows. S_i denotes the set of formulas that can be derived from the grammar rules within i steps, and S is the set of formulas that can be derived from the grammar rules in *some finite number* of steps.

Given the above meaning, it's natural to prove properties about the set of expressions S using induction on i. More precisely, if we want to show a property P is true about S, then we:

- Establish P to be true for every expression in S_0.
- For every $i \geq 0$, we assume that P holds for every expression in S_i and prove it holds for all expressions in S_{i+1}.

The above shows that P holds on S_i, for every $i \in \mathbb{N}$, and hence P holds for every formula in S.

Let us now consider Definition 1.4 and argue that it is well defined. Before showing that there is a unique smallest set satisfying conditions (1), (2), and (3) of Definition 1.4, let us argue that there is at least one set satisfying (1), (2), and (3). Take the set of all possible strings built over the alphabet $\mathsf{Prop} \cup \{(,), \neg, \vee\}$. This set clearly satisfies all the conditions. But why should there be a smallest set? Observe that if two sets A and B satisfy the conditions in Definition 1.4, then so does $A \cap B$. More generally, if $\{A_i\}_{i \in I}$ is a (possibly infinite) collection of sets satisfying (1), (2), and (3), then so does the set $\cap_{i \in I} A_i$. Thus, the intersection of *all* sets which satisfy the conditions, is the unique, smallest (with respect to set inclusion) set identified by Definition 1.4. Hence, it is well defined.

Let S denote the set identified by grammar in (1.1) and T the set defined in Definition 1.4. We will argue that these two sets are the same. First, let us show that S satisfies conditions (1), (2), and (3) of Definition 1.4. Observe that, by definition, for every i, $S_i \subseteq S_{i+1}$. Hence, the sets increase with index, i.e., for every $i < j$, $S_i \subseteq S_j$; this can be formally established by induction, and we leave this as an exercise. Since $\mathsf{Prop} \subseteq S_0$, this means $\mathsf{Prop} \subseteq S_i$ for every i, and therefore S satisfies condition (1). Next, consider any $\varphi, \psi \in S$. By definition of S, this means there is are $i, j \in \mathbb{N}$ such that $\varphi \in S_i$ and $\psi \in S_j$. Taking $k = \max(i, j)$, we can conclude that $\{\varphi, \psi\} \subseteq S_k$. Thus, by definition, $(\neg\varphi)$ and $(\varphi \vee \psi)$ both belong to S_{k+1}, and hence are in S. Therefore, S satisfies conditions (2) and (3) of Definition 1.4. Finally, since S satisfies all conditions of Definition 1.4 and T is the smallest set with these properties, we can conclude that $T \subseteq S$.

To prove the other inclusion (that $S \subseteq T$), we will prove that for every i, $S_i \subseteq T$ by induction. In the base case observe that $S_0 = \mathsf{Prop} \subseteq T$, since T satisfies condition (1). Assume as induction hypothesis, that for all $j \leq i$, $S_j \subseteq T$. In the induction step, we need to establish the claim that $S_{i+1} \subseteq T$. Let $\varphi \in S_{i+1}$ be an arbitrary element. By the definition of S_{i+1}, there are 3 cases to consider. If $\varphi \in S_i$ then by induction hypothesis $\varphi \in T$. If $\varphi = (\neg\psi)$ for some $\psi \in S_i$, then by induction hypothesis $\psi \in T$, and since T satisfies condition (2), $\varphi = (\neg\psi) \in T$. Finally, if $\varphi = (\psi_1 \vee \psi_2)$ for some $\psi_1, \psi_2 \in S_i$ then by induction hypothesis, $\psi_1, \psi_2 \in T$ and since T satisfies condition (3), $\varphi = (\psi_1 \vee \psi_2) \in T$. This completes the proof that $S \subseteq T$, and therefore, $S = T$.

The above argument, establishing that the smallest set satisfying some closure conditions is the same as the set of objects derived from grammar rules, can be generalized to any grammar (not just context-free grammars. In general, if one defines a set as the smallest set S that satisfies conditions of the form "if these elements belong to S than these other elements must belong to S", then the smallest set is well-defined. But if you also have conditions saying "if these elements belong to S, these elements should *not* belong to S," then the "smallest" set may not be a well-defined.

Inductive or *recursive* definitions are ubiquitous in computer science, and reasoning about such definitions is naturally done using some form of induction. In fact, the prevalence of induction in computer science is because of the ubiquity of recursive definitions. Here are some other examples of recursive definitions.

- Consider the operational semantics of a program. The set of states/configurations that the program can reach is best defined recursively. It is the smallest set R of states such that (a) R contains the initial states of the program, and (b) if a state s is in R and the program can transition in one step from s to s', then $s' \in R$. Because of the recursive nature of this definition, reasoning about programs often involves induction. For example, to show that a program does not throw an exception, one needs to prove that any reachable state is one that does not throw an exception. This is typically established using induction.
- Consider *lists* in programs. The `cons` operator constructs a new list by adding an element to the head of another list. Lists over the elements E can then be defined as the smallest set L such that (a) L contains `nil`, the empty list, and (b) if $\ell \in L$ and $e \in E$, then $\mathtt{cons}(e, \ell)$ is in L as well.
- The set of *natural numbers* can also be defined recursively. Let us denote by succ, the successor function [2]. Then the set of natural numbers is the smallest set $\mathbb{N}$ such that (a) $\mathbb{N}$ contains 0, and (b) for every $n \in \mathbb{N}$, $\mathsf{succ}(n)$ belongs to $\mathbb{N}$.

Other logical operators and Operator Precedence

Conjunction and *implication* are logical operators that arise quite often when expressing properties. These operators can be defined in terms of $\neg$ and $\vee$. Let φ and ψ be wffs. $(\varphi \wedge \psi)$ (read "φ *and* ψ") denotes the formula $(\neg((\neg\varphi) \vee (\neg\psi)))$. And $(\varphi \rightarrow \psi)$ (read "φ *implies* ψ") denotes the formula $((\neg\varphi) \vee \psi)$. Another useful wff is $\top$ (read "true"); it denotes the formula $(p \vee (\neg p))$, where p is (any) proposition. Finally the wff $\bot$ (read "false") is the formula $(\neg\top)$.

[2] Intuitively, given a number n, $\mathsf{succ}(n)$ is the next number, namely, $n + 1$. However, this interpretation is only in our minds. succ is just some function.

Writing formulas strictly according to the syntax presented is cumbersome because of many parentheses and subscripts. Therefore, we will make the following notational simplifications.

- The outermost parentheses will be dropped. Thus we will write $p_3 \vee (p_2 \vee p_1)$ instead of $(p_3 \vee (p_2 \vee p_1))$
- We will sometimes omit subscripts of propositions. Thus we will write p instead of p_1, or q instead of p_2, r instead of p_3, or s instead of p_4, and so on.
- The following precedence of operators will be assumed: $\neg$, $\wedge$, $\vee$, $\rightarrow$. Thus $\neg p \wedge q \rightarrow r$ will mean $(((\neg p) \wedge q) \rightarrow r)$.

Definition 1.4 for wffs in propositional logic has the nice property that the structure of a formula can be interpreted in a unique way. There is no ambiguity in its interpretation. For example, if $\neg p_1 \vee p_2$ were a wff, then it is unclear whether we mean the formula $\varphi = ((\neg p_1) \vee p_2)$ or $\psi = (\neg(p_1 \vee p_2))$ [3] — in φ $\vee$ is the topmost operator, while in ψ $\neg$ is the topmost operator. Our syntax does not have such issues. This will be exploited often in inductive definitions and in algorithms. This observation can be proved by structural induction, but we skip its proof.

Theorem 1.5 (Unique Readability)

Any wff can be uniquely read, i.e., it has a unique topmost logical operator and well defined immediate sub-formulas.

1.2 Semantics

We will now provide a meaning or *semantics* to the formulas. Our definition will follow the inductive definition of the syntax, just like in Example 1.3. The semantics of formulas in a logic, are typically defined with respect to a *model*, which identifies a "world" in which certain facts are true. In the case of propositional logic, this world or model is a *truth valuation* or *assignment* that assigns a truth value (true/false) to every proposition. The *truth value truth* will be denoted by T, and the truth value *falsity* will be denoted by F.

Definition 1.6 A *(truth) valuation* or *assignment* is a function v that assigns truth values to each of the propositions, i.e., $\mathsf{v} : \mathsf{Prop} \rightarrow \{\mathsf{T}, \mathsf{F}\}$.

The value of a proposition p under valuation v is given by $\mathsf{v}(p)$.

We will define the semantics through a *satisfaction relation*, which is a binary relation $\models$ between valuations and formulas. The statement $\mathsf{v} \models \varphi$ should be read as "v satisfies φ" or "φ is true in v" or "v is a model of φ". It is defined inductively following the syntax of formulas. In the definition below, we say $\mathsf{v} \not\models \varphi$ when $\mathsf{v} \models \varphi$ does not hold.

Definition 1.7 For a valuation v and wff φ, the satisfaction relation, $\mathsf{v} \models \varphi$, is defined inductively based on the structure of φ as follows.

[3] For the formulas here, we are not using the precedence rules given before.

- $\mathsf{v} \models p$ if and only if $\mathsf{v}(p) = \mathsf{T}$.
- $\mathsf{v} \models (\neg\varphi)$ if and only if $\mathsf{v} \not\models \varphi$.
- $\mathsf{v} \models (\varphi \vee \psi)$ if either $\mathsf{v} \models \varphi$ or $\mathsf{v} \models \psi$.

Example 1.8 Let us look at a couple of examples to see how the inductive definition of the satisfaction relation can be applied. Consider the formula $\varphi = \neg(\neg p \vee \neg q) \vee (\neg p \vee \neg q)$. Recall with respect to the notational simplifications we identified, φ is the formula $((\neg((\neg p) \vee (\neg q))) \vee ((\neg p) \vee (\neg q)))$. Consider the valuation v_1 that sets all propositions to T. Now $\mathsf{v}_1 \models \varphi$ can be seen from the following observations.

$\mathsf{v}_1 \models p$	because $\mathsf{v}_1(p) = \mathsf{T}$
$\mathsf{v}_1 \not\models \neg p$	semantics of $\neg$
$\mathsf{v}_1 \models q$	because $\mathsf{v}_1(q) = \mathsf{T}$
$\mathsf{v}_1 \not\models \neg q$	semantics of $\neg$
$\mathsf{v}_1 \not\models \neg p \vee \neg q$	semantics of $\vee$
$\mathsf{v}_1 \models \neg(\neg p \vee \neg q)$	semantics of $\neg$
$\mathsf{v}_1 \models \neg(\neg p \vee \neg q) \vee (\neg p \vee \neg q)$	semantics of $\vee$

Consider v_2 that assigns all propositions to F. Once again $\mathsf{v}_2 \models \varphi$. The reasoning behind this observation is as follows.

$\mathsf{v}_2 \not\models p$	because $\mathsf{v}_2(p) = \mathsf{F}$
$\mathsf{v}_2 \models \neg p$	semantics of $\neg$
$\mathsf{v}_2 \models \neg p \vee \neg q$	semantics of $\vee$
$\mathsf{v}_2 \models \neg(\neg p \vee \neg q) \vee (\neg p \vee \neg q)$	semantics of $\vee$

The semantics in Definition 1.7 defines a satisfaction relation between valuations and formulas. However, one could defined the semantics of propositional logic differently, by considering the formula as a "program" or "circuit" that computes a truth value based on the assignment. This approach is captured by the following definition of the *value* of a wff under a valuation.

Definition 1.9 The *value of a wff φ under valuation* v, denoted by $\mathsf{v}[\![\varphi]\!]$, is inductively defined as follows.

$$\mathsf{v}[\![p]\!] = \mathsf{v}(p)$$
$$\mathsf{v}[\![(\neg\varphi)]\!] = \begin{cases} \mathsf{F} \text{ if } \mathsf{v}[\![\varphi]\!] = \mathsf{T} \\ \mathsf{T} \text{ if } \mathsf{v}[\![\varphi]\!] = \mathsf{F} \end{cases}$$
$$\mathsf{v}[\![\varphi \vee \psi]\!] = \begin{cases} \mathsf{F} \text{ if } \mathsf{v}[\![\varphi]\!] = \mathsf{v}[\![\psi]\!] = \mathsf{F} \\ \mathsf{T} \text{ otherwise} \end{cases}$$

Example 1.10 Let us consider $\varphi = \neg(\neg p \vee \neg q) \vee (\neg p \vee \neg q)$ and v_1 which assigns all propositions to T. $\mathsf{v}_1[\![\varphi]\!]$ can be computed as follows.

$\mathsf{v}_1[\![p]\!] = \mathsf{T}$	because $\mathsf{v}_1(p) = \mathsf{T}$
$\mathsf{v}_1[\![\neg p]\!] = \mathsf{F}$	semantics of $\neg$
$\mathsf{v}_1[\![q]\!] = \mathsf{T}$	because $\mathsf{v}_1(q) = \mathsf{T}$
$\mathsf{v}_1[\![\neg q]\!] = \mathsf{F}$	semantics of $\neg$
$\mathsf{v}_1[\![\neg p \vee \neg q]\!] = \mathsf{F}$	semantics of $\vee$
$\mathsf{v}_1[\![\neg(\neg p \vee \neg q)]\!] = \mathsf{T}$	semantics of $\neg$
$\mathsf{v}_1[\![\neg(\neg p \vee \neg q) \vee (\neg p \vee \neg q)]\!] = \mathsf{T}$	semantics of $\vee$

Definitions 1.7 and 1.9 are both equivalent in some sense. This is captured by the following theorem.

Theorem 1.11 *For any truth valuation* v *and wff* φ, $\mathsf{v} \models \varphi$ *if and only if* $\mathsf{v}[\![\varphi]\!] = \mathsf{T}$

The proof of Theorem 1.11 is by structural induction on the formula φ. It is left as an exercise for the reader.

It is convenient to associate with every wff the set of truth valuations under which the formula holds.

Definition 1.12 The *models* of wff φ is the set of valuations that *satisfy* φ. More precisely,

$$[\![\varphi]\!] = \{\mathsf{v} \mid \mathsf{v} \models \varphi\}.$$

Observe that as per the definition, $[\![\bot]\!] = \emptyset$.

The *relevance lemma* says that whether φ holds under a valuation depends only on how the valuation maps the propositions that *syntactically occur* in the formula. This is intuitively obvious; surely, whether $(p \wedge q) \vee r$ holds is independent of whether the proposition s is mapped to true/false.For a wff φ, the set of propositions appearing in φ, denoted $\mathsf{occ}(\varphi)$, is inductively defined as follows.

$$\begin{aligned} &\mathsf{occ}(p) = \{p\} \\ &\mathsf{occ}(\neg\varphi) = \mathsf{occ}(\varphi) \\ &\mathsf{occ}(\varphi \vee \psi) = \mathsf{occ}(\varphi) \cup \mathsf{occ}(\psi) \end{aligned}$$

The relevance lemma is then as follows.

Lemma 1.13 (Relevance Lemma)

Let v_1 *and* v_2 *be truth valuations such that for all* $p \in \mathsf{occ}(\varphi)$, *we have* $\mathsf{v}_1(p) = \mathsf{v}_2(p)$, *i.e.,* v_1 *and* v_2 *agree on the truth values assigned to all propositions in* $\mathsf{occ}(\varphi)$. *Then* $\mathsf{v}_1 \models \varphi$ *if and only if* $\mathsf{v}_2 \models \varphi$.

Proof By structural induction on φ.

Base Case $\varphi = p$ Observe that, $\mathsf{v}_1 \models \varphi$ iff $\mathsf{v}_1(p) = \mathsf{T} = \mathsf{v}_2(p)$ iff $\mathsf{v}_2 \models \varphi$.

Induction Step $\varphi = (\neg\psi)$ Since $\mathsf{occ}(\psi) \subseteq \mathsf{occ}(\varphi)$, we have by induction hypothesis, $\mathsf{v}_1 \models \psi$ iff $\mathsf{v}_2 \models \psi$. Therefore, by the semantics of $\neg$, $\mathsf{v}_1 \models \varphi$ iff $\mathsf{v}_2 \models \varphi$.

Induction Step $\varphi = (\psi_1 \vee \psi_2)$ Since $\mathsf{prop}(\psi_i) \subseteq \mathsf{prop}(\varphi)$ (for $i \in \{1, 2\}$), we have by induction hypothesis, $\mathsf{v}_1 \models \psi_i$ iff $\mathsf{v}_2 \models \psi_i$. Therefore, by the semantics of $\vee$, $\mathsf{v}_1 \models \varphi$ iff $\mathsf{v}_2 \models \varphi$. □

Lemma 1.13 implies that, to determine if a formula holds in a valuation, we only need to consider the assignment to the finitely many propositions occurring in the formula. Thus, instead of thinking of valuations as assigning truth values to all (infinitely many) propositions, we can think of them as functions with a finite domain.

1.3 Satisfiability and Validity

Two formulas that are syntactically different, could however, be "semantically equivalent". But what do we mean by semantic equivalence? Intuitively, this is when the truth value of each formula in every valuation is the same.

Definition 1.14 (Logical Equivalence)

A wff φ is said to be *logically equivalent* to ψ iff any of the following equivalent conditions hold.

- for every valuation v, $\mathsf{v} \models \varphi$ iff $\mathsf{v} \models \psi$,
- for every valuation v, $\mathsf{v}[\![\varphi]\!] = \mathsf{v}[\![\psi]\!]$,
- $[\![\varphi]\!] = [\![\psi]\!]$.

We denote this by $\varphi \equiv \psi$.

Let us consider an example to see how we may reason about two formulas being logically equivalent.

Example 1.15 Consider the wffs $\varphi_1 = p \wedge (q \vee r)$ and $\varphi_2 = (p \wedge q) \vee (p \wedge r)$, where p, q and r are propositions. Though φ_1 and φ_2 are syntactically different, they are semantically equivalent. To prove that $\varphi_1 \equiv \varphi_2$, we need to show that they two formulas evaluate to the same truth value under every valuation. One convenient way to organize such a proof is as *truth table*, where different cases in the case-by-case analysis correspond to different rows. Each row of the truth table corresponds to a (infinite) collection of valuations based on the value assigned to propositions p, q and r; the columns correspond to the value of different (sub)-formulas under each valuation in this collection. For example, a truth table reasoning for φ_1 and φ_2 will look as follows.

p	q	r	$q \vee r$	φ_1	$p \wedge q$	$p \wedge r$	φ_2
F	F	F	F	F	F	F	F
F	F	T	T	F	F	F	F
F	T	F	T	F	F	F	F
F	T	T	T	F	F	F	F
T	F	F	F	F	F	F	F
T	F	T	T	T	F	T	T
T	T	F	T	T	T	F	T
T	T	T	T	T	T	T	T

Notice that since the columns corresponding to φ_1 and φ_2 are identical in every row, and every valuation corresponds to some row in the table, it follows that φ_1 and φ_2 are logically equivalent.

Let us now consider $\varphi_1' = \psi_1 \wedge (\psi_2 \vee \psi_3)$ and $\varphi_2' = (\psi_1 \wedge \psi_2) \vee (\psi_1 \wedge \psi_3)$, where ψ_1, ψ_2 and ψ_3 are arbitrary wffs. Once again φ_1' and φ_2' are logically equivalent, no matter what ψ_1, ψ_2 and ψ_3 are. The reasoning is essentially the same as above. The rows of the truth table now classify valuations based on the value of formulas ψ_1, ψ_2 and ψ_3 under them.

ψ_1	ψ_2	ψ_3	$\psi_2 \vee \psi_3$	φ_1	$\psi_1 \wedge \psi_2$	$\psi_1 \wedge \psi_3$	φ_2
F	F	F	F	F	F	F	F
F	F	T	T	F	F	F	F
F	T	F	T	F	F	F	F
F	T	T	T	F	F	F	F
T	F	F	F	F	F	F	F
T	F	T	T	T	F	T	T
T	T	F	T	T	T	F	T
T	T	T	T	T	T	T	T

Truth table based reasoning, as carried out in Example 1.15, is a very convenient way to organize proofs of propositional logic. We will often use it. Example 1.15 highlights another important observation. Let φ and ψ be logically equivalent formulas. Let φ' and ψ' be formulas obtained by substituting propositions occurring in φ and ψ by arbitrary formulas. Then $\varphi' \equiv \psi'$.

Definition 1.16 (Logical Consequence)

Let Γ be a (possibly infinite) set of formulas and let φ be a wff. We say that φ is a *logical consequence* of Γ (denoted $\Gamma \models \varphi$) iff for every valuation v, if for every $\psi \in \Gamma$, $\mathsf{v} \models \psi$ then $\mathsf{v} \models \varphi$. In other words, any model that satisfies every formula in Γ also satisfies φ.

We could equivalently have defined it as $\Gamma \models \varphi$ iff $\bigcap_{\psi\in\Gamma}[\![\psi]\!] \subseteq [\![\varphi]\!]$.

Example 1.17 Consider the set $\Gamma = \{\psi_1 \rightarrow \psi_2, \psi_2 \rightarrow \psi_1, \psi_1 \vee \psi_2\}$, where ψ_1 and ψ_2 are arbitrary formulas. We will show that $\Gamma \models \psi_1$. Once again, we will use a truth table to classify valuations into row based on the value that ψ_1 and ψ_2 evaluate to. Such a truth table looks as follows.

ψ_1	ψ_2	$\psi_1 \rightarrow \psi_2$	$\psi_2 \rightarrow \psi_1$	$\psi_1 \vee \psi_2$
F	F	T	T	F
F	T	T	F	T
T	F	F	T	T
T	T	T	T	T

Notice that there is only one row where columns 3, 4, and 5 are all T; this corresponds the valuations where ψ_1 and ψ_2 evaluate to T, and under every such valuation, all formulas in Γ are satisfied. In this row, since ψ_1 also evaluates to T, we have that $\Gamma \models \psi_1$.

It is worth observing one special case of Definition 1.16 — when $\Gamma = \emptyset$. In this case, every valuation satisfies every formula in Γ (vaccuously, since there are none to satisfy). Therefore, if $\emptyset \models \varphi$, then every truth assignment satisfies φ. Such formulas are called *tautologies*, and they represent universal truths that hold in every model/world/assignment.

Definition 1.18 (Tautologies)

A wff φ is a *tautology* or is *valid* if for every valuation v, $\mathsf{v} \models \varphi$. In other words, $\emptyset \models \varphi$. We will denote this simply as $\models \varphi$.

Example 1.19 We will show $\varphi = \psi_1 \to (\psi_2 \to \psi_1)$ is a tautology, no matter what formulas ψ_1 and ψ_2 are. The proof is once again organized as truth table, and we show that in all rows the formula φ evaluates to T.

ψ_1	ψ_2	$\psi_2 \to \psi_1$	$\psi_1 \to (\psi_2 \to \psi_1)$
F	F	T	T
F	T	F	T
T	F	T	T
T	T	T	T

The last important notion we would like to introduce is that of *satisfiability*.

Definition 1.20 (Satisfiability)

A formula φ is satisfiable if there is some valuation v such that $\mathsf{v} \models \varphi$. In other words, $[\![\varphi]\!] \neq \emptyset$. If a formula is not satisfiable, we say it is *unsatisfiable*.

Example 1.21 $\varphi = (p \lor q) \land (\neg p \lor \neg q)$ is satisfiable because the valuation v that maps p to T and q to F satisfies φ, i.e., $\mathsf{v} \models \varphi$.

Based on Definitions 1.18 and 1.20, it is easy to see that there is a close connection between satisfiability and validity.

Proposition 1.22 *A wff φ is valid if and only if $\neg\varphi$ is unsatisfiable.*

Proof Let v be any valuation. If φ is valid, we know that $\mathsf{v} \models \varphi$. Therefore, by the semantics of $\neg$, we have $\mathsf{v} \not\models \neg\varphi$. Thus $\neg\varphi$ is unsatisfiable. Conversely, if $\neg\varphi$ is unsatisfiable, then $\mathsf{v} \not\models \neg\varphi$. Again, by the semantics of $\neg$, $\mathsf{v} \models \varphi$. Thus, φ is valid. □

We conclude this section by considering two fundamental computational problems — *satisfiability* and *validity*.

Satisfiability Given a formula φ, determine if φ is satisfiable.
Validity Given a formula φ, determine if φ is a tautology.

The satisfiability and validity problems have very simple algorithms to solve them. To check if φ, over propositions $\{p_1, \ldots p_n\}$, is satisfiable (is a tautology), compute $\mathsf{v}[\![\varphi]\!]$ for every truth assignment v to the propositions $\{p_1, \ldots p_n\}$. The running time for this algorithm is $O(2^n)$. One of the most important open questions in computer science is whether this is the best algorithm for these problems. The following theorem by Cook and Levin, supports the belief that this exponential algorithm is unlikely to be improved in the worst case.

Theorem 1.23 (Cook-Levin)

The satisfiability problem for propositional logic is NP*-complete.*

Proof Any proof showing a problem to be NP-complete has two parts. First is an argument that the problem belongs to NP and the second that it is hard.

Membership in NP**.** Given a formula φ, the NP-algorithm to check satisfiability is as follows — Guess a truth assignment v, evaluate φ on the truth assignment, and accept if $\mathsf{v}[\![\varphi]\!] = \mathsf{T}$; otherwise reject. Guessing (nondeterministically) a truth assignment takes time which is linear in the number of propositions in φ, which is linear in the size of φ, and computing $\mathsf{v}[\![\varphi]\!]$ also takes time that is linear in the size of φ, where the evaluation algorithm computes the value in a "bottom-up" fashion. Thus, the total running time is polynomial.

NP**-hardness.** Consider $A \in \mathsf{NP}$. Let $M = (Q, \Sigma, \Gamma, \delta, q_0, q_{\mathsf{acc}}, q_{\mathsf{rej}}, \sqcup, \rhd)$ be a nondeterministic TM recognizing A in time n^ℓ, where Q is the set of control states, Σ is the input alphabet, Γ is the tape alphabet, δ is the transition function, q_0 is the initial state, q_{acc} is the unique accepting state, q_{rej} is the unique rejecting state, $\sqcup$ is the blank symbol, and $\rhd$ is the left end marker symbol that appears on the leftmost cell of each tape. Without loss of generality, we assume that q_{acc} and q_{rej} are the only halting states of M. We will also assume that M has a read only input tape, and a *single* read/write work tape.

For an input x, we will construct (in polynomial time) a formula $f_M(x)$ such that M accepts x (i.e., $x \in A$) iff $f_M(x)$ is satisfiable. $f_M(x)$ will encode constraints on a computation of M on x such that a satisfying assignment to $f_M(x)$ will describe "how M accepts x". That is, $f_M(x)$ will encode that

- M starts in the initial configuration with input x,
- Each configuration follows from the previous one in accordance with the transition function of M,
- The accepting state is reached in the last step.

Let us formalize this intuition by giving a precise construction. We begin by identifying the set of propositions we will use, and their informal interpretation.

Propositional Variables. The propositions of $f_M(x)$ will be as follows.

Name	Meaning if set to T	Total Number
$\mathsf{InpSymb}(b, p)$	Input tape stores b at position p	$O(\lvert x\rvert)$
$\mathsf{TapeSymb}(b, p, i)$	Work tape stores b in cell p at time i	$O(\lvert x\rvert^{2\ell})$
$\mathsf{InpHd}(h, i)$	Input head in cell h at time i	$O(\lvert x\rvert \cdot \lvert x\rvert^{\ell})$
$\mathsf{TapeHd}(h, i)$	Work tape's head in cell h at time i	$O(\lvert x\rvert^{2\ell})$
$\mathsf{State}(q, i)$	State is q at time i	$O(\lvert x\rvert^{\ell})$

Abbreviations. In order to define $f_M(x)$, the following abbreviations will be useful.

- $\bigwedge_{k=1}^{m} X_k$ means $X_1 \wedge X_2 \wedge \cdots \wedge X_m$

- $\nabla(X_1, X_2, \dots X_m)$ will denote a formula that is satisfiable iff exactly one of $X_1, \dots X_m$ is set to true. In other words,

$$\nabla(X_1, X_2, \dots X_m) = (X_1 \vee X_2 \vee \cdots \vee X_m) \wedge \bigwedge_{k \neq l} (\neg X_k \vee \neg X_l)$$

Overall Reduction. The overall form of $f_M(x)$ will be as follows.

$$f_M(x) = \varphi_{\text{initial}} \wedge \varphi_{\text{consistent}} \wedge \varphi_{\text{transition}} \wedge \varphi_{\text{accept}}$$

where

- φ_{initial} says that “configuration at time 0 is the initial configuration with input x”
- $\varphi_{\text{consistent}}$ says that “at each time, truth values to variables encode a valid configuration”
- $\varphi_{\text{transition}}$ says that “configuration at each time follows from the previous one by taking a transition”
- φ_{accept} says that “the last configuration is an accepting configuration”

We now outline what each of the above formulas is.

Initial Conditions Let $x = a_1 a_2 \cdots a_n$

$$\begin{aligned}
\varphi_{\text{initial}} = {} & \mathsf{State}(q_0, 0) \\
& \qquad \text{“At time 0, state is } q_0\text{”} \\
& \wedge \mathsf{InpSymb}(\triangleright, 0) \wedge \mathsf{TapeSymb}(\triangleright, 0, 0) \\
& \qquad \text{“Leftmost cells contain } \triangleright\text{”} \\
& \bigwedge\nolimits_{p=1}^{n} \mathsf{InpSymb}(a_p, p) \\
& \qquad \text{“At time 0, cells 1 through } n \text{ hold } x\text{”} \\
& \bigwedge\nolimits_{p=1}^{n^\ell} \mathsf{TapeSymb}(\sqcup, p, 0) \\
& \qquad \text{“At time 0, all work tape cells are blank”} \\
& \mathsf{InpHd}(0, 0) \wedge \mathsf{TapeHd}(0, 0) \\
& \text{“At time 0, all heads at the leftmost position”}
\end{aligned}$$

Consistency Assume that the tape alphabet is $\Gamma = \{b_1, b_2, \dots b_t\}$ and the set of states is $Q = \{q_0, q_1, \dots q_m\}$.

$$\begin{aligned}
\varphi_{\text{consistent}} = & \bigwedge_{i=0}^{n^\ell} \nabla(\mathsf{State}(q_0, i), \dots \mathsf{State}(q_m, i)) \\
& \qquad \text{“At any time } i\text{, state is unique”} \\
& \bigwedge_{i=0}^{n^\ell} \mathsf{TapeSymb}(\triangleright, 0, i) \\
& \qquad \text{“At any time, leftmost cell contains } \triangleright\text{”} \\
& \bigwedge_{i=0}^{n^\ell} \bigwedge_{p=0}^{n^\ell} \nabla(\mathsf{TapeSymb}(b_1, p, i), \dots \mathsf{TapeSymb}(b_t, p, i)) \\
& \qquad \text{“work tape cells contain unique symbols”} \\
& \bigwedge_{i=0}^{n^\ell} \nabla(\mathsf{InpHd}(0, i), \dots \mathsf{InpHd}(n, i)) \\
& \qquad \text{“At any time, input head is in one cell”} \\
& \bigwedge_{i=0}^{n^\ell} \nabla(\mathsf{TapeHd}(0, i), \dots \mathsf{TapeHd}(n^\ell, i)) \\
& \qquad \text{“At any time, work tape head is in one cell”}
\end{aligned}$$

Transition Consistency Consider a non-halting state q (i.e., $q \notin \{q_{\mathsf{acc}}, q_{\mathsf{rej}}\}$), input symbol c_{in} and tape symbol c_w. Let the transition at state q, when reading these symbols be given by

$$\delta(q, c_{in}, c_w) = \{(q^{(1)}, d_{in}^{(1)}, c_w^{(1)}, d_w^{(1)}), \dots (q^{(s)}, d_{in}^{(s)}, c_w^{(s)}, d_w^{(s)})\}.$$

Here $(q^{(i)}, d_{in}^{(i)}, c_w^{(i)}, d_w^{(i)}) \in \delta(q, c_{in}, c_w)$ means that if M is in state q and reads c_{in} on the input tape and c_w on the work tape, then one possible transition is to state $q^{(i)}$, moving the input head in direction $d_{in}^{(i)}$, writing $c_w^{(i)}$ on the work tape, and moving the work tape head in direction $d_w^{(i)}$. Direction -1 denotes moving the head *left* and $+1$ denotes moving the head *right*. We will first define a formula $\Delta_{q,c_{in},c_w}^{i,p_{in},p_w}$ that says that at time i if the state is q and the symbol read on the input tape is c_{in} and on the work tape is c_w at positions p_{in}, p_w, respectively, then at time $i + 1$ the state, symbols written and new head position is one of the tuples described by the δ function.

$$\begin{aligned}
\Delta_{q,c_{in},c_w}^{i,p_{in},p_w} = & (\mathsf{State}(q, i) \wedge \mathsf{InpHd}(p_{in}, i) \wedge \mathsf{InpSymb}(c_{in}, p_{in}, i) \wedge \\
& \mathsf{TapeHd}(p_w, i) \wedge \mathsf{TapeSymb}(c_w, p_w, i)) \to \\
& \nabla(\mathrm{Ch}_{i,p_{in},p_w}^1, \mathrm{Ch}_{i,p_{in},p_w}^2, \dots \mathrm{Ch}_{i,p_{in},p_w}^s)
\end{aligned}$$

where

$$\begin{aligned}
\mathrm{Ch}_{i,p_{in},p_w}^t = & \mathsf{State}(q^{(t)}, i + 1) \wedge \mathsf{InpHd}(p_{in} + d_{in}^{(t)}, i + 1) \wedge \\
& \mathsf{TapeSymb}(c_w^{(t)}, p_w, i + 1) \wedge \mathsf{TapeHd}(p_w + d_w^{(t)}, i + 1)
\end{aligned}$$

For a halting state q (i.e., $q \in \{q_{\mathsf{acc}}, q_{\mathsf{rej}}\}$), we define $\Delta_{q,c_{in},c_w}^{i,p_{in},p_w}$ as saying that the state, symbols, and head positions don't change. In other words,

$$\begin{aligned}
\Delta_{q,c_{in},c_w}^{i,p_{in},p_w} = & (\mathsf{State}(q, i) \wedge \mathsf{InpHd}(p_{in}, i) \wedge \mathsf{InpSymb}(c_{in}, p_{in}) \wedge \\
& \mathsf{TapeHd}(p_w, i) \wedge \mathsf{TapeSymb}(c_w, p_w, i)) \to \\
& (\mathsf{State}(q, i + 1) \wedge \mathsf{InpHd}(p_{in}, i + 1) \\
& \wedge \mathsf{TapeSymb}(c_w, p_w, i + 1) \wedge \mathsf{TapeHd}(p_w, i + 1))
\end{aligned}$$

when q is a halting state.

Now Transition Consistency itself can be defined as follows.

$$\varphi_{\text{transition}} = \bigwedge_{i=0}^{n^\ell} \bigwedge_{p_{in}=0}^{n} \bigwedge_{p_w=0}^{n^\ell} \{$$

$$\bigwedge_{b \neq c} \neg \mathsf{TapeHd}(p_w, i) \rightarrow$$
$$\neg(\mathsf{TapeSymb}(b, p_w, i) \wedge \mathsf{TapeSymb}(c, p_w, i+1))$$
"If head is not in some position, then symbol does not change"

$$\bigwedge_{q=q_0}^{q_m} \bigwedge_{c_{in}=b_1}^{b_t} \bigwedge_{c_w=b_1}^{b_t} \Delta_{q,c_{in},c_w}^{i,p_{in},p_w} \}$$
"If head is in some position, then a transition is taken"

Acceptance

$$\varphi_{\text{accept}} = \mathsf{State}(q_{\mathsf{acc}}, n^\ell)$$

We can argue that M accepts x if and only if $f_M(x)$ is satisfiable. Further $f_M(x)$ can be constructed in time that is polynomial in the size of x; the size of M also plays a role but that is fixed. □

The Cook-Levin Theorem (Theorem 1.23) is an important result in computer science. It was the first result establishing the intractibility of a problem. Moreover, it also implies the intractibility of the validity problem. This is because there is a formula φ is valid if and only if $\neg\varphi$ is unsatisfiable.

Proposition 1.24 *A formula φ is valid if and only if $\neg\varphi$ is unsatisfiable.*

Proof The proposition can be established by the following sequence of observations. φ is valid iff for every valuation v, $\mathsf{v}[\![\varphi]\!] = \mathsf{T}$ (definition of validity) iff for every valuation v, $\mathsf{v}[\![\neg\varphi]\!] = \mathsf{F}$ (from the semantics of $\neg$) iff $\neg\varphi$ is unsatisfiable (definition of unsatisfiability). □

Using Proposition 1.24 we establish the coNP-hardness of validity.

Theorem 1.25 *The validity problem for propositional logic is* coNP*-complete.*

Proof Observe that there is a simple NP algorithm to check that a formula φ is *not* valid — Guess a valuation v, and check that $\mathsf{v}[\![\varphi]\!] = \mathsf{F}$. Since checking non-validity has a NP algorithm, it means that the validity problem has a coNP algorithm, and is therefore, in coNP.

To prove the coNP-hardness of the validity problem, we make the following observations. First, for any two problems A and B, if $A \leq_{\mathsf{P}} B$ then $\overline{A} \leq_{\mathsf{P}} \overline{B}$. Thus, from the NP-hardness of the satisfiability problem (Theorem 1.23), we can conclude that the problem of checking if a formula is *unsatisfiable* is coNP-hard. Since unsatisfiability is coNP-hard, it follows that validity is also coNP-hard based on the observation in Proposition 1.24. □

1.4 Compactness Theorem

The compactness theorem is an important property about propositional logic. In this section, we will look at a couple of different proofs of this theorem.

A (finite or infinite) set of formulas Γ is *satisfiable* if there is a valuation v such that for every $\varphi \in \Gamma$, $\mathsf{v} \models \varphi$ (or $\mathsf{v}[\![\varphi]\!] = \mathsf{T}$); we will denote this by $\mathsf{v} \models \Gamma$. A set of formulas Γ is *finitely satisfiable* if every finite subset Γ_0 of Γ is satisfiable. These two notions, satisfiability and finite satisfiability, are equivalent — this is the content of the *compactness theorem*.

Theorem 1.26 (Compactness)
A set of formulas Γ is satisfiable if and only if Γ is finitely satisfiable.

Observe that if Γ is satisfiable then the satisfying assignment (say v) also satisfies every subset of Γ and therefore also every finite subset of Γ. Thus if Γ is satisfiable then it is also finitely satisfiable. The challenge is, therefore, in proving the converse — that finite satisfiability implies satisfiability. If Γ is a finite set, then clearly finite satisfiability implies satisfiability because Γ itself is a finite subset of Γ. So the interesting case is when Γ is infinite. We will provide a couple of very different proofs for this result.

Before moving on to present our proofs for Theorem 1.26, we highlight an important consequence of the theorem.

Corollary 1.27 *Let Γ be a (possibly infinite) set of formulas and let φ be a formula. If $\Gamma \models \varphi$ then there is a finite subset $\Delta \subseteq \Gamma$ such that $\Delta \models \varphi$.*

Proof Let $\Gamma \models \varphi$. Then $\Gamma \cup \{\neg\varphi\}$ is unsatisfiable. By Theorem 1.26, there is a finite subset $\Delta' \subseteq \Gamma \cup \{\neg\varphi\}$ that is unsatisfiable. Taking $\Delta = \Delta' \setminus \{\neg\varphi\}$, we observe that $\Delta \subseteq \Gamma$ and $\Delta \cup \{\neg\varphi\}$ is unsatisfiable. Thus, $\Delta \models \varphi$. □

It is useful to note that the above argument works even when $\neg\varphi \notin \Delta'$.

1.4.1 Compactness using König's Lemma

We present a simple and elegant proof of the compactness theorem that uses König's Lemma. This proof approach works only for propositional logic, and does not extend to first order logic. Let us begin by recalling König's lemma for binary trees.

A binary tree is said to have paths of *arbitrary length* if for each natural number n, there is a path in the tree whose length is $\geq n$. An infinite path in the binary tree is an infinite sequence of vertices of the tree such that successive vertices in the sequence are connected by an edge. Observe that if a binary tree has an infinite path then it also has paths of arbitrary length. This is because for every n, the prefix of the infinite path with $n + 1$ vertices, is a path in the binary tree of length n. König's Lemma says that the converse of this is also true.

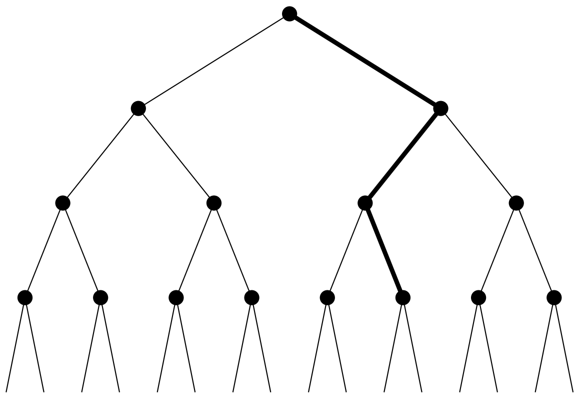

Fig. 1.2 The bold path in the tree, corresponds to the (partial) assignment $\mathsf{v}(p_0) = \mathsf{T}, \mathsf{v}(p_1) = \mathsf{F}, \mathsf{v}(p_2) = \mathsf{T}$.

Lemma 1.28 (König)

A binary tree with paths of arbitrary length has an infinite path.

Proof Suppose u is a vertex that does not have paths of arbitrary length starting from it, then by definition, there must be a number m such that all paths starting from u are of length at most m. Now, if a vertex u has the property that none of its children v have paths of arbitrary length starting from them, then u also cannot have paths of arbitrary length starting from it. The contrapositive of this statement is that if u is vertex with paths of arbitrary length starting from it, then at least one of its children v also has paths of arbitrary length.

Suppose a binary tree has paths of arbitrary length. Then the root is a vertex that has paths of arbitrary length starting from it. The infinite path is given by $v_0, v_1, \ldots$ where v_0 is the root. v_{i+1} is the left child of v_i, if the left child has paths of arbitrary length, and v_{i+1} is the right child of v_i otherwise. □

Let us fix the set of propositions in our logic to be $\mathsf{Prop} = \{p_i \mid i \in \mathbb{N}\}$. Truth assignments to Prop can be thought of as (infinite) paths in the complete, infinite binary tree — vertices at level i correspond to proposition p_i and if the path takes the left child at level i, then it corresponds to the assignment setting p_i to 0; otherwise it sets p_i to 1. Finite paths in this tree, correspond to partial assignments. So a path of length i corresponds to an assignment that sets values to propositions $\{p_0, p_1, \ldots p_{i-1}\}$. For example, in Fig. 1.2 , the bold path corresponds to the (partial) assignment v that sets $\mathsf{v}(p_0) = \mathsf{T}$, $\mathsf{v}(p_1) = \mathsf{F}$, and $\mathsf{v}(p_2) = \mathsf{T}$. Recall that, whether a formula φ holds in an assignment, depends only on the truth values assigned to the propositions that appear in φ (Lemma 1.13). Thus, partial assignments can determine the truth of formulas that only mention the propositions that have been assigned. For example, the partial assignment indicated by the bold path in Fig. 1.2 can determine the truth of any formula that only mentions p_0, p_1, and p_2.

Proof (Of Theorem 1.26) Let Γ be a set of formulas that is finitely satisfiable. Logical equivalence ($\equiv$) partitions Γ into equivalence classes. Take Γ' to be a subset of Γ that contains exactly one representative from each equivalence class. That is, $\Gamma' \subseteq \Gamma$ such that

- For every $\varphi \neq \psi \in \Gamma'$, $\varphi \not\equiv \psi$, and
- For every $\varphi \in \Gamma$, there is $\psi \in \Gamma'$ such that $\varphi \equiv \psi$.

Since $\Gamma' \subseteq \Gamma$, Γ' is also finitely satisfiable.

Recall that $\mathsf{occ}(\varphi)$ is defined to be the set of propositions that appear in φ. For $i \geq 0$, define Γ_i to be

$$\Gamma_i = \{\varphi \in \Gamma' \mid \mathsf{occ}(\varphi) \subseteq \{p_0, p_1, \ldots p_{i-1}\}\}.$$

Observe that Γ_i defines an non-decreasing sequence of sets, i.e., for every $i, \Gamma_i \subseteq \Gamma_{i+1}$. Also, $\Gamma' = \cup_{i \geq 0} \Gamma_i$. The most important observation about Γ_i is that it is a finite set — since Γ' has only one formula from each equivalence class of $\equiv$, each formula in Γ_i corresponds to a unique subset of assignments of $\{p_0, \ldots p_{i-1}\}$ to $\{\mathsf{F}, \mathsf{T}\}$. Thus, we have $|\Gamma_i| \leq 2^{2^i}$. Since Γ' is finitely satisfiable, Γ_i is satisfiable for every i.

Define T_Γ as the following set of (partial) truth assignments.

$$T_\Gamma = \bigcup_{i>0} \{\mathsf{v} : \{p_0, \ldots p_{i-1}\} \rightarrow \{\mathsf{F}, \mathsf{T}\} \mid \mathsf{v} \models \Gamma_i\}.$$

Recall that we say $\mathsf{v} \models \Gamma_i$ if v satisfies all formulas in Γ_i. Consider an assignment $\mathsf{v} \in T_\Gamma$ with domain $\{p_0, p_1, \ldots p_{i-1}\}$. By definition $\mathsf{v} \models \Gamma_i$. For $j < i$, since $\Gamma_j \subseteq \Gamma_i$, $\mathsf{v} \models \Gamma_j$. Further, Γ_j only has propositions $\{p_0, \ldots p_{j-1}\}$, we also have $\mathsf{v}' \models \Gamma_j$, where v' is the restriction of v to the domain $\{p_0, \ldots p_{j-1}\}$. So $\mathsf{v}' \in T_\Gamma$. Viewed as paths in the infinite binary tree (see Fig. 1.2), v is a path of length i, v' its prefix of length j. What we observe is that T_Γ is "closed" under prefix of paths. Thus, if we restrict our attention to the assignments in T_Γ then they form a subtree of the infinite binary tree.

Let us consider T_Γ. In the previous paragraph we observed that it forms a subtree of the infinite binary tree. It has paths of arbitrary length; this is because every Γ_i is satisfiable, and an (partial) assignment satisfying Γ_i is a path of length i in the tree. Since T_Γ is a binary tree, by König's lemma, it has an infinite path. The infinite path corresponds to a (full) truth assignment, say v_*. Further, since every prefix of v_* is a (finite) path in T_Γ, it means that the prefix of length i (viewed as a partial assignment) satisfies Γ_i. Therefore, for every i, $\mathsf{v}_* \models \Gamma_i$, and hence $\mathsf{v}_* \models \Gamma'$. Now, since every formula $\varphi \in \Gamma$ is logically equivalent to some formula $\psi \in \Gamma'$, it means $\mathsf{v}_* \models \Gamma$. Thus, Γ is satisfiable. □

1.4.2 Compactness using Henkin Models

The proof of the compactness theorem we present in the section, relies on constructing a truth assignment for a set of formulas Γ through the process of *saturation*, where we add formulas to the set Γ as long as it remains finitely satisfiable. This is an approach proposed by Henkin.

Let us fix Γ to be a finitely satisfiable set of formulas. We begin by making an important observation about such sets, namely, it can always be extended by adding a formula or its nagation, while preserving the property of finite satisfiability.

Lemma 1.29 *Let Γ be finitely satisfiable, and let φ be any formula. Then either $\Gamma \cup \{\varphi\}$ or $\Gamma \cup \{\neg\varphi\}$ is finitely satisfiable.*

Proof Assume (for contradiction), neither $\Gamma \cup \{\varphi\}$ nor $\Gamma \cup \{\neg\varphi\}$ is finitely satisfiable. By definition of finite satisfiability, this means that there are finite subsets $\Gamma_0 \subseteq \Gamma \cup \{\varphi\}$ and $\Gamma_1 \subseteq \Gamma \cup \{\neg\varphi\}$ that are *not* satisfiable. Consider the (finite) set $\Delta = (\Gamma_0 \cup \Gamma_1) \setminus \{\varphi, \neg\varphi\}$. Observe that since $\Delta \subseteq \Gamma$, Δ is satisfiable. Let v be a satisfying truth assignment for Δ. Then either $\mathsf{v} \models \varphi$ or $\mathsf{v} \models \neg\varphi$. Therefore, either $\mathsf{v} \models \Gamma_0$ or $\mathsf{v} \models \Gamma_1$, which contradicts our assumption that both Γ_0 and Γ_1 are unsatisfiable. □

The set of all formulas of propositional logic are *countable*, i.e., there is a 1-to-1, onto function $f : \mathbb{N} \to \mathcal{F}$, where $\mathcal{F}$ is the set of all propositional logic formulas. Therefore, we can *enumerate* all the formulas in propositional logic. Let $\varphi_0, \varphi_1, \ldots$ be an enumeration of all formulas. Let us, inductively, define a sequence of sets of formulas as follows.

$$\begin{aligned} \Delta_0 &= \Gamma \\ \Delta_n &= \begin{cases} \Delta_{n-1} \cup \{\varphi_{n-1}\} & \text{if this is finitely satisfiable} \\ \Delta_{n-1} \cup \{\neg\varphi_{n-1}\} & \text{otherwise} \end{cases} \end{aligned}$$

Observe that the sequence is non-decreasing, i.e., for every n, $\Delta_n \subseteq \Delta_{n+1}$. Further, by induction on n, using Lemma 1.29, we can conclude that Δ_n is finitely satisfiable for all n. Finally, define

$$\Delta = \bigcup_{n \in \mathbb{N}} \Delta_n.$$

Since Δ_n is finitely satisfiable for all $n \in \mathbb{N}$, we can conclude that Δ is also finitely satisfiable.

Proposition 1.30 *Δ is finitely satisfiable.*

Proof Consider any finite subset $X = \{\psi_1, \ldots \psi_m\}$ of Δ. Observe, by definition Δ, for each i, there is some n_i such that $\psi \in \Delta_{n_i}$. Taking $n = \max\{n_1, \ldots n_m\}$, observe that $X \subseteq \Delta_n$. Since Δ_n is finitely satisfiable, X is satisfiable. This means that Δ is finitely satisfiable. □

Finite satisfiability of Δ implies that Δ is a *complete* set.

Proposition 1.31 *For any formula φ, $\neg\varphi \in \Delta$ if and only if $\varphi \notin \Delta$.*

Proof Without loss of generality assume that φ is the nth formula, i.e., $\varphi = \varphi_n$. Now by definition, in step n of the construction of Δ, if $\varphi \notin \Delta$ then $\neg\varphi \in \Delta$. On the other hand, if $\{\varphi, \neg\varphi\} \subseteq \Delta$ then since $\{\varphi, \neg\varphi\}$ is not satisfiable, Δ would not be finitely satisfiable. But since Δ is finitely satisfiable, it must be the case that at most one out of φ and $\neg\varphi$ belong to Δ. □

We are now ready to complete the proof of Theorem 1.26. That is, we will show that Γ (which is finitely satisfiable) is satisfiable. Consider the truth assignment v defined as follows.

$$\mathsf{v}(p) = \begin{cases} \mathsf{T} \text{ if } p \in \Delta \\ \mathsf{F} \text{ if } \neg p \in \Delta \end{cases}$$

Note that v is well-defined because by Proposition 1.31, for any proposition p, exactly one among p and $\neg p$ is in Δ. v shows that Δ is satisfiable, because of the following result.

Proposition 1.32 *For any formula* $\varphi \in \Delta$, $\mathsf{v} \models \varphi$.

Proof Consider an arbitrary $\varphi \in \Delta$. Let $P = \mathsf{occ}(\varphi)$ and $P^{\neg} = \{\neg p \mid p \in P\}$. Consider the set

$$U = (\Delta \cap P) \cup (\Delta \cap P^{\neg}) \cup \{\varphi\}.$$

Since U is a finite subset of Δ, by Proposition 1.30, we have U is satisfiable. Let v' be a truth assignment such that $\mathsf{v}' \models U$. Observe that for every $p \in \Delta \cap P$, $\mathsf{v}'(p) = \mathsf{T}$ and for every $\neg p \in \Delta \cap P^{\neg}$, $\mathsf{v}'(p) = \mathsf{F}$. Therefore, v and v' agree on all propositions in P. By Lemma 1.13, since $\mathsf{v}' \models \varphi$, we have $\mathsf{v} \models \varphi$. □

Proposition 1.32 establishes the fact that $\mathsf{v} \models \Delta$. Since $\Gamma \subseteq \Delta$, $\mathsf{v} \models \Gamma$. Therefore, Γ is satisfiable.

1.4.3 An Application of Compactness: Coloring Infinite Planar Graphs

In this section we present an application of the compactness theorem. We will show that all infinite planar graphs are 4-colorable. We begin by recalling the graph coloring problem, and its connection to propositional logic.

Definition 1.33 (Graphs)

An undirected graph $G = (V, E)$ is a set of vertices V, and a set of edges $E \subseteq V \times V$, such that E is symmetric (i.e., $(u, v) \in E$ iff $(v, u) \in E$) and irreflexive (i.e., $(u, u) \notin E$ for any $u \in V$).

Definition 1.34 (Coloring)

A k-coloring of graph $G = (V, E)$ is a function $c : V \to \{1, 2, \ldots k\}$ such that if $(u, v) \in E$ then $c(u) \neq c(v)$. If G has a k-coloring then G is said to be k-colorable.

The problem of determining whether a graph is k-colorable can be "reduced" to checking the satisfiability of a set of formulas.

Proposition 1.35 *For any graph $G = (V, E)$ (with possibly infinitely many vertices), there is a set of formulas $\Gamma_{G,k}$ such that G is k-colorable iff $\Gamma_{G,k}$ is satisfiable.*

Proof For each vertex $u \in V$ and $1 \leq i \leq k$, take the proposition r_{ui} to denote "vertex u has color i". $\Gamma_{G,k}$ is the following set of formulas.

- For each $u \in V$, the formula $r_{u1} \vee r_{u2} \vee \cdots \vee r_{uk}$. Intuitively these formulas capture the constraint that every vertex gets at least one of the k colors.
- For each $u \in V$ and $1 \leq i, j \leq k$ with $i \neq j$, the formula $\neg r_{ui} \vee \neg r_{uj}$. These formulas capture the constraint that a vertex does not get two different colors.
- For each edge $(u, v) \in E$ and color $1 \leq i \leq k$, the formula $\neg r_{ui} \vee \neg r_{vi}$. These formulas ensure that adjacent vertices do not get the same color.

For a coloring c, define the valuation v_c such that $\mathsf{v}_c(r_{ui}) = \mathsf{T}$ iff $c(u) = i$. Similarly for a valuation v, define a function $c_{\mathsf{v}}(u) = i$ iff $\mathsf{v}(r_{ui}) = \mathsf{T}$. Observe that

- If c is a valid k-coloring of G then v_c satisfies $\Gamma_{G,k}$, and
- If v satisfies $\Gamma_{G,k}$ then c_{v} is a valid k-coloring of G.

Proof of the above observations is left as exercise. □

Finite, planar graphs are graphs with finitely many vertices such that there is a drawing of the graph on the plane where the edges do not cross. A celebrated result about finite, planar graphs is that 4 colors are sufficient to color the graph.

Theorem 1.36 (Appel-Haken)

Every finite planar graph is 4-colorable.

We will show that the compactness theorem in fact shows that Theorem 1.36 can be extended to infinite graphs as well.

Corollary 1.37 *All infinite planar graphs are 4-colorable.*

Proof Let G be an infinite planar graph. Consider the set of formulas $\Gamma_{G,4}$ constructed in Proposition 1.35. Observe that $\Gamma_{G,4}$ is finitely satisfiable. This can be seen as follows. Let Γ_0 be any finite subset of $\Gamma_{G,4}$. Let G_0 be the graph induced by the vertices u such that the proposition p_{ui} appears in Γ_0 for some i. Now, G_0 is a finite, planar graph and so by Theorem 1.36 has a 4-coloring c. Then by the proof of Proposition 1.35, the valuation v_c satisfies $\Gamma_{G_0,4}$. Since $\Gamma_0 \subseteq \Gamma_{G_0,4}$, we have v_c satisfies Γ_0.

Since $\Gamma_{G,4}$ is finitely satisfiable, by the compactness theorem, $\Gamma_{G,4}$ is satisfiable. Let v be a satisfying assignment for $\Gamma_{G,4}$. Again, by Proposition 1.35, c_{v} is a valid 4-coloring of G. □

Chapter 2
Proof Systems

If logic is the science of valid inference, then proofs embody its heart. But what are mathematical proofs? They are a sequence of statements where each statement in the sequence is either a self evident truth, or "logically" follows from previous observations. Thus, sound derivation principles are identified by correct proofs.

Example 2.1 Euclid's Elements sets out axioms (or postulates), which are self evident truths, and proves all results in geometry from these truths formally. Euclid lays out five axioms for geometry.

A1 A straight line can be drawn from any point to any point.
A2 A finite line segment can be extended to an infinite straight line.
A3 A circle can be drawn with any point as center and any given radius.
A4 All right angles are equal.
A5 If a straight line falling on two straight lines makes the interior angles on the same side less than two right angles, the straight lines, if produced indefinitely, will meet on that side on which the angles are less than two right angles.

Using these axioms, Euclid proves a number of results in geometry. He uses previously proved propositions in the proofs of later observations. An example of such a result, is the proof that the sum of the interior angles of a triangle is 180°.

Claim The interior angles of a triangle sum to two right angles.

Proof Consider the diagram in Fig. 2.1 . The proposition is proved using the following sequence of statements.

1. Extend one side (say) BC to D [A2]
2. Draw a line parallel to AB through point C; call it CE [P31]
3. Since AB is parallel to CE, BAC = ACE and ABC = ECD [P29]
4. Thus, the sum of the interior angles = ACB + ACE + ECD = 180°

References [P31] and [P29] in steps 2 and 3, allude to previously propositions 31 and 29, proved in the book. □

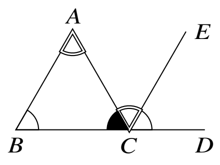

Fig. 2.1 Proof that the sum of the internal angles of a triangle are 180°

Example 2.1 highlights the basic elements of identifying good proofs — one needs to identify axioms, and the principle by which new conclusions can be drawn from previously established facts. A formal proof system for a logic identifies such *axioms* and *rules of inference*. We will introduce two such proof systems for propositional logic — a Frege-style proof system, and resolution — to give a flavor of different types of proof systems.

2.1 A Frege-style Proof System

Proof systems are most convenient presented as a collection of rules of the form

$$\frac{\Gamma}{\varphi}$$

where Γ is a set of formulas (schemas) and φ is a formula (schema). Such rules can be interpreted as follows — if every formula in Γ can be established then φ can be concluded from these observations. One special case is when $\Gamma = \emptyset$. In this case the formula below the line can be concluded without establishing anything; in other words, it is an axiom. Instead of explicitly writing $\emptyset$ above the line, we simply don't write anything, and present this axiom in the form

$$\frac{}{\varphi}$$

Before presenting our first proof system, observe that all propositional logic formulas can be expressed using just implication and $\bot$. To see this observe that $\neg\varphi$ is the same as $\varphi \to \bot$ and $\varphi \vee \psi$ is $(\varphi \to \bot) \to \psi$. Our first proof system, shown in Fig. 2.2 , assumes that are formulas are written using implication and $\bot$.

Our first proof system has 3 axiom *schemas* and one rule of inference. The formulas φ, ψ, and ρ in Fig. 2.2 , can be *any formulas*. For example, taking $\varphi = p$ and $\psi = p$, we get $p \to (p \to p)$ as an *instantiation* of the first axiom schema, while taking $\varphi = p$ and $\psi = p \to p$, we get $p \to ((p \to p) \to p)$ as a different instantiation of the *same* schema. The rule of inference in this proof system, is a very commonly used rule. It, therefore, has a special name; it is called *modus ponens*.

$$\frac{}{\varphi \to (\psi \to \varphi)} \qquad \frac{}{(\varphi \to (\psi \to \rho)) \to ((\varphi \to \psi) \to (\varphi \to \rho))}$$

$$\frac{}{((\varphi \to \bot) \to \bot) \to \varphi} \qquad \frac{\varphi \qquad \varphi \to \psi}{\psi}$$

Fig. 2.2 A Frege-style Proof System

Proofs in our (formal) proof system, will be like the usual proofs in mathematics — they will be a sequence of statements. However, instead of using english statements, here they will simply be well formed formulas of propositional logic. The statement (or formula) being proved is the last one in the sequence. The sequence of formulas in a proof should be consistent with the axioms and rule of inference of the proof system, for it to be *valid* proof. This is captured in the definition below.

Definition 2.2 (Proofs)

A *proof* of φ from a set (possibly infinite) of hypotheses Γ is a finite sequence of wffs $\psi_1, \psi_2, \ldots \psi_m$ such that $\psi_m = \varphi$, and for every $k \in \{1, 2, \ldots m\}$, either

- $\psi_k \in \Gamma$, or
- ψ_k is an axiom, or
- ψ_k follows from ψ_i and ψ_j, with $i, j < k$, by modus ponens.

The *length* of such a proof is the number of wffs in the sequence, namely, m. If there is a proof of φ from Γ, we denote this by $\Gamma \vdash \varphi$. When $\Gamma = \emptyset$, we write this as $\vdash \varphi$ (as opposed to $\emptyset \vdash \varphi$).

Let us look at some proofs in our system.

Example 2.3 Let us construct a proof of $q \to p$ from the hypothesis $\{p\}$. Such a proof is as follows.

1. $p \to (q \to p)$	Axiom 1, taking $\varphi = p$, and $\psi = q$
2. p	Hypothesis in set Γ
3. $q \to p$	Modus Ponens on lines 1 and 2

Thus, $\{p\} \vdash q \to p$.

We will now show that $\vdash \bot \to ((p \to q) \to (p \to p))$.

1. $(p \to (q \to p)) \to ((p \to q) \to (p \to p))$
   Axiom 2, taking $\varphi = p$, $\psi = q$ and $\rho = p$
2. $p \to (q \to p)$
   Axiom 1, taking $\varphi = p$, and $\psi = q$
3. $(p \to q) \to (p \to p)$
   Modus ponens on lines 1 and 2
4. $((p \to q) \to (p \to p)) \to (\bot \to ((p \to q) \to (p \to p)))$
   Axiom 1, taking $\varphi = (p \to q) \to (p \to p)$, and $\psi = \bot$
5. $\bot \to ((p \to q) \to (p \to p))$
   Modus ponens on lines 3 and 4

Finally, let us show $\vdash p \rightarrow p$.

1. $(p \rightarrow ((p \rightarrow p) \rightarrow p)) \rightarrow ((p \rightarrow (p \rightarrow p)) \rightarrow (p \rightarrow p))$
   Axiom of 2, taking $\varphi = p$, $\psi = p \rightarrow p$ and $\rho = p$
2. $(p \rightarrow ((p \rightarrow p) \rightarrow p))$
   Axiom 1, taking $\varphi = p$, and $\psi = p \rightarrow p$
3. $(p \rightarrow (p \rightarrow p)) \rightarrow (p \rightarrow p)$
   Modus ponens on lines 1 and 2
4. $p \rightarrow (p \rightarrow p)$
   Axiom 1, taking $\varphi = p$, $\psi = p$
5. $p \rightarrow p$
   Modus ponens on lines 3 and 4

In proof systems, like the one we are considering in this section, there is a very useful theorem that makes writing proofs easy. This is called the *deduction theorem*. Some proof systems have it as an explicit rule.

Theorem 2.4 (Deduction Theorem)
If $\Gamma \cup \{\varphi\} \vdash \psi$ *then* $\Gamma \vdash \varphi \rightarrow \psi$.

First, observe that the converse of Theorem 2.4, is clearly true, i.e., if $\Gamma \vdash \varphi \rightarrow \psi$ then $\Gamma \cup \{\varphi\} \vdash \psi$. Establishing this left as an exercise. The proof of the deduction theorem is a more difficult exercise. The informal outline of the proof is as follows. Assume that $\rho_1, \rho_2, \ldots \rho_m$ is a proof of ψ from $\Gamma \cup \{\varphi\}$. One shows by induction on i that, for each line i, we have $\Gamma \vdash \varphi \rightarrow \rho_i$.

The deduction theorem simplifies the task of writing down proofs in our proof system.

Example 2.5 Consider the task of showing $\vdash (\varphi \rightarrow \psi) \rightarrow ((\psi \rightarrow \rho) \rightarrow (\varphi \rightarrow \rho))$, where φ, ψ, and ρ are arbitrary wffs. Our approach to solving this problem, would instead be to instead establish $\{\varphi \rightarrow \psi, \psi \rightarrow \rho, \varphi\} \vdash \rho$. If we succeed, we will get the desired result by using the deduction theorem a few times.

1. $\{\varphi \rightarrow \psi, \psi \rightarrow \rho, \varphi\} \vdash \varphi \rightarrow \psi$	Hypothesis
2. $\{\varphi \rightarrow \psi, \psi \rightarrow \rho, \varphi\} \vdash \varphi$	Hypothesis
3. $\{\varphi \rightarrow \psi, \psi \rightarrow \rho, \varphi\} \vdash \psi$	Modus Ponens on lines 1 and 2
4. $\{\varphi \rightarrow \psi, \psi \rightarrow \rho, \varphi\} \vdash \psi \rightarrow \rho$	Hypothesis
5. $\{\varphi \rightarrow \psi, \psi \rightarrow \rho, \varphi\} \vdash \rho$	Modus Ponens on lines 3 and 4
6. $\{\varphi \rightarrow \psi, \psi \rightarrow \rho\} \vdash \varphi \rightarrow \rho$	Deduction Theorem
7. $\{\varphi \rightarrow \psi\} \vdash (\psi \rightarrow \rho) \rightarrow (\varphi \rightarrow \rho)$	Deduction Theorem
8. $\vdash (\varphi \rightarrow \psi) \rightarrow ((\psi \rightarrow \rho) \rightarrow (\varphi \rightarrow \rho))$	Deduction Theorem

2.1.1 Completeness Theorem

Proofs in a formal proof system like the one in Fig. 2.2 try capture the notion of correct logical inference. But what do we mean by “correct logical inference”? There are two aspects to such a question. First, any conclusion drawn by a proof must be logically correct, i.e., consistent with the semantics we defined. In other words, we should not be able to conclude any false facts using a proof. This is often referred to as the *soundness* of the proof system. The second is that the proof system must be rich enough to be able to prove all true facts. This is called the *completeness* of the proof system. We make this connection between provability and the semantics we gave precise in the following theorem.

Theorem 2.6 (Soundness and Completeness)

For any set of formulas Γ *(possibly even infinite) and any wff* φ *the following two properties hold.*

Soundness *If* $\Gamma \vdash \varphi$ *then* $\Gamma \models \varphi$.
Completeness *If* $\Gamma \models \varphi$ *then* $\Gamma \vdash \varphi$.

Thus, any formula proved without any hypotheses is a tautology, and every tautology has a proof from the empty set of hypotheses in our proof system. We will not prove Theorem 2.6. We will instead prove such a soundness and completeness theorem for the second proof system that we will introduce for propositional logic. Proving soundness is usually easy. It requires making sure that the axioms and proof rules are consistent with the semantics of the logic. In this case it requires showing that every axiom in the proof system is indeed a tautology, and modus ponens is consistent with logical consequence. Proving completeness is typically hard.

2.2 Resolution

Notice that the proofs for formulas that we constructed in Examples 2.3 and 2.5 are very symbolic and mechanical — one doesn't need to understand what the formula we are trying to prove is saying or what the meaning of the hypotheses is. Instead the proofs are constructed by looking at the pattern of formulas. This raises the prospect of trying to mechanize the process of searching for proofs of formulas. However, the proof system in Sect. 2.1 is not good for this purpose. This is because at any point during the construction of the proof, one can extend it by using any one of the axiom schemas. Since each axiom schema can be instantiated in infinitely many possible ways, this makes mechanization difficult. A proof system that is amenable to mechanization is one that has very few choices at any step of the proof construction. *Resolution* is such a proof system. Resolution has no axioms and only one rule of inference.

Resolution is a method for *refutations*, i.e., it proves that a formula is "not true" or more precisely *unsatisfiable*; recall that a formula φ is unsatisfiable if $\mathsf{v} \not\models \varphi$ for all valuations v. Thus unlike the proof system in Sect. 2.1 which proves validity of a formula directly, resolution works by showing that the negation of the formula is unsatisfiable. One can imagine resolution refutations like a proof by contradiction, where one assumes the negation of what one is trying to establish, and trying to show that that is impossible.

Resolution works when the formulas are represented in *conjunctive normal form* (CNF). We begin, therefore, by introducing conjunctive normal form formulas. CNF formulas are built using propositions and their negations, and the logical operators of $\wedge$ and $\vee$. It will be convenient to think of $\vee$ and $\wedge$ being present implicitly (see Definition 2.7 below) instead of explictly in the syntax.

Definition 2.7 (Conjunctive Normal Form)

Conjunctive normal form formulas are defined as follows.

- A *literal* is a proposition p or its negation $\neg p$.
- A *clause* is a disjunction of literals. We will think of a clause as a set of literals, implicitly assuming that the literals are disjuncted. In this interpretation, a truth valuation satisfies a clause if some literal in the set evaluates to 1 under the truth valuation.
- The *empty clause* is the clause containing no literals. By definition, no truth assignment satisfies the empty clause.
- A formula is said to be in *conjunctive normal form* (CNF) if it is conjunction of clauses. Again, we will think of a CNF formula as a set of clauses, with the conjunction being implicit in the syntax. With this interpretation, a truth valuation satisfies a CNF formula if it satisfies every clause in the set representing the formula.

CNF formulas as sets of sets of literals

In Definition 2.7, clauses are represented as sets of literals, and CNF formulas as sets of clauses. Why is this well-defined? The reason is because $\vee$ and $\wedge$ are both *idempotent*, *commutative*, and *associative*. Idempotence means that disjuncting/conjuncting a formula with itself is equivalent to the formula. In other words, for any wff φ we have

$$\varphi \vee \varphi \equiv \varphi \qquad \varphi \wedge \varphi \equiv \varphi.$$

Idempotence of disjunction and conjunction ensures that representing them as sets which don't have repeated elements is faithful with the semantics. Commutativity and associativity mean that the order in which formulas are disjuncted/conjuncted does not change its semantics. That is, for any formulas φ, ψ, and ρ,

$$\begin{array}{ll} \varphi \vee \psi \equiv \psi \vee \varphi & \varphi \wedge \psi \equiv \psi \wedge \varphi \\ (\varphi \vee \psi) \vee \rho \equiv \varphi \vee (\psi \vee \rho) & (\varphi \wedge \psi) \wedge \rho \equiv \varphi \wedge (\psi \wedge \rho). \end{array}$$

Thus, commutativity and associativity ensure that sets (which are ordered collections) are faithful representations of disjunctions and conjunctions of formulas.

CNF formulas are formulas in a restricted form. However, they are not semantically restrictive. That is, every wff φ can be shown to be equivalent to a formula ψ in CNF — we can push negations all the way in using DeMorgan's Laws, and then distribute $\vee$ over $\wedge$. We will look at this conversion process more closely later in this chapter. Let us look at some examples of CNF and non-CNF formulas.

Example 2.8 The formulas $(p_1 \wedge q_1) \vee (p_2 \wedge q_2)$, $\neg(p \wedge q)$ are examples of formulas that are not in CNF — neither formula has $\wedge$ as the topmost connective. The formulas $(p_1 \vee p_2) \wedge (p_1 \vee q_2) \wedge (q_1 \vee p_2) \wedge (q_1 \vee q_2)$ and $\neg p \vee \neg q$ are formulas in CNF as they are conjunctions of clauses.

We will represent CNF formulas as set of set of literals, without explict conjunctions and disjunctions. For example, $\{\{p_1, p_2\}, \{p_1, q_2\}, \{q_1, p_2\}, \{q_1, q_2\}\}$ is the way the formula $(p_1 \vee p_2) \wedge (p_1 \vee q_2) \wedge (q_1 \vee p_2) \wedge (q_1 \vee q_2)$ will be represented. Similarly, $\{\{\neg p, \neg q\}\}$ will be the representation of $\neg p \vee \neg q$.

The resolution proof system is a sequence of transformations that preserve satisfiability, until the empty clause (which is by definition not satisfiable) is obtained. The transformations involve the single rule of inference that constructs the *resolvent* of two clauses.

Definition 2.9 (Resolvent) The only rule in the resolution proof system is as follows.

$$\frac{C \cup \{p\} \qquad D \cup \{\neg p\}}{C \cup D}$$

The conclusion $C \cup D$ is called the *resolvent* of $C \cup \{p\}$ and $D \cup \{\neg p\}$ with respect to proposition p.

Two clauses may have multiple resolvents depending on which proposition one chooses to resolve with respect to. The resolvent of two clauses may be the empty clause if C and D are empty sets. Let us look at some examples to clarify this definition.

Example 2.10 Consider the clauses $\{p, \neg q, \neg r\}$ and the clause $\{\neg p, \neg q\}$. The resolvent of these two clauses (with respect to p) is the clause $\{\neg q, \neg r\}$.

On the other hand, if we consider clauses $\{p, \neg q\}$ and $\{\neg p, q\}$, we have two possible resolvents. If we resolve with respect to p, we get $\{q, \neg q\}$, and if we resolve with respect to q, we get $\{p, \neg p\}$.

Definition 2.11 (Refutations)

A *resolution refutation* of a (possibly infinite) set of clauses Γ is a sequence of clauses $C_1, C_2, \ldots C_m$ such that each clause C_k is either in Γ or a resolvent of two clauses C_i and C_j $(i, j < k)$, and the last clause C_m in the refutation is the empty clause.

Example 2.12 The set of clauses $\Gamma = \{\{p, q\}, \{\neg p, r\}, \{\neg q, r\}, \{\neg r\}\}$ has the following resolution refutation.

1. $\{\neg p, r\}$	
2. $\{\neg r\}$	
3. $\{\neg p\}$	Resolvent of 1 and 2
4. $\{\neg q, r\}$	
5. $\{\neg q\}$	Resolvent of 2 and 4
6. $\{p, q\}$	
7. $\{q\}$	Resolvent of 3 and 6
8. $\{\}$	Resolvent of 5,7

2.2.1 Proving Tautologies with Resolution

As mentioned before, resolution refutations establish the unsatisfiability of a formula given in CNF. To prove that a formula is a tautology using resolution, we need to use Proposition 1.24. That is, to prove that φ is a tautology, we need to convert $\neg\varphi$ into CNF. This can be done as follows.

1. Push negations inside using DeMorgan's Laws. Recall that DeMorgan's laws say the following.

$$\neg(\psi_1 \wedge \psi_2) \equiv \neg\psi_1 \vee \neg\psi_2 \qquad \neg(\psi_1 \vee \psi_2) \equiv \neg\psi_1 \wedge \neg\psi_2$$

2. Remove double negations, because $\neg\neg\psi \equiv \psi$
3. Distribute $\vee$ over $\wedge$, using the distributive law, which says

$$\psi_1 \vee (\psi_2 \wedge \psi_3) \equiv (\psi_1 \vee \psi_2) \wedge (\psi_1 \vee \psi_3) \qquad (\psi_1 \wedge \psi_2) \vee \psi_3 \equiv (\psi_1 \vee \psi_3) \wedge (\psi_2 \vee \psi_3)$$

Example 2.13 Let $\varphi = (\neg p_1 \vee \neg q_1) \wedge (\neg p_2 \vee \neg q_2)$. We can convert $\neg\varphi$ to CNF as follows.

1. Pushing negations inside using DeMorgan's Laws, we get

$$(\neg\neg p_1 \wedge \neg\neg q_1) \vee (\neg\neg p_2 \wedge \neg\neg q_2)$$

2. Removing double negations, we get

$$(p_1 \wedge q_1) \vee (p_2 \wedge q_2)$$

3. Distributing $\vee$ over $\wedge$, we get

$$(p_1 \vee p_2) \wedge (p_1 \vee q_2) \wedge (q_1 \vee p_2) \wedge (q_1 \vee q_2)$$

The above method while semantically correct, can be expensive. The main reason is that when we distribute $\vee$ over $\wedge$, the resulting formula can be exponentially larger. For example, if we have the formula

$$\bigvee_{i=1}^{n}(p_i \wedge q_i)$$

and we distribute $\vee$ over $\wedge$, we will get a CNF formula where each clause is of the form $(r_1 \vee r_2 \vee \cdots \vee r_n)$, where r_i is either p_i or q_i. This will result in a formula with 2^n clauses (as we have two choices for each r_i).

The exponential blowup can be avoided by using a translation proposed by Tsejtin. Tseijtin's method does not construct a logically equivalent CNF formula. Instead, for the formula $\neg\varphi$, it constructs a CNF formula ψ with the property that $\neg\varphi$ is satisfiable if and only if ψ is satisfiable. This weaker correspondence between $\neg\varphi$ and ψ is sufficient in this context; we will have φ is a tautology if and only if ψ is unsatisfiable.

Tsejtin's Method.

Let us describe the conversion of formula $\neg\varphi$. The first step is to introduce new *extension* propositions x_ψ for each subformula ψ of φ as follows.

- For each proposition in φ, the extension proposition x_p is the same as p.
- For each negated subformula $\neg\psi$, $x_{\neg\psi}$ is taken to be the literal $\neg x_\psi$.
- For all other subformulas ψ, x_ψ is a new proposition.

Having identified the extension propositions, the CNF formula that we will construct corresponding to $\neg\varphi$ is as follows. Here, we will use the representation of CNF formulas as sets of sets of literals. So for φ, we define Γ_φ to be the following set of clauses.

- The singleton clause $\{\neg x_\varphi\}$
- For each subformula $\psi \wedge \rho$, we add the CNF formula equivalent to "$x_{\psi\wedge\rho} \leftrightarrow x_\psi \wedge x_\rho$. In other words, we will have the clauses

$$\{\neg x_{\psi\wedge\rho}, x_\psi\} \qquad \{\neg x_{\psi\wedge\rho}, x_\rho\} \qquad \{x_{\psi\wedge\rho}, \neg x_\psi, \neg x_\rho\}$$

- For each subformula $\psi \vee \rho$, we add the CNF formula equivalent to "$x_{\psi\vee\rho} \leftrightarrow x_\psi \vee x_\rho$. In other words, we will have the clauses

$$\{x_{\psi\vee\rho}, \neg x_\psi\} \qquad \{x_{\psi\vee\rho}, \neg x_\rho\} \qquad \{\neg x_{\psi\vee\rho}, x_\psi, x_\rho\}$$

Observe that the number of subformulas of a given formula φ is linear in the size of φ. Thus the number of extension propositions we introduce is linear in φ. Further, for each subformula, we are introducing only a constant (3 to be precise) number of clauses. This means that resulting set of clauses Γ_φ is linear in the size of φ.

Example 2.14 Let us apply Tsejtin's construction to the formula in Example 2.13. Recall that $\varphi = (\neg p_1 \vee \neg q_1) \wedge (\neg p_2 \vee \neg q_2)$. The first step is to identify the new extension propositions we need. In this case there are only 3 that we will add — x_φ corresponding to φ, x_1 corresponding to $(\neg p_1 \vee \neg q_1)$, and x_2 corresponding to

$(\neg p_2 \vee \neg q_2)$. Our CNF formula Γ_φ will be obtained by adding 3 clauses for each of these interesting subformulas corresponding to x_φ, x_1 and x_2. Thus, we have Γ_φ is the following set

$$\left\{\begin{array}{l} \{\neg x_\varphi\}, \\ \{\neg x_\varphi, x_1\}, \{\neg x_\varphi, x_2\}, \{x_\varphi, \neg x_1, \neg x_2\}, \\ \{x_1, p_1\}, \{x_1, q_1\}, \{\neg x_1, \neg p_1, \neg q_1\}, \\ \{x_2, p_2\}, \{x_2, q_2\}, \{\neg x_2, \neg p_2, \neg q_2\} \end{array}\right\}$$

In the above description, we have replaced $\neg\neg p$ by p for $p \in \{p_1, q_1, p_2, q_2\}$.

2.2.2 Completeness of Resolution

We will now prove that resolution is a "correct" proof system. In other words, we will prove the soundness and completeness of resolution. What that means for resolution is that a set of clauses Γ has a resolution refutation if and only if Γ is unsatisfiable. Note that we will establish this for any set Γ, including those that are infinite. Recall that a set of clauses Γ is satisfiable if there is a truth assignment v such that for every clause $C \in \Gamma$, there is some literal $\ell \in C$ such that $\mathsf{v}[\![\ell]\!] = \mathsf{T}$.

Theorem 2.15 (Soundness)

If a set of clauses Γ has a resolution refutation, then Γ is unsatisfiable.

Proof The crux of the proof of the soundness theorem, is to establish the "correctness" of the resolution proof rule. That is captured by the following lemma.

Lemma 2.16 (Resolution Lemma)

Let Δ be a set of clauses and let C be the resolvent of two clauses $D, E \in \Delta$. Then for any assignment v, $\mathsf{v} \models \Delta$ if and only if $\mathsf{v} \models \Delta \cup \{C\}$. □

Proof (Of Lemma 2.16) Observe that if $\mathsf{v} \models \Delta \cup \{C\}$, then clearly $\mathsf{v} \models \Delta$.

Consider a truth assignment such that $\mathsf{v} \models \Delta$. Without loss of generality, let us assume there is a proposition p such that $p \in D$ and $\neg p \in E$. If $\mathsf{v}(p) = \mathsf{T}$ then since $\mathsf{v} \models E$, there must be a literal $\ell \in E$ (obviously $\ell \neq \neg p$) such that $\mathsf{v}[\![\ell]\!] = \mathsf{T}$. Since $\ell \in C$, we have $\mathsf{v} \models C$. On the other hand, if $\mathsf{v}(p) = \mathsf{F}$ then since $\mathsf{v} \models D$, there must be a literal $\ell \in D$ (obviously $\ell \neq p$) such that $\mathsf{v}[\![\ell]\!] = \mathsf{T}$. Since $\ell \in C$, we have $\mathsf{v} \models C$. □

Using Lemma 2.16, we are ready to complete the proof of Theorem 2.15. Let $C_1, C_2, \ldots C_m$ be a resolution refutation of Γ. Let us define a sequence of sets of clauses inductively as follows.

$$\Gamma_0 = \Gamma \qquad \Gamma_i = \Gamma_{i-1} \cup \{C_i\}$$

Thus, $\Gamma_i = \Gamma \cup \{C_1, C_2, \ldots C_i\}$. Since $C_m = \{\}$, Γ_m is clearly unsatisfiable. Therefore, by Lemma 2.16 (and induction), $\Gamma = \Gamma_0$ is also unsatisfiable. □

Theorem 2.17 (Completeness)
If a set of clauses Γ *is unsatisfiable then there is a resolution refutation of* Γ.

We will present a proof of the completeness theorem due to David and Putnam. For finite Γ, the proof is constructive. That is, when Γ is unsatisfiable, it gives a specific construction of a refutation for Γ.

Proof Let Γ be an unsatisfiable set of clauses. By the compactness theorem (Theorem 1.26), there is a finite subset $\Delta \subseteq \Gamma$ that is unsatisfiable; this can be seen by taking $\varphi = \bot$ in Corollary 1.27. We will prove the completeness theorem by induction on the number of propositions appearing in Δ. Before outlining the proof, it is useful to point out that since Δ is unsatisfiable, it must be a *non-empty* set of clauses. This is because an empty set of clauses is (by definition) satisfiable.

For the base case, observe that if Δ contains no propositions, then Δ contains the empty clause. Then the refutation for Δ is simply $\{\}$.

Let us now consider the induction step. Let us call a clause C *trivial* if there is a proposition p such that $\{p, \neg p\} \subseteq C$. Trivial clauses are valid, and hence they can be removed from Δ without affecting its satisfiability. Thus, without loss of generality, we will assume that Δ does not have any trivial clauses. Let p be a proposition that appears in Δ. With respect to proposition p, Δ can be partitioned into 3 sets.

$$\begin{aligned}
\Delta_0^p &= \{C \in \Delta \mid C \cap \{p, \neg p\} = \emptyset\} \\
\Delta_+^p &= \{C \in \Delta \mid p \in C\} \\
\Delta_-^p &= \{C \in \Delta \mid \neg p \in C\}
\end{aligned}$$

Thus, Δ_0^p are clauses where p does not appear, Δ_+^p are those where p appears positively, and Δ_-^p are those where p appears negatively. Let us construct a new set of clauses as follows.

$$\Delta_p = \Delta_0^p \cup \{C \cup D \mid C \cup \{p\} \in \Delta_+^p \text{ and } D \cup \{\neg p\} \in \Delta_-^p\}$$

Thus, Δ_p has all the clauses in Δ_0^p and all resolvents of clauses from Δ_+^p and Δ_-^p. Observe that p no longer appears in Δ_p. If we can argue that Δ_p is unsatisfiable then we can complete the proof by using the induction hypothesis — the refutation for Δ is just all the steps to create Δ_p followed by a refutation for Δ_p.

To finish the proof, we need to establish the following lemma.

Lemma 2.18 *If* Δ_p *is satisfiable then so is* Δ. □

Proof (Of Lemma 2.18) Let v be a truth assignment that satisfies Δ_p. Let v' be the truth assignment that is identical to v, except that it flips the assignment to p. Observe that since v and v' only differ on the assignment to p, they agree on all the propositions appearing in Δ_p. Therefore, v' also satisfies Δ_p.

Let us assume without loss of generality, that $\mathsf{v}(p) = \mathsf{T}$ and $\mathsf{v}'(p) = \mathsf{F}$. We will show that either v or v' satisfies Δ. Observe that both v and v' satisfy Δ_0^p (because p does not appear in Δ_0^p). Also, v satisfies Δ_+^p (because all clauses in Δ_+^p have p) and v' satisfies Δ_-^p (because all clauses in Δ_-^p have $\neg p$). Now if v satisfies Δ_-^p, v satisfies

Δ. Similarly, if v' satisfies Δ_+^p then v' satisfies Δ. So the problem is if neither of these hold. In that case that is a clause $C \cup \{p\} \in \Delta_+^p$ that is not satisfied by v' and there is a clause $D \cup \{\neg p\} \in \Delta_-^p$ that is not satisfied by v. But then their resolvent $C \cup D \in \Delta_p$ is not satisfied by either v or v', which contradicts our assumption that both v and v' satisfy Δ_p. □

With the proof of Lemma 2.18, we have completed the proof of Theorem 2.17.□

2.3 Craig's Interpolation Theorem and Proof Complexity

Craig's Interpolation theorem is a classical result in logic that holds for many different logics. The theorem has been used in different contexts in the broad area of formal methods and verification — in hardware and software specification; reasoning about large knowledge databases; type inference; combination of theorem provers for different first order theories; model checking of finite and infinite state systems through the construction of abstractions. In this section we look at some of its connections to theoretical computer science and complexity theory in particular. We will introduce the theorem for propositional logic, and its connection with proof lengths for propositional logic formulas.

2.3.1 Craig's Interpolation Theorem

Before we state and prove the interpolation theorem, it will be convenient to introduce some notation. A list of propositions $p_1, p_2, \ldots p_n$ will be denoted by $\overrightarrow{p}$ when the actual number of propositions in the list is unimportant. A formula φ over propositions $p_1, p_2, \ldots p_n, q_1, q_2, \ldots q_m$ will be sometimes denoted as $\varphi(\overrightarrow{p}, \overrightarrow{q})$, making explicit the propositions that *may* syntactically appear in φ. Finally, for a truth valuation v and a list of propositions $\overrightarrow{p}$, $\mathsf{v} \restriction_{\overrightarrow{p}}$ will denote the restriction of v to the propositions $\overrightarrow{p}$; note that v has an infinite domain, the domain of $\mathsf{v} \restriction_{\overrightarrow{p}}$ is finite and restricted to $\overrightarrow{p}$.

Theorem 2.19 (Craig)
Suppose $\models \varphi(\overrightarrow{p}, \overrightarrow{q}) \rightarrow \psi(\overrightarrow{q}, \overrightarrow{r})$. *Then there is a formula* $\eta(\overrightarrow{q})$ *such that* $\models \varphi(\overrightarrow{p}, \overrightarrow{q}) \rightarrow \eta(\overrightarrow{q})$ *and* $\models \eta(\overrightarrow{q}) \rightarrow \psi(\overrightarrow{q}, \overrightarrow{r})$. η *is said to the* interpolant *of* φ *and* ψ.

Before presenting the proof of Theorem 2.19, let us examine its statement. There are different equivalent ways of presenting this result. Recall that $\models \varphi \rightarrow \psi$ iff $\varphi \models \psi$. Therefore, we could say that if ψ is a logical consequence of φ then η is a formula that captures the reason why, using only the common propositions. In this case, since $\varphi \models \eta$, we could think of η as an "abstraction" of φ (as η "forgets" the constraints φ imposes on $\overrightarrow{p}$) that is sufficient to ensure ψ. Another formulation of Craig's theorem is as follows. Suppose $\varphi(\overrightarrow{p}, \overrightarrow{q}) \wedge \psi(\overrightarrow{q}, \overrightarrow{r})$ is unsatisfiable (or equivalently, $\varphi \models \neg\psi$), then there is a formula $\eta(\overrightarrow{q})$ over the common propositions such that $\varphi \models \eta$ and $\eta \wedge \psi$ is unsatisfiable (i.e., $\eta \models \neg\psi$). Informally, η is an abstraction of φ that captures why $\varphi \wedge \psi$ is unsatisfiable. We will consider this formulation of Craig's theorem in terms of unsatisfiability, when we revisit resolution later in this section.

Proof (Of Theorem 2.19) The proof of Craig's Interpolation Theorem is quite simple in this context of propositional logic. Let us define

$$M = \{\mathsf{v} \restriction_{\overrightarrow{q}} \mid \mathsf{v}[\![\varphi]\!] = \mathsf{T}\}.$$

Observe that since the domain (and range) of all functions in M is finite, M is a finite set. Let us, without loss of generality, take $M = \{\mathsf{v}_1, \mathsf{v}_2, \ldots \mathsf{v}_n\}$. The interpolant η will essentially say that "one among the assignments in M hold". Formally,

$$\eta(\overrightarrow{q}) = \vee_{i=1}^{n}(q_1^{(i)} \wedge q_2^{(i)} \wedge \cdots \wedge q_k^{(i)})$$

where

$$q_j^{(i)} = \begin{cases} q_j & \text{if } \mathsf{v}_i(q_j) = \mathsf{T} \\ \neg q_j & \text{otherwise} \end{cases} \qquad \text{and} \qquad \overrightarrow{q} \text{ is } q_1, \ldots q_k$$

Clearly from the definition of M and η, we have $\varphi(\overrightarrow{p}, \overrightarrow{q}) \models \eta(\overrightarrow{q})$. Let us now argue that $\eta(\overrightarrow{q}) \models \psi(\overrightarrow{q}, \overrightarrow{r})$. Consider an arbitrary valuation v such that $\mathsf{v}[\![\psi]\!] = \mathsf{F}$. Since we have $\varphi \models \psi$, it must be that $\mathsf{v}[\![\varphi]\!] = \mathsf{F}$. Consider any assignment v' such that $\mathsf{v} \restriction_{\overrightarrow{q},\overrightarrow{r}} = \mathsf{v}' \restriction_{\overrightarrow{q},\overrightarrow{r}}$. Since v and v' agree on the propositions appearing in ψ, we have $\mathsf{v}'[\![\psi]\!] = \mathsf{F}$. Again, since ψ is a logical consequence of φ, it must be that $\mathsf{v}'[\![\varphi]\!] = \mathsf{F}$. Thus, no matter how the assignment to propositions $\overrightarrow{p}$ is changed from v, we will not be able to satisfy φ. This means that $\mathsf{v} \restriction_{\overrightarrow{q}} \notin M$. Thus, by our construction of η, $\mathsf{v}[\![\eta]\!] = \mathsf{F}$. This establishes that $\eta \models \psi$. □

Example 2.20 Let us look at a simple example that illustrates the construction of the interpolant in the proof of Theorem 2.19. Consider $\varphi = p \wedge (q_1 \vee q_2)$ and let $\psi = (q_1 \vee q_2 \vee r)$. It is easy to see that $\models \varphi \rightarrow \psi$. The set M constructed in the proof in this case would be $M = \{\mathsf{v}_{\mathsf{TT}}, \mathsf{v}_{\mathsf{TF}}, \mathsf{v}_{\mathsf{FT}}\}$, where v_{ij} is the function

$$\mathsf{v}_{ij}(q_1) = i \qquad \text{and} \qquad \mathsf{v}_{ij}(q_2) = j$$

Given M, the proof constructs the follow formula as interpolant.

$$\eta = (q_1 \wedge q_2) \vee (q_1 \wedge \neg q_2) \vee (\neg q_1 \wedge q_2).$$

2.3.2 Size of Interpolants

The interpolant η constructed in the proof of Theorem 2.19 is exponential in the number of common variables, and therefore, could be exponential in the size of the formulas φ and ψ. Can this be improved? Small interpolants can have a big impact in the contexts where interpolants are used, like in formal verification. Or can we prove that, in the worst case, the interpolant needs to be exponential in the size of the input formulas? Unfortunately, like many questions in theoretical computer science, this remains open and unresolved — we cannot prove or disprove that the construction in the proof of Theorem 2.19 is the best. However, in this section, we will show that it is unlikely that we can construct polynomial sized interpolants for all formulas. Or more precisely, we will show that resolving whether there exist polynomial sized interpolants for all formulas, is closely related to other open questions in complexity theory.

In order to present these results on the size of interpolants, we need to introduce new circuit complexity classes. Circuit complexity for a problem is based on a *non-uniform* model of computation. The idea is the following. Imagine you have the ability to choose a different algorithm for each input length; how much resource would you need? The "programs" for each input length are circuits, and different aspects of these circuits correspond to different computational resources one may care about. Running time in this context, roughly corresponds to circuit size. This leads us to analogs of P, NP, coNP in the context of non-uniform complexity, which are defined next.

Definition 2.21 (Circuits)

A *Boolean Circuit* C is a sequence of assignments $A_1, A_2, \ldots A_n$, where each A_i is one of the following forms.

$$\begin{aligned}
&P_i = \mathsf{F}\\
&P_i = \mathsf{T}\\
&P_i = ?\\
&P_i = P_j \wedge P_k,\ \ j, k < i\\
&P_i = P_j \vee P_k,\ \ j, k < i\\
&P_i = \neg P_j,\ \ j < i
\end{aligned}$$

where each P_i is a variable that appears on the left-hand side in only A_i. The size of such a circuit, denoted $|C|$, is n.

The variable P_i is said to be an *input variable* if the line A_i corresponding to it is of the form $P_i = ?$. The input variables of C will be denoted by $I(C)$. Given an assignment $\mathsf{v} : I(C) \to \{\mathsf{F}, \mathsf{T}\}$, the value of C under v is the value assigned to the variable P_n in the last line. There is a natural order on the variables (based on which line they are assigned a value) which also imposes an order on the input variables. Thus, an assignment $I(C) \to \{\mathsf{F}, \mathsf{T}\}$ can be thought of as a string x, where $x[i]$ is the value assigned to the ith input variable. Under such an interpretation, the value of C under string x, will be denoted by $C(x)$.

Example 2.22 Circuits and formulas are two ways to represent Boolean functions. It is instructive to see how they differ by looking at an example.

Consider the boolean function parity$(x_1, x_2, \ldots x_n)$ which computes whether the number of propositions in $\{x_1, x_2, \ldots x_n\}$ set to T is odd or even. For example, we could write down the formula for some simple cases.

$$\begin{aligned}\text{parity}(x_1,x_2) &= (x_1 \vee x_2) \wedge \neg(x_1 \wedge x_2)\\ \text{parity}(x_1,x_2,x_3) &= (((x_1 \vee x_2) \wedge \neg(x_1 \wedge x_2)) \vee x_3)\\ &\quad \wedge\neg(((x_1 \vee x_2) \wedge \neg(x_1 \wedge x_2)) \wedge x_3)\end{aligned}$$

More generally, we can inductively write down the formula for parity$(x_1, x_2, \dots x_n)$ by observing that the parity of $x_1, x_2, \dots x_n$ is odd iff either (a) the parity of $x_1, \dots x_{n-1}$ is even and x_n is T, or (b) parity of $x_1, \dots x_{n-1}$ is odd and x_n is F. This results in the following definition.

$$\begin{aligned}\text{parity}(x_1,x_2,\dots x_{n-1},x_n) &= (\text{parity}(x_1,x_2,\dots x_{n-1}) \vee x_n)\\ &\quad \wedge\neg(\text{parity}(x_1,x_2,\dots x_{n-1}) \wedge x_n)\end{aligned}$$

Observe that the formula we wrote down for parity(x_1, x_2, x_3) is based on exactly this definition. Observe that the size of the formula parity$(x_1, \dots x_n)$ is double the size of parity$(x_1, \dots x_{n-1})$. This means that the size of parity for n arguments is $O(2^n)$.

A circuit for the same formula does not grow as rapidly. This is because it can "reuse" previous computed answers. Let us look at the circuit corresponding to the formula parity(x_1, x_2, x_3) we wrote above.

$$\begin{aligned}&x_1 = ?\\ &x_2 = ?\\ &x_3 = ?\\ &P_4 = x_1 \vee x_2\\ &P_5 = x_1 \wedge x_2\\ &P_6 = \neg P_5\\ &P_7 = P_4 \wedge P_6\\ &P_8 = P_7 \vee x_3\\ &P_9 = P_7 \wedge x_3\\ &P_{10} = \neg P_9\\ &P_{11} = P_8 \wedge P_{10}\end{aligned}$$

Notice that variable P_7 stores parity(x_1, x_2) and it is reused without paying an extra cost. More generally, if C_{n-1} is the circuit computing parity$(x_1, \dots x_{n-1})$ with last line $P_{k_{n-1}}$, the circuit C_n computing parity$(x_1, \dots x_n)$ is given by

$$\begin{aligned}&C_{n-1}\\ &x_n = ?\\ &Q_1 = P_{k_{n-1}} \vee x_n\\ &Q_2 = P_{k_{n-1}} \wedge x_n\\ &Q_3 = \neg Q_2\\ &P_{k_n} = Q_1 \wedge Q_3\end{aligned}$$

Now if $|C_n| = |C_{n-1}| + 5$. Therefore, in general, we get $|C_n| = O(n)$. Thus the circuit for the same syntactic formula can be exponentially smaller.

Definition 2.23 A language $A \subseteq \{0,1\}^*$ is said to be in $\mathsf{P/poly}$ iff there are constants $c, k \in \mathbb{N}$, and a (infinite) family of circuits $\{C_i\}_{i \in \mathbb{N}}$ such that (a) for every n, $|C_n| \leq cn^k$, and (b) for every x, $x \in A$ iff $C_{|x|}(x) = \mathsf{T}$.

A language $A \subseteq \{0,1\}^*$ is said to be in $\mathsf{NP/poly}$ ($\mathsf{coNP/poly}$) iff there are constants $c, k \in \mathbb{N}$, and a (infinite) family of circuits $\{C_i\}_{i \in \mathbb{N}}$ such that (a) for every n, $|C_n| \leq cn^k$, and (b) for every x, $x \in A$ iff for *some* (*all*) r, $C_{|x|}(x, r) = \mathsf{T}$.

One can think of $\mathsf{P/poly}$ (or $\mathsf{NP/poly}$ or $\mathsf{coNP/poly}$) as the collection of problems that be solved in polynomial time (or nondeterministic polynomial time or co-nondeterministic polynomial time) *given* a polynomially long *advice string*, namely, the description of the appropriate circuit. Just like in the case of (uniform/regular) complexity classes where it is open whether $\mathsf{P} \overset{?}{=} \mathsf{NP} \cap \mathsf{coNP}$, the same question is open in the non-uniform context as well. That is, a long standing open problem is whether $\mathsf{P/poly} \overset{?}{=} \mathsf{NP/poly} \cap \mathsf{coNP/poly}$.

Mundici's theorem says that establishing the existence of interpolants with polynomial circuit representation for all pairs of formulas, is equivalent to resolving the $\mathsf{P/poly}$ versus $\mathsf{NP/poly} \cap \mathsf{coNP/poly}$ question. Thus, it is likely to be difficult to establish.

Theorem 2.24 (Mundici)

If for any φ and ψ such that $\varphi \models \psi$ there is an interpolant whose circuit size is polynomial in φ and ψ then

$$\mathsf{P/poly} = \mathsf{NP/poly} \cap \mathsf{coNP/poly}$$

Proof Consider a problem $L \in \mathsf{NP/poly} \cap \mathsf{coNP/poly}$. Thus, there are families of circuits $\{A_n\}_{n \in \mathbb{N}}$ and $\{B_n\}_{n \in \mathbb{N}}$ such that for each n, A_n and B_n have size bounded by a polynomial function of n and for any binary string w, $w \in L$ iff $\exists p.\ A_{|w|}(p, w) = \mathsf{T}$ iff $\forall r.\ B_{|w|}(w, r) = \mathsf{T}$. These observations are just a consequence of $L \in \mathsf{NP/poly}$ and $L \in \mathsf{coNP/poly}$.

Based on the previous paragraph, we have, for any n, if for some assignment of values to $\overrightarrow{q}$, if $\exists \overrightarrow{p}.\ A_n(\overrightarrow{p}, \overrightarrow{q})$ holds then it also the case that $\forall \overrightarrow{r}.\ B_n(\overrightarrow{q}, \overrightarrow{r})$ holds. Therefore, we have $A_n(\overrightarrow{p}, \overrightarrow{q}) \models B_n(\overrightarrow{q}, \overrightarrow{r})$. By our assumption on interpolants, we have an interpolant (as a circuit) $C_n(\overrightarrow{q})$ such that $|C_n|$ is bounded by a polynomial in $|A_n|$ and $|B_n|$. Since $|A_n|$ and $|B_n|$ are bounded by polynomials in n, we have $|C_n|$ is bounded by a polynomial in n. Further, since C_n is an interpolant, we have if $\exists \overrightarrow{p}.\ A_n(\overrightarrow{p}, \overrightarrow{q})$ holds then $C_n(\overrightarrow{q})$ holds, and if $C_n(\overrightarrow{q})$ holds then $\forall \overrightarrow{r}.\ B_n(\overrightarrow{q}, \overrightarrow{r})$ holds. Thus, $\{C_n\}$ is a family of polynomially sized circuits deciding L, and thereby demonstrating that $L \in \mathsf{P/poly}$. □

2.3.3 Interpolants from Refutations

Constructing small interpolants for all formulas is likely to be difficult. However, it turns out that, if a formula $A(\overrightarrow{p}, \overrightarrow{q}) \rightarrow B(\overrightarrow{q}, \overrightarrow{r})$ has a short proof in some proof systems, then the proof can be used to construct an interpolant, whose size is propositional to the original proof. One proof system that admits such a result is resolution.

Theorem 2.25 *Let the collection of clauses* $\Gamma = \{A_i(\overrightarrow{p},\overrightarrow{q})\}_{i=1}^{k} \cup \{B_j(\overrightarrow{q},\overrightarrow{r})\}_{j=1}^{\ell}$ *have a resolution refutation of length n. Then there is a circuit* $C(\overrightarrow{q})$ *such that*

$$\bigwedge_i A_i(\overrightarrow{p},\overrightarrow{q}) \models C(\overrightarrow{q}) \text{ and } C(\overrightarrow{q}) \wedge \bigwedge_j B_j(\overrightarrow{q},\overrightarrow{r}) \text{ is unsatisfiable.}$$

Further $|C|$ *is* $O(n)$.

Proof Since $\Gamma = \{A_i(\overrightarrow{p},\overrightarrow{q})\}_i \cup \{B_j(\overrightarrow{q},\overrightarrow{r})\}_j$ is unsatisfiable, every truth valuation v fails to satisfy at least one of the $A_i(\overrightarrow{p},\overrightarrow{q})$ or $B_j(\overrightarrow{q},\overrightarrow{r})$. We can also restate the properties of an interpolant C as

$$\models \neg C(\overrightarrow{q}) \to \neg \bigwedge_i A_i(\overrightarrow{p},\overrightarrow{q}) \text{ and}$$
$$\models C(\overrightarrow{q}) \to \neg \bigwedge_j B_j(\overrightarrow{q},\overrightarrow{r})$$

Thus, $C(\overrightarrow{q})$ can be thought of as a way of labelling truth valuations: those labelled F will not satisfy some clause $A_i(\overrightarrow{p},\overrightarrow{q})$ and those labelled T will not satisfy some $B_j(\overrightarrow{q},\overrightarrow{r})$.

The structure of the proof will be as follows. Let $\psi_1, \psi_2, \ldots \psi_n$ be a resolution refutation of Γ. Let the set of common variables $\overrightarrow{q} = \{q_1, \ldots q_m\}$. Our circuit for the interpolant will be a sequence of the form $q_1 = ?, q_2 = ?, \ldots q_m = ?, P_1 = E_1, P_2 = E_2, \ldots P_n = E_n$ — the first m lines asserting that q_i's are variables, and then having one line $P_i = E_i$ corresponding to each line ψ_i of our refutation. It will be convenient to consider expressions E_i on the right hand side that have more than one logical connective. We will find it convenient to talk about the "circuit corresponding to a line ψ_i in the refutation". What we mean by this is look at the sequence of assignments upto (and including) line $P_i = E_i$; we will denote this as circuit C_i. The value of C_i (with respect to a truth assignment) will simply be the value variable P_i gets when we compute this circuit.

As mentioned above, we will construct the circuit line by line, corresponding to the refutation. With each line ψ_i in the refutation, we can associate a set of truth assignments, namely those that *do not* satisfy ψ_i, i.e., $M_i = \{\mathsf{v} \mid \mathsf{v}[\![\psi_i]\!] = \mathsf{F}\}$. The invariant we will maintain as we build the circuit line by line, is that C_i "correctly labels" assignments belonging to M_i. That is, for $\mathsf{v} \in M_i$, if $C_i(\mathsf{v}) = \mathsf{F}$ then v does not satisfy some clause $A_i(\overrightarrow{p},\overrightarrow{q})$ and if $C_i(\mathsf{v}) = \mathsf{T}$ then v does not satisfy some clause $B_j(\overrightarrow{q},\overrightarrow{r})$. Notice that the last line $\psi_n = \{\}$, and so M_n is the set of all truth assignments. Thus, if the invariant is maintained, C_n will indeed be an interpolant because it will "correctly" label all assignments.

Let us now describe how we construct the circuit. For each line ψ_e, we will add a line $P_e = E_e$. What E_e is will depend on the justification for the line ψ_e in the refutation. Let us begin with the cases when ψ_e is a clause belonging to Γ. There are two cases to consider.

- Suppose $\psi_e \in \{A_i(\overrightarrow{p}, \overrightarrow{q})\}_{i=1}^k$. Observe that any $\mathsf{v} \in M_e$ does not satisfy a clause in $\{A_i(\overrightarrow{p}, \overrightarrow{q})\}_{i=1}^k$ and could be safely labeled F. Thus, the line corresponding to ψ_e will be $P_e = \mathsf{F}$.
- If $\psi_e \in \{B_j(\overrightarrow{q}, \overrightarrow{r})\}_{j=1}^{\ell}$, then each $\mathsf{v} \in M_e$ could be labeled T because it does not satisfy $\psi_e \in \{B_j(\overrightarrow{q}, \overrightarrow{r})\}_{j=1}^{\ell}$. Thus, we have $P_e = \mathsf{T}$.

The next cases to consider are when ψ_e is a resolvent of two clauses. So let $\psi_e = \rho_1 \cup \rho_2$ and let it be the resolvent of clauses ψ_a and ψ_b in the refutation. We need to consider different cases based on the proposition with respect to which ψ_e is a resolvent. Let us begin by considering the case when the proposition being resolved is one of the common variables. Without loss of generality, let us take $\psi_a = \rho_1 \cup \{q\}$ and $\psi_b = \rho_2 \cup \{\neg q\}$. Consider an arbitrary assignment $\mathsf{v} \in M_e$, i.e., $\mathsf{v}[\![\psi_e]\!] = \mathsf{F}$. From the soundness of the resolution proof rule, we know that either $\mathsf{v} \in M_a$ or $\mathsf{v} \in M_b$. If $\mathsf{v}(q) = \mathsf{F}$ then $\mathsf{v} \in M_a$; it may also, in addition, be the case that $\mathsf{v} \in M_b$, but that is unimportant. We could label such assignments in the same manner as the labeling corresponding to line ψ_a, i.e., as per the value of variable P_a. Similarly, if $\mathsf{v}(q) = \mathsf{T}$ then $\mathsf{v} \in M_b$ and so it can be labeled in the same manner as P_b. This gives us that the line corresponding to ψ_e in this case should be $P_e = (\neg q \wedge P_a) \vee (q \wedge P_b)$.

Let us now consider the case when ψ_e is a resolvent of ψ_a and ψ_b, but the resolution step is taken with respect to a proposition (say s) that is either in $\overrightarrow{p}$ or in $\overrightarrow{r}$. Once again any $\mathsf{v} \in M_e$ must also belong to $M_a \cup M_b$, but now since s is not in $\overrightarrow{q}$ it cannot be explicitly mentioned in the line $P_e = E_e$, as we did in the previous case. Again, without loss of generality, let us assume that $\psi_e = \rho_1 \cup \rho_2$, $\psi_a = \rho_1 \cup \{s\}$, $\psi_b = \rho_2 \cup \{\neg s\}$. Let v' be the assignment that is identical to v, except that it flips the assignment to s. Without loss of generality, we can assume that $\mathsf{v}(s) = \mathsf{F}$ and $\mathsf{v}'(s) = \mathsf{T}$, and for all other propositions t, $\mathsf{v}(t) = \mathsf{v}'(t)$. Observe that $\mathsf{v} \in M_a$ and $\mathsf{v}' \in M_b$. Further, $C_a(\mathsf{v}) = C_a(\mathsf{v}')$ and $C_b(\mathsf{v}) = C_b(\mathsf{v}')$; this is because C_a and C_b do not mention s. Let us consider two cases based on whether $s \in \overrightarrow{p}$ or $s \in \overrightarrow{r}$.

- Suppose $s \in \overrightarrow{p}$. Observe that in this case, for any j, $\mathsf{v}[\![B_j]\!] = \mathsf{v}'[\![B_j]\!]$, because s does not appear in B_j. Now, since C_a and C_b satisfy our invariant, we have, if $C_a(\mathsf{v}) = \mathsf{T}(= C_a(\mathsf{v}'))$ then for some j, $\mathsf{v}[\![B_j]\!] = \mathsf{F}(= \mathsf{v}'[\![B_j]\!])$. Similarly, if $C_b(\mathsf{v}') = \mathsf{T}(= C_b(\mathsf{v}))$ then for some j, $\mathsf{v}'[\![B_j]\!] = \mathsf{F}(= \mathsf{v}[\![B_j]\!])$. Thus, if either C_a or C_b label v, v' by T, then C_e must do the same. Otherwise, both C_a and C_b label v and v' as F, and this common label is correct as per our invariant. Therefore, we have $P_e = P_a \vee P_b$ in this case.
- Now let us consider $s \in \overrightarrow{q}$. In this case, for any i, $\mathsf{v}[\![A_i]\!] = \mathsf{v}'[\![A_i]\!]$. Using an argument similar to the previous case, we can argue that if either C_a or C_b label v, v' by F then C_e must do the same. Otherwise, C_a and C_b agree on the label, and that is indeed the correct label. Therefore, dually, in this case we have, $P_e = P_a \wedge P_b$.

The proof that the invariant is maintained follows inductively, from the arguments we have made for each case. Thus, C_n is indeed the interpolant. It is worth noting that in C_n we have a sequence of initial assignments $q_1 = ?, q_2 = ?, \ldots q_m = ?$ that assert that $\overrightarrow{q}$ are input variables. This seems to suggest that the size of C_n also depends on

$|\overrightarrow{q}|$ and hence on Γ. However, instead of asserting that all variables in $\overrightarrow{q}$ are input variables, we could assert only those variables in $\overrightarrow{q}$ that appear in the refutation. Thus, $|C_n|$ is indeed linear in the size of the refutation. □

Example 2.26 Let us look at an example to see how an interpolant can be constructed from a refutation. Consider the set of clauses

$$\Gamma = \{\overbrace{\{p,q\},\{\neg p,r\}}^{A},\overbrace{\{\neg q,r\},\{\neg r\}}^{B}\}.$$

Here p is a proposition that only appears in the A-clauses, and q and r are propositions that appear in both A and B clauses. Γ is unsatisfiable, and we are looking for an interpolant that only mentions the common variables q, r. Below we have the refutation (on the left) alongside the circuit for the interpolant (on the right) as per the proof of Theorem 2.25.

	$q = ?$
	$r = ?$
$\{\neg p, r\}$	$P_1 = 0$
$\{\neg r\}$	$P_2 = 1$
$\{\neg p\}$	$P_3 = (\neg r \wedge P_1) \vee (r \wedge P_2)$
$\{\neg q, r\}$	$P_4 = 1$
$\{\neg q\}$	$P_5 = (\neg r \wedge P_4) \vee (r \wedge P_2)$
$\{p, q\}$	$P_6 = 0$
$\{q\}$	$P_7 = P_3 \vee P_6$
$\{\}$	$P_8 = (\neg q \wedge P_7) \vee (q \wedge P_5)$

Observations like Theorem 2.25 have also been established for other proof systems of propositional logic. That is, in these proof systems, a short proof for a fact can be converted into a construction of a small interpolant. Theorem 2.25 can be strengthened for certain special sets of clauses — one can show that in certain special cases, not only is the interpolant small, but it is also *monotonic*.

Definition 2.27 (Monotone Circuits)

A *monotone circuit* C is one where there are no assignments of the form $P_i = \neg P_j$.

A monotone circuit enjoys the following monotonic property. Let us say $\mathsf{v}_1 \leq \mathsf{v}_2$ if for all proposition p, if $\mathsf{v}_1(p) = \mathsf{T}$ then $\mathsf{v}_2(p) = \mathsf{T}$, i.e., v_2 sets at least as many propositions to T as v. The value of a monotonic circuit with respect to this ordering on assignments is monotonic. In other words, if C is monotone, then for any $\mathsf{v}_1, \mathsf{v}_2$ such that $\mathsf{v}_1 \leq \mathsf{v}_2$, we have $C(\mathsf{v}_1) = \mathsf{T}$ implies $C(\mathsf{v}_2) = \mathsf{T}$.

Theorem 2.28 *Let the collection of clauses* $\Gamma = \{A_i(\overrightarrow{p},\overrightarrow{q})\}_{i=1}^{k} \cup \{B_j(\overrightarrow{q},\overrightarrow{r})\}_{j=1}^{\ell}$ *have a resolution refutation of length n. Further assume that either* $\overrightarrow{q}$ *occur only positively in* A_i*s or* $\overrightarrow{q}$ *occur only negatively in* B_j*s. Then there is a* monotone *circuit* $C(\overrightarrow{q})$ *such that*

$$\bigwedge_i A_i(\overrightarrow{p}, \overrightarrow{q}) \models C(\overrightarrow{q}) \text{ and } C(\overrightarrow{q}) \wedge \bigwedge_j B_j(\overrightarrow{q}, \overrightarrow{r}) \text{ is unsatisfiable.}$$

In addition, $|C|$ is $O(n)$.

Proof The construction of the interpolant is identical to that in the proof of Theorem 2.25. All cases in that proof go through except for the case when the line ψ_e is a resolvent with respect to proposition $q \in \overrightarrow{q}$, i.e., the common variables. In this case, in the proof of Theorem 2.25, the circuit was $P_e = (\neg q \wedge P_a) \vee (q \wedge P_b)$, where ψ_e is the resolvent of lines $\psi_a = \rho_1 \cup \{p\}$ and $\psi_b = \rho_2 \cup \{\neg q\}$. This is not monotonic because if the use of $\neg q$. To prove our result, we need to change the circuit in this case. We will change it by forcing it to be monotone in the most naïve way — we will remove the offending $\neg q$ and write $P_e = P_a \vee (q \wedge P_b)$.

The resulting construction is correct, but the inductive argument using the invariant from Theorem 2.25 does not go through! Let us see what the problem is. Recall, that for any line ψ_e, we defined the set $M_e = \{\mathsf{v} \mid \mathsf{v}[\![\psi_e]\!] = \mathsf{F}\}$. The invariant we proved in Theorem 2.25 was for any valuation $\mathsf{v} \in M_e$, we have

- If $\mathsf{v}[\![C_e]\!] = \mathsf{F}$ then $\mathsf{v}[\![A_i]\!] = \mathsf{F}$ for some i, and
- If $\mathsf{v}[\![C_e]\!] = \mathsf{T}$ then $\mathsf{v}[\![B_j]\!] = \mathsf{F}$ for some j.

Let us try to prove this invariant by induction as in the proof of Theorem 2.25. Consider the case that we just changed, i.e., of a resolvent with respect to a common variable. So $\psi_e = \rho_1 \cup \rho_2$ is the resolvent of lines $\psi_a = \rho_1 \cup \{q\}$ and $\psi_b = \rho_2 \cup \{\neg q\}$. And we have, $P_e = P_a \vee (q \wedge P_b)$. Consider a valuation $\mathsf{v} \in M_e$. The problem with carrying out this inductive proof occurs when $\mathsf{v}(q) = \mathsf{T}$; the other case of $\mathsf{v}(q) = \mathsf{F}$ goes through rather simply. Then $\mathsf{v} \in M_b$. Now if $C_a(\mathsf{v}) = \mathsf{F}$ or $C_b(\mathsf{v}) = \mathsf{T}$ then $C_e(\mathsf{v}) = C_b(\mathsf{v})$ and correctness follows from the inductive assumptions on C_b. The problem occurs when $C_a(\mathsf{v}) = \mathsf{T}$ and $C_b(\mathsf{v}) = \mathsf{F}$.

The way to fix the problem is to prove a *stronger* invariant. In our old invariant, we proved that our circuit for line e was correct on the truth assignments in $M_e = \{\mathsf{v} \mid \mathsf{v}[\![\psi_e]\!] = \mathsf{F}\}$. Our strengthening will show that the circuit for line e is correct on a larger set of truth assignments. Depending on whether we consider the case when $\overrightarrow{q}$ appears only positively in $\{A_i(\overrightarrow{p}, \overrightarrow{q})\}_i$ or the case when $\overrightarrow{q}$ occurs only negatively in $\{B_j(\overrightarrow{q}, \overrightarrow{r})\}_j$, the invariant (and the proof) is slightly different. We will present the proof only for the case when $\overrightarrow{q}$ occurs negatively in $\{B_j(\overrightarrow{q}, \overrightarrow{r})\}_j$. We will state the modified invariant for the other case, but leave the details to be filled out by the reader.

To describe the invariant in the case when $\overrightarrow{q}$ occurs negatively in $\{B_j(\overrightarrow{q}, \overrightarrow{r})\}_j$, we need to introduce some notation. For a clause ψ, $\psi \restriction_{\overrightarrow{p},\overrightarrow{q}}$ be the clause obtained by removing all literals involving propositions in $\overrightarrow{r}$. On the other hand, $\psi \restriction_{\neg\overrightarrow{q},\overrightarrow{r}}$ is the clause obtained from ψ by removing all literals of proposition $\overrightarrow{p}$ as well as all positive literals of $\overrightarrow{q}$. Our stronger invariant will be for every valuation v

- if $\mathsf{v}[\![\psi \restriction_{\overrightarrow{p},\overrightarrow{q}}]\!] = \mathsf{F}$ and $C_e(\mathsf{v}) = \mathsf{F}$ then $\mathsf{v}[\![A_i]\!] = \mathsf{F}$ for some i, and
- if $\mathsf{v}[\![\psi \restriction_{\neg\overrightarrow{q},\overrightarrow{r}}]\!] = \mathsf{F}$ and $C_e(\mathsf{v}) = \mathsf{T}$ then $\mathsf{v}[\![B_j]\!] = \mathsf{F}$ for some j.

Notice that since $\{\} \restriction_{\overrightarrow{p},\overrightarrow{q}} = \{\} \restriction_{\neg\overrightarrow{q},\overrightarrow{r}} = \{\}$, proving this new invariant guarantees that C_n (where n is length of the resolution refutation) is an interpolant.

We now argue that the stronger invariant holds for the new construction. We consider each case in order.

$\psi_e \in \{A_i\}_i$: In this case, we have $\psi_e \upharpoonright_{\overrightarrow{p},\overrightarrow{q}} = \psi_e$ and $P_e = \mathsf{F}$. The invariant, therefore, holds.

$\psi_e \in \{B_j\}_j$: Again we have $\psi_e \upharpoonright_{-\overrightarrow{q},\overrightarrow{r}} = \psi_e$. Since $P_e = \mathsf{T}$, the invariant holds.

Resolvent w.r.t $\overrightarrow{q}$: Let $\psi_e = \rho_1 \cup \rho_2$ be the resolvent of lines $\psi_a = \rho_1 \cup \{q\}$ and $\psi_b = \rho_2 \cup \{\neg q\}$. Recall we have $P_e = P_a \vee (q \wedge P_b)$.

1. Consider v such that $\mathsf{v}[\![\psi_e \upharpoonright_{\overrightarrow{p},\overrightarrow{q}}]\!] = \mathsf{F}$ and $C_e(\mathsf{v}) = \mathsf{F}$. In this case, if $\mathsf{v}(q) = \mathsf{T}$ then $\mathsf{v}[\![\psi_b \upharpoonright_{\overrightarrow{p},\overrightarrow{q}}]\!] = \mathsf{v}[\![\psi_b]\!] = \mathsf{F}$. Also, since $C_e(\mathsf{v}) = \mathsf{F}$, it must be that $C_b(\mathsf{v}) = \mathsf{F}$, and so correctness follows by induction. On the other hand, if $\mathsf{v}(q) = \mathsf{F}$ then $\mathsf{v}[\![\psi_a \upharpoonright_{\overrightarrow{p},\overrightarrow{q}}]\!] = \mathsf{v}[\![\psi_a]\!] = \mathsf{F}$. Also, $C_a(\mathsf{v}) = \mathsf{F}$ and correctness follows by induction.
2. Consider v such that $v[\![\psi_e \upharpoonright_{-\overrightarrow{q},\overrightarrow{r}}]\!] = \mathsf{F}$ and $C_e(\mathsf{v}) = \mathsf{T}$. If $C_a(\mathsf{v}) = \mathsf{T}$ then since $\mathsf{v}[\![\psi_a \upharpoonright_{-\overrightarrow{q},\overrightarrow{r}}]\!] = \mathsf{v}[\![\rho_1 \upharpoonright_{-\overrightarrow{q},\overrightarrow{r}}]\!] = \mathsf{F}$, the invariant follows by induction. Notice, how the stronger invariant helped the proof go through in this case which was problematic before. On the other hand, if $\mathsf{v}[\![q \wedge C_b]\!] = \mathsf{T}$ then $\mathsf{v}(q) = \mathsf{T}$. So $\mathsf{v}[\![\psi_b \upharpoonright_{-\overrightarrow{q},\overrightarrow{r}}]\!] = \mathsf{F}$ and then invariant holds by induction.

Resolvent w.r.t. $\overrightarrow{p}$: Let $\psi_e = \rho_1 \cup \rho_2$ be the resolvent of lines $\psi_a = \rho_1 \cup \{p\}$ and $\psi_b = \rho_2 \cup \{\neg p\}$. Recall $P_e = P_a \vee P_b$.

1. Suppose $C_e(\mathsf{v}) = \mathsf{F}$ and $\mathsf{v}[\![\psi_e \upharpoonright_{\overrightarrow{p},\overrightarrow{q}}]\!] = \mathsf{F}$. Then we know $C_a(\mathsf{v}) = C_b(\mathsf{v}) = \mathsf{F}$. Further either $\mathsf{v}[\![\psi_a]\!] = \mathsf{F}$ or $\mathsf{v}[\![\psi_b]\!] = \mathsf{F}$. Thus, correctness of construction by induction.
2. Suppose $\mathsf{v}[\![\psi_e \upharpoonright_{-\overrightarrow{q},\overrightarrow{r}}]\!] = \mathsf{F}$ and $C_e(\mathsf{v}) = \mathsf{T}$. Now $\psi_e \upharpoonright_{-\overrightarrow{q},\overrightarrow{r}} = \psi_a \upharpoonright_{-\overrightarrow{q},\overrightarrow{r}} \cup \psi_b \upharpoonright_{-\overrightarrow{q},\overrightarrow{r}}$, and so $\mathsf{v}[\![\psi_a \upharpoonright_{-\overrightarrow{q},\overrightarrow{r}}]\!] = \mathsf{v}[\![\psi_b \upharpoonright_{-\overrightarrow{q},\overrightarrow{r}}]\!] = \mathsf{F}$. Further since $C_e(\mathsf{v}) = \mathsf{T}$, either $C_a(\mathsf{v}) = \mathsf{T}$ or $C_b(\mathsf{v}) = \mathsf{T}$. So correctness follows by induction.

Resolvent w.r.t. $\overrightarrow{r}$: Proof similar to previous case.

The proof of correctness when $\overrightarrow{q}$ appears positively in $\{A_i(\overrightarrow{p}, \overrightarrow{q})\}_i$ is similar, though the invariant is slightly different. For a clause ψ, take $\psi \upharpoonright_{\overrightarrow{p},+\overrightarrow{q}}$ to be the clause obtained by removing literals of $\overrightarrow{r}$ and negative literals of $\overrightarrow{q}$. In addition, $\psi \upharpoonright_{\overrightarrow{q},\overrightarrow{r}}$ is the clause obtained by removing literals of $\overrightarrow{p}$. The invariant we will prove about the construction is, for every v,

- if $\mathsf{v}[\![\psi \upharpoonright_{\overrightarrow{p},+\overrightarrow{q}}]\!] = \mathsf{F}$ and $C_e(\mathsf{v}) = \mathsf{F}$ then $\mathsf{v}[\![A_i]\!] = \mathsf{F}$ for some i, and
- if $\mathsf{v}[\![\psi \upharpoonright_{\overrightarrow{q},\overrightarrow{r}}]\!] = \mathsf{F}$ and $C_e(\mathsf{v}) = \mathsf{T}$ then $\mathsf{v}[\![B_j]\!] = \mathsf{F}$ for some j.

The proof is similar and skipped. □

Example 2.29 Let us consider the set of clauses Γ from Example 2.26.

$$\Gamma = \{\overbrace{\{p, q\}, \{\neg p, r\}}^{A}, \overbrace{\{\neg q, r\}, \{\neg r\}}^{B}\}.$$

Notice that the common propositions, q and r, appear only positively in A. The refutation alongside the interpolant construction is as follows.

	$q = ?$
	$r = ?$
$\{\neg p, r\}$	$P_1 = 0$
$\{\neg r\}$	$P_2 = 1$
$\{\neg p\}$	$P_3 = P_1 \vee (r \wedge P_2)$
$\{\neg q, r\}$	$P_4 = 1$
$\{\neg q\}$	$P_5 = P_4 \vee (r \wedge P_2)$
$\{p, q\}$	$P_6 = 0$
$\{q\}$	$P_7 = P_3 \vee P_6$
$\{\}$	$P_8 = P_7 \vee (q \wedge P_5)$

2.3.4 Lower bounds on Resolution Refutations

The results in Sect. 2.2 connecting resolution refutation lengths and size of interpolants, allows one to prove lower bounds on the length of resolution refutations for formulas. In particular we can show that there are sets of clauses Γ for which the shortest resolution refutations are exponential in the size of Γ. Thus, not every unsatisfiable formula has a short proof in resolution. The specific example we consider relates to cliques in graphs and their coloring. Let us recall these classical problems.

Recall that in Proposition 1.35, we showed that determining if a graph is k-colorable can be reduced to checking the satisfiability of a set of formulas. More specifically, let us fix the graph $G = (V, E)$ to have n vertices. Any such graph can be represented by an assignment to propositions $\{q_{uv} \mid u, v \in \{1, 2, \ldots n\}\}$, with the interpretation that $(u, v) \in E$ iff q_{uv} is set to T. Using r_{ui} to denote "vertex u has color i", there is a set of clauses $\mathrm{color}_{n,k}(\overrightarrow{q}, \overrightarrow{r})$ such that $\mathsf{v} \models \mathrm{color}_{n,k}(\overrightarrow{q}, \overrightarrow{r})$ iff the graph (over n vertices) represented by $\mathsf{v} \upharpoonright_{\overrightarrow{q}}$ has a k-coloring given by $\mathsf{v} \upharpoonright_{\overrightarrow{r}}$. The set $\mathrm{color}_{n,k}$ is almost identical to the construction given in the proof of Proposition 1.35. The only difference is that we will use the clause $\neg q_{uv} \vee \neg r_{ui} \vee \neg r_{vi}$, for every pair of vertices $u, v \in \{1, 2, \ldots n\}$ and color $i \in \{1, 2, \ldots k\}$, instead of $\neg r_{ui} \vee \neg r_{vi}$ for every edge (u, v) as given in Proposition 1.35. Note that $\mathrm{color}_{n,k}$ has $O(n^2 k + nk^2)$ clauses.

Whether a graph is k-colorable is related to the presence of graph structures called *cliques*.

Definition 2.30 A k-clique in a graph $G = (V, E)$ is a subset $U \subseteq V$ such that $|U| = k$ and for every $u, v \in U$, with $u \neq v$, we have $(u, v) \in E$.

Like graph coloring, the problem of determining if a graph has a k-clique can be reduced to satisfiability.

Proposition 2.31 *For any n, k, there is a set of $O(n^2k^2)$ clauses $clique_{n,k}(\overrightarrow{p}, \overrightarrow{q})$ such that $\mathsf{v} \models clique_{n,k}(\overrightarrow{p}, \overrightarrow{q})$ iff the graph represented by $\mathsf{v} \upharpoonright_{\overrightarrow{q}}$ has a k-clique given by $\mathsf{v} \upharpoonright_{\overrightarrow{p}}$*

Proof The proof of this observation is similar to that of Proposition 1.35. We will introduce propositions that encode the k-clique, and clauses will specify constraints that characterize properties of a k-clique. Let proposition p_{iu}, for $i \in \{1, \ldots k\}$ and $u \in \{1, \ldots n\}$, denote that "the ith vertex in clique is u". Then $\text{clique}_{n,k}(\overrightarrow{p}, \overrightarrow{q})$ is the following set of clauses.

- For each $1 \leq i \leq k$, the clause $p_{i1} \vee p_{i2} \vee \cdots \vee p_{in}$. These clauses capture the constraint that the ith vertex of the clique must be among $\{1, \ldots n\}$.
- For each $1 \leq i \leq k$, and $1 \leq u, v \leq n$ such that $u \neq v$, the clause $\neg p_{iu} \vee \neg p_{iv}$. Intuitively, this says that the ith vertex of the clique can be at most one vertex.
- For each $1 \leq i, j \leq k$ with $i \neq j$, and $1 \leq u \leq n$, the clause $\neg p_{iu} \vee \neg p_{ju}$. These clauses say that the ith and jth vertex of the clique cannot be the same vertex u.
- For each $1 \leq i, j \leq k$ and $1 \leq u, v \leq n$ with $u \neq v$, we have the clause $\neg p_{iu} \vee \neg p_{jv} \vee q_{uv}$. These clauses together say that if u, v are vertices in the clique then they have an edge between them.

The proof that these clauses satisfy the proposition is left to the reader. □

Observe that if a graph G has a k-clique then it cannot be colored using $k - 1$ colors. This is because each of the vertices in the clique must get different colors. Thus a graph with a k-clique needs at least k colors. This leads us to the following observation.

Proposition 2.32 *For any n, k, $clique_{n,k}(\overrightarrow{p}, \overrightarrow{q}) \cup color_{n,k-1}(\overrightarrow{q}, \overrightarrow{r})$ is unsatisfiable.*

Proof A satisfying assignment for $\text{clique}_{n,k}(\overrightarrow{p}, \overrightarrow{q}) \cup \text{color}_{n,k-1}(\overrightarrow{q}, \overrightarrow{r})$ is a graph encoded by $\overrightarrow{q}$ that has k-clique (identified by $\overrightarrow{p}$) and can be colored using $k - 1$ colors (with the coloring encoded by $\overrightarrow{r}$). This is clearly impossible. □

Since $\text{clique}_{n,k}(\overrightarrow{p}, \overrightarrow{q}) \cup \text{color}_{n,k-1}(\overrightarrow{q}, \overrightarrow{r})$ is unsatisfiable, it must have a resolution refutation. How long is its refutation? Our goal will be to prove that this is exponential in n. Since the size of $\text{clique}_{n,k}(\overrightarrow{p}, \overrightarrow{q}) \cup \text{color}_{n,k-1}(\overrightarrow{q}, \overrightarrow{r})$ itself is polynomial in n, this would be an example that has a "long" proof. In order to establish this result, we present a celebrated result in circuit complexity whose proof can be found in textbooks like [**?**].

Theorem 2.33 (Razborov, Alon-Bopanna)

Any monotone circuit that evaluates to T on n-vertex graphs containing a k-clique and evaluates to F on n-vertex graphs that are $k - 1$ colorable must have size at least $n^{\Omega(\sqrt{k})}$, when $k \leq n^{\frac{1}{4}}$.

Theorem 2.33 combined with Theorem 2.28 gives us the desired lower bound on proof lengths.

Theorem 2.34 *Any resolution refutation of $clique_{n,k}(\overrightarrow{p}, \overrightarrow{q}) \cup color_{n,k-1}(\overrightarrow{q}, \overrightarrow{r})$ must have length at least $n^{\Omega(\sqrt{k})}$, when $k \leq n^{\frac{1}{4}}$.*

Proof Let there be a resolution refutation of length ℓ. Observe that $\overrightarrow{q}$ appears only positively in $\mathrm{clique}_{n,k}(\overrightarrow{p}, \overrightarrow{q})$ and only negatively in $\mathrm{color}_{n,k}(\overrightarrow{q}, \overrightarrow{r})$. Thus, $\mathrm{clique}_{n,k}(\overrightarrow{p}, \overrightarrow{q}) \cup \mathrm{color}_{n,k-1}(\overrightarrow{q}, \overrightarrow{r})$ satisfies the conditions of Theorem 2.28, and so there is a monotone interpolant of size $O(\ell)$. The interpolant is a monotone circuit satisfying conditions of Theorem 2.33. Thus, ℓ must be at least $n^{\Omega(\sqrt{k})}$. □

Theorem 2.34 establishes that resolution proofs can be long as an exponential function of the size of the input clauses. Historically, the above theorem was not the first example of a proof that some formulas have long resolution proofs. The first such result was established by Haken. He showed that the *pigeon hole principle* has long resolution proofs. Recall that the pigeon hole principle says that if there are n holes and $n + 1$ pigeons then some hole may contain more than one pigeon. Let $\mathrm{pigeonhole}_n$ be the propositional logic formula that says that every pigeon goes to a hole and no hole contains more than one pigeon. Then $\mathrm{pigeonhole}_n$ is unsatisfiable, and Haken showed that any resolution refutation of this fact must be exponential in n. The broad principle of using interpolation and lower bounds from monotonic circuit complexity have been used to establish that other proof systems can also have long proofs.

Understanding how long resolution proofs can be, and when they can be long, helps our theoretical understanding of the limits of SAT solvers — what examples they may work well and when they can take long. But a probably more important reason for studying proof lengths in some proof system is because of its connections to some fundamental questions in complexity theory. Essentially, the goal in proof complexity is to understand if there is a proof system for propositional logic that has the property that all facts have short proofs. Investigating whether this is true or not has important implications in complexity theory. Let us see why.

Definition 2.35 A proof system Π is *super* if every tautology φ has a proof in Π such that length of the proof is bounded by a polynomial function of $|\varphi|$.

Now Theorem 2.34 says that resolution is not a super proof system. But are there other proof systems that are super? This is intimately tied to another open question in complexity theory.

Theorem 2.36 (Cook-Reckhow)

Propositional logic has a super proof systems if and only if $\mathsf{NP} = \mathsf{coNP}$.

Proof There are two directions to this proof. Assume that there is a super proof system. Then the problem to determine if a given formula is valid, is in NP — the NP algorithm simply guesses the proof and checks that it is a proof in our super proof system. Since the problem of checking validity is coNP-complete, it follows that $\mathsf{coNP} = \mathsf{NP}$; the NP algorithm for an arbitrary problem $A \in \mathsf{coNP}$ is simple to reduce it to validity checking and then use the NP algorithm based on the super proof system.

On the other hand, suppose $\mathsf{NP} = \mathsf{coNP}$ then there is nondeterministic Turing machine N, running in polynomial time, that checks if a given propositional logic formula is valid. Proofs for a formula φ in our new "super" proof system will simply be the nondeterministic choices that cause N to accept φ; notice, that these proofs will be polynomially long because N only has computations that are polynomially long. □

In the light of Theorem 2.36, to resolve the NP versus coNP question, we need to prove that there are no super proof systems for proposition logic. Cook proposed an approach to tackling this problem. Consider concrete natural proof systems for propositional logic, one by one, and show that they are not super. Then use the intuition developed in this process to generalize and prove the absence of any super proof system. Resolution was the first proof system shown to be not super. Since then other proof systems have also been proved to be not super. However, there are still natural proof systems for which we have not been able to prove exponential lower bounds for proof lengths. One such proof system is the Frege proof system we introduced. It is still open whether there are tautologies for which proofs in the Frege proof system will be exponentially long.

Chapter 3
First Order Logic

Syntax, Semantics, and Overview

First order logic is a formal language to describe and reason about *predicates*. Modern efforts to study this logic grew out of a desire to study the foundations of mathematics in number theory and set theory. It has a careful treatment of functions, variables, and quantification. First order logic deals with predicates as opposed to propositions — declarative statements that are either true or false — which is the subject of study in propositional logic. A predicate is a proposition that depends on the value of some variables. For example truth of the statement "x is prime", depends on the value x takes (and of course also on the meaning of "is prime"). If $x = 2$ the statement "x is prime" would be true and if $x = 4$ it would be false. Predicates may depend on more than one variable. For example, the truth of $P(x, y) \overset{\Delta}{=} x + y = 0$ depends on the values of both x and y.

One way to convert a predicate into a proposition by substituting values for the predicate variables. For example, for the predicate P defined in the previous paragraph, $P(2, -2)$ denotes the proposition "2+(-2) = 0". Another way to obtain propositions in predicate logic is by using *quantifiers*, which allows one to express statements like the predicate holds for all values of the variable, or the predicate holds for some values of the variable. In this chapter, we introduce the syntax and semantics of first order logic, and some of the questions we will explore in this book.

3.1 Syntax

First order logic formulas are defined over a *vocabulary* or *signature* that identifies *non-logical symbols*, namely, the predicates, constants, and functions that can be used in the formulas.

Definition 3.1 A *vocabulary* or *signature* is $\tau = \{\mathcal{C}, \mathcal{F}, \mathcal{R}\}$, where

- $\mathcal{C} = \{c_1, c_2, \ldots\}$ is a set of *constant* symbols,
- $\mathcal{F} = \{\mathcal{F}^k\}_k$ is a collection of sets with $\mathcal{F}^k = \{f_1^k, f_2^k, \ldots\}$ being the set of k-ary function symbols, and

- $\mathcal{R} = \{\mathcal{R}^k\}_k$ is a collection of sets with $\mathcal{R}^k = \{R^k_1, R^k_2, \ldots\}$ being the set of k-ary relation symbols.

Note that any of the above sets of constants, k-ary function symbols or k-ary relation symbols can be empty, finite, or infinite. A signature is *purely relational* or simply *relational* if there are no constants or functions in the signature, i.e., $C = \mathcal{F} = \emptyset$. A signature is *finite* if the total number of symbols in the signature is finite.

We will typically consider signatures that are finite. When the arity of a function or relation symbol is clear from the context, we will drop the superscript.

Formulas in first-order logic over signature τ are sequences of symbols, where each symbol is one of the following.

1. The symbol =
2. An element of the infinite set $\mathcal{V} = \{x_1, x_2, x_3, \ldots\}$ of *variables*
3. Constant symbols, function symbols and relation symbols in τ
4. The symbol $\neg$ called *negation*
5. The symbol $\vee$ called *disjunction*
6. The symbol $\exists$ called the *existential quantifier*
7. The symbols (and) called *parenthesis*

As always, not all such sequences are formulas; only *well formed* sequences are formulas in the logic. In order to define well formed formulas, we first need to define the set of *terms*.

Definition 3.2 The set of *terms* over signature $\tau = \{C, \mathcal{F}, \mathcal{R}\}$ is inductively defined as follows.

1. Every variable $x \in \mathcal{V}$ is a term.
2. Every constant symbol c in τ is a term.
3. If f is a k-ary function in τ and $t_1, t_2, \ldots t_k$ are terms then $f(t_1, t_2, \ldots t_k)$ is a term.

We could capture this definition succinctly by the following BNF grammar.

$$t ::= x \mid c \mid f(t, t, \ldots t)$$

where x is a variable, c is constant symbol and f is a function symbol.

Having defined terms, we can use them to define well formed formulas (wff) or just formulas for short.

Definition 3.3 A *well formed formula (wff)* over signature τ is inductively defined as follows.

1. If t_1, t_2 are terms then $t_1 = t_2$ is a wff.
2. If $t_1, t_2, \ldots t_k$ are terms and R is a k-ary relation symbol in τ then $R(t_1, t_2, \ldots t_k)$ is a wff.
3. If φ is a wff then $(\neg\varphi)$ is a wff.

4. If φ and ψ are wffs then $(\varphi \vee \psi)$ is a wff.
5. If φ is a wff and x is a variable then $(\exists x\varphi)$ is a wff.

More succinctly, we could capture the above definitions of terms and formulas by the following BNF grammar.

$$\varphi ::= t = t \mid R(t, t, \ldots t) \mid (\neg\varphi) \mid (\varphi \vee \varphi) \mid (\exists x\varphi)$$

where x is a variable, t is term (given by Definition 3.2, and R is a relation symbol, and x is a variable.

Atomic formulas are wffs that do not have any logical operators, i.e., either of the form $t_1 = t_2$ or $R(t_1, t_2, \ldots t_k)$, where each t_i is term and R is a k-ary relation symbol. Finally, a *literal* is formula that either atomic or the negation of an atomic formula.

It is useful to introduce logical operators in addition to those in Definition 3.3. These operators can be "syntactically" defined in terms of the operators in Definition 3.3. As in propositional logic, we can define the Boolean connectives *conjunction* as $\varphi \wedge \psi = (\neg((\neg\varphi) \vee (\neg\psi)))$, *implication* as $\varphi \rightarrow \psi = ((\neg\varphi) \vee \psi)$, *true* as $\top = (\varphi \vee (\neg\varphi))$, and *false* as $\bot = (\neg\top)$. Finally, we can define *universal quantification* as $(\forall x\varphi) = (\neg(\exists x(\neg\varphi)))$.

To avoid too many parenthesis, and at the same time have an unambiguous interpretation of formulas, we will assume the following precedence of operators (from increasing to decreasing): $\neg$, $\wedge$, $\vee$, $\rightarrow$, $\forall$, $\exists$. Thus $\forall x\forall y\ x = y \rightarrow \neg R(x, y)$ means $(\forall x(\forall y\ (x = y \rightarrow (\neg R(x, y)))))$. We will also drop the outermost parentheses, and since $\wedge$ and $\vee$ are associative, drop parentheses in formulas involving the conjunction/disjunction of multiple formulas.

Example 3.4 Consider signature $\tau = \{R\}$ where R is a binary relation symbol. The following are formulas over this signature.

- *Reflexivity:* $\forall x R(x, x)$
- *Irreflexivity:* $\forall x(\neg R(x, x))$
- *Symmetry:* $\forall x\forall y(R(x, y) \rightarrow R(y, x))$
- *Anti-symmetry:* $\forall x\forall y((R(x, y) \wedge R(y, x)) \rightarrow x = y)$
- *Transitivity:* $\forall x\forall y\forall z((R(x, y) \wedge R(y, z)) \rightarrow R(x, z))$

Non-examples of formulas include $R(x)$ (R expects two arguments); x (a variable is not a formula); $(R(x, y) \vee R(z, x)$ (mismatched parentheses); $\exists x$ (x is quantified but there is no formula provided as argument).

3.2 Semantics

The semantics of formulas in any logic is defined with respect to a *model*. In the context of propositional logic, models were truth assignments to the propositions. For first order logic, models will be objects that help identify the interpretation of constants and relation symbols. Such models are called *structures*.

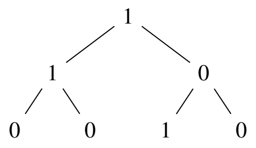

Fig. 3.1 Example of labeled binary tree.

Definition 3.5 A structure $\mathcal{A}$ of signature τ is $\mathcal{A} = (A, \{c^{\mathcal{A}}\}_{c\in\tau}, \{f^{\mathcal{A}}\}_{f\in\tau}, \{R^{\mathcal{A}}\}_{R\in\tau})$ where

- A is a non-empty set called the *domain/universe* of the structure,
- For each constant symbol $c \in \tau$, $c^{\mathcal{A}} \in A$ is its interpretation,
- For each k-array function symbol $f \in \tau$, $f^{\mathcal{A}} : A^k \longrightarrow A$ is its interpretation, and
- For each k-ary relation symbol $R \in \tau$, $R^{\mathcal{A}} \subseteq A^k$ is its interpretation.

The structure $\mathcal{A}$ is said to be *finite* if the universe A is finite. The universe of a structure $\mathcal{A}$ will be denoted by $u(\mathcal{A})$.

Many mathematical objects can be studied through the lens of logic. Let us look at some example signatures and structures.

Example 3.6 Consider the signature $\tau_G = \{E\}$, where E is a binary relation. We use this signature to study graphs. A graph $H = (V, E)$ modeled as a structure is $\mathcal{G} = (G, E^{\mathcal{G}})$, where the universe G is the set of vertices V, and for a pair of vertices $u, v \in G$ $(= V)$, $E^{\mathcal{G}}uv$ [1] holds iff $(u, v) \in E$.

Example 3.7 Let $\tau_O = \{<, S\}$ where $<$ and S are binary relation symbols. A finite order structure is $O = (O, <^O, S^O)$, where O is the universe of elements, $<$ is interpreted to be an ordering relation, and S as the "successor" relation.

Example 3.8 Let $\tau_A = \{\circ\}$ where $\circ$ is a binary function. A group would be a structure with a universe, where the operation $\circ$ is associative, has an identity, and every element has an inverse.

Example 3.9 Labeled binary trees, where vertices are labeled by elements of Σ, can be represented as a structure in the following manner. Let $\tau_T = \{<, S_0, S_1, (Q_a)_{a\in\Sigma}\}$ where $<, S_0, S_1$ are binary relation symbols, Q_a is a unary relation symbol. A tree (labeled by symbols in Σ) is a structure $\mathcal{T} = (T, <^{\mathcal{T}}, S_0^{\mathcal{T}}, S_1^{\mathcal{T}}, (Q_a^{\mathcal{T}})_{a\in\Sigma})$ where elements of T are called vertices, $<$ is the ancestor relation, S_0 and S_1 are the left and right child relations, respectively, and Q_a holds in all vertices labeled by a.

For example, consider the binary tree shown in Fig. 3.1 . Let us see how this tree is represented as a structure. The universe will consist of the vertices of the tree. We could use any names for the vertices. But it is convenient to name them in a manner that makes the edge relation explicit — the root will be ε, and for a vertex

[1] For a relation symbol R, we will sometimes write $R^{\mathcal{A}}a_1a_2\cdots a_n$ instead of $(a_1, a_2, \ldots a_n) \in R^{\mathcal{A}}$.

w, its left child will be $w0$, while its right child will be $w1$. Given this, the tree in Fig. 3.1 corresponds to the following structure. $\mathcal{T} = (\{\varepsilon, 0, 1, 00, 01, 10, 11\}, <^{\mathcal{T}} = \{(u, uv) \mid v \neq \varepsilon\}, S_0^{\mathcal{T}} = \{(u, u0) \mid u \in \{\varepsilon, 0, 1\}\}, S_1^{\mathcal{T}} = \{(u, u1) \mid u \in \{\varepsilon, 0, 1\}\}, Q_0 = \{1, 00, 01, 11\}, Q_1 = \{\varepsilon, 0, 10\})$.

In order to define the semantics of a first order logic formula, we need a structure, and an *assignment*. An assignment maps every variable to an element in the universe of the structure.

Definition 3.10 For a τ-structure $\mathcal{A}$, an *assignment* over $\mathcal{A}$ is a function $\alpha : \mathcal{V} \to u(\mathcal{A})$ that assigns every variable $x \in \mathcal{V}$ a value $\alpha(x) \in u(\mathcal{A})$.

Fixing the values of the variable, and the interpretation of the function symbols, ensures that each term evaluates to value in $u(\mathcal{A})$. For a term t, we will abuse notation and define this value as $\alpha(t)$ and this can be defined inductively as follows.

- For a variable x, $\alpha(x)$ is simply the value α assigns to x.
- For constant symbol c, $\alpha(c) = c^{\mathcal{A}}$.
- For term $f(t_1, t_2, \ldots t_k)$, $\alpha(f(t_1, t_2, \ldots t_k)) = f^{\mathcal{A}}(\alpha(t_1), \ldots \alpha(t_k))$.

For an assignment α over $\mathcal{A}$, $\alpha[x \mapsto a]$ is the assignment

$$\alpha[x \mapsto a](y) = \begin{cases} \alpha(y) & \text{for } y \neq x \\ a & \text{for } x = y \end{cases}$$

We now have all the elements to define the semantics of a formula. The satisfaction relation will be a ternary relation — $\mathcal{A} \models \varphi[\alpha]$ to be read as "φ is true/holds in $\mathcal{A}$ under assignment α". The relation will be defined inductively on the structure of the formula. In defining the relation, we will also say $\mathcal{A} \not\models \varphi[\alpha]$ to mean that $\mathcal{A} \models \varphi[\alpha]$ does not hold.

Definition 3.11 The relation $\mathcal{A} \models \varphi[\alpha]$ is inductively defined as follows.

- $\mathcal{A} \models t_1 = t_2[\alpha]$ iff $\alpha(t_1) = \alpha(t_2)$
- $\mathcal{A} \models R(t_1, \ldots t_n)[\alpha]$ iff $(\alpha(t_1), \alpha(t_2), \ldots \alpha(t_n)) \in R^{\mathcal{A}}$
- $\mathcal{A} \models (\neg\varphi)[\alpha]$ iff $\mathcal{A} \not\models \varphi[\alpha]$
- $\mathcal{A} \models (\varphi \vee \psi)[\alpha]$ iff $\mathcal{A} \models \varphi[\alpha]$ or $\mathcal{A} \models \psi[\alpha]$
- $\mathcal{A} \models (\exists x\varphi)[\alpha]$ iff for some $a \in u(\mathcal{A})$, $\mathcal{A} \models \varphi[\alpha[x \mapsto a]]$

Example 3.12 Consider a structure (graph) over the vocabulary of graphs ($\tau_G = \{E\}$) $\mathcal{G} = (\{1, 2, 3, 4\}, E^{\mathcal{G}} = \{(1, 2), (2, 3), (3, 4), (4, 1)\})$. For any assignment α, $\mathcal{G} \models \forall x \exists y E(x, y)[\alpha]$ because

$$\begin{aligned}
&\mathcal{G} \models \exists y E(x, y)[\alpha[x \mapsto 1]] \text{ because} \\
&\qquad \mathcal{G} \models E(x, y)[\alpha[x \mapsto 1][y \mapsto 2]], \\
&\mathcal{G} \models \exists y E(x, y)[\alpha[x \mapsto 2]] \text{ because} \\
&\qquad \mathcal{G} \not\models E(x, y)[\alpha[x \mapsto 2][y \mapsto 3]], \\
&\mathcal{G} \models \exists y E(x, y)[\alpha[x \mapsto 3]] \text{ because} \\
&\qquad \mathcal{G} \not\models E(x, y)[\alpha[x \mapsto 3][y \mapsto 4]], \text{ and} \\
&\mathcal{G} \models \exists y E(x, y)[\alpha[x \mapsto 4]] \text{ because} \\
&\qquad \mathcal{G} \not\models E(x, y)[\alpha[x \mapsto 4][y \mapsto 1]].
\end{aligned}$$

Notice that in Example 3.12, the actual assignment to variables x and y did not matter when determining the satisfaction of the formula in the graph. This is because they are *bound* by the universal and existential quantifiers in the formula φ. This leads us to the important notion of bound and free variables in a formula. We begin by defining the scope of a quantifier.

Definition 3.13 For a wff $\varphi = (\exists x \psi)$, ψ is said to be the *scope* of the quantifier $\forall x$.

Definition 3.14 Every occurrence of the variable x in $\varphi = (\exists x \psi)$ is called a *bound occurrence* of x in φ.

Any occurrence of x which is not bound is called a *free occurrence* of x in φ.

The free variables in wff φ will be denoted by free(φ). The notation $\varphi(x_1, x_2, \dots x_n)$ will be used to indicate that free(φ) $\subseteq \{x_1, \dots x_n\}$.

Let us look at an example to understand the subtle definition of bound and free variables.

Example 3.15 Consider $\varphi = P(\mathbf{x}, \mathbf{y}) \vee (\exists x(\exists y R(x, y)) \vee Q(x, \mathbf{y}))$ the free variables are shown in **bold**. Notice that a variable may occur both bound and free. As we will establish soon, we can change the names of bound variables without affecting the meaning of formulas. Thus $\psi = P(\mathbf{x}, \mathbf{y}) \vee (\exists u(\exists v R(u, v)) \vee Q(u, \mathbf{y}))$ is an equivalent formula. Therefore, we will typically assume that bound and free variables are disjoint. In addition, since bound variables can be renamed without affecting its meaning, we can also assume that every bound variable is in the scope of a unique quantifier. Thus, instead of $P(\mathbf{x}, \mathbf{y}) \vee (\exists u(\exists v R(u, v)) \vee (\exists v Q(u, v)))$, we will consider the equivalent formula $P(\mathbf{x}, \mathbf{y}) \vee (\exists u(\exists v R(u, v)) \vee (\exists z Q(u, z)))$

The satisfaction of a formula in a structure $\mathcal{A}$ under assignment α only depends on the values α assigns to the free variables; the values assigned to the bound variables in α are unimportant.

Theorem 3.16 *For a formula φ and assignments α_1 and α_2 such that for every $x \in$ free(φ), $\alpha_1(x) = \alpha_2(x)$, $\mathcal{A} \models \varphi[\alpha_1]$ iff $\mathcal{A} \models \varphi[\alpha_2]$.*

Theorem 3.16 can be proved by induction on the structure of the formula φ. The proof is left as an exercise for the reader. Theorem 3.16 suggests that if a formula has no free variables, its truth is independent of the assignment. Formulas without any free variables (i.e., those all of whose variables are bound) are an important class of formulas and have special name.

Definition 3.17 A *sentence* is a formula φ none of whose variables are free, i.e., free(φ) $= \emptyset$.

An immediate consequence of Theorem 3.16 is that the truth of sentences is independent of the assignment.

Proposition 3.18 *For a sentence φ, and any two assignments α_1 and α_2, $\mathcal{A} \models \varphi[\alpha_1]$ iff $\mathcal{A} \models \varphi[\alpha_2]$.*

Proposition 3.18 is an immediate consequence of Theorem 3.16. Thus, for a sentence φ, we say $\mathcal{A} \models \varphi$ whenever $\mathcal{A} \models \varphi[\alpha]$ for some α.

Definition 3.19 For a sentence φ, $\mathcal{A}$ is said to be a *model* of φ iff $\mathcal{A} \models \varphi$. We will denote by $[\![\varphi]\!]$ the set of all models of φ.

3.2.1 Satisfiability, Validity, and First order theories

Satisfiability and validity/tautologies are defined in a manner similar to that for propositional logic — a formula is satisfiable if there is some model and assignment in which it is true, and it is valid if it is true in all models and assignments.

Definition 3.20 A formula φ over signature τ is said to be *satisfiable* iff for some τ-structure $\mathcal{A}$ and assignment α, $\mathcal{A} \models \varphi[\alpha]$.

A formula φ over signature τ is said to be logically *valid* iff for every τ-structure $\mathcal{A}$ and assignment α, $\mathcal{A} \models \varphi[\alpha]$. We will denote this by $\models \varphi$.

We can also define with a formula φ is a *logical consequence* of a set of formulas Γ in exactly the same way as we defined it for propositional logic.

Definition 3.21 For a set of formulas Γ, we say $\mathcal{A} \models \Gamma[\alpha]$ iff for every $\varphi \in \Gamma$, $\mathcal{A} \models \varphi[\alpha]$.

We say φ is a *logical consequence* of Γ, denoted by $\Gamma \models \varphi$, if and only if for every $\mathcal{A}$ and α, $\mathcal{A} \models \Gamma[\alpha]$ implies that $\mathcal{A} \models \varphi[\alpha]$. Thus, if $\emptyset \models \varphi$ then $\models \varphi$.

The following observation is an immediate consequence of the definition of logical consequence.

Proposition 3.22 $\Gamma \cup \{\varphi\} \models \psi$ *iff* $\Gamma \models \varphi \to \psi$

Finally two formulas are (semantically) equivalent, if the hold in exactly the same set of structures and assignments.

Definition 3.23 Formulas φ and ψ are said to be *logically equivalent* (denoted $\varphi \equiv \psi$) if for every $\mathcal{A}$ and assignment α, $\mathcal{A} \models \varphi[\alpha]$ iff $\mathcal{A} \models \psi[\alpha]$.

A *first order theory* T over signature τ is any set of sentences over signature τ. A theory T is said to be *inconsistent* if there is a sentence φ such that $\{\varphi, \neg\varphi\} \subseteq T$. If T is not inconsistent then it is said to be *consistent*. Finally, T is *complete* if for every sentence φ over signature τ either $\varphi \in T$ or $\neg\varphi \in T$.

First order theories are typically identified by structures or axioms (i.e., sentences) as follows. For a structure $\mathcal{A}$, the first order theory of $\mathcal{A}$, denoted $\mathrm{Th}(\mathcal{A})$, is defined as

$$\mathrm{Th}(\mathcal{A}) = \{\varphi \text{ a sentence} \mid \mathcal{A} \models \varphi\}.$$

Thus, $\mathrm{Th}(\mathcal{A})$ is the set of all sentences that are true in the structure $\mathcal{A}$. Notice that, since for any sentence φ exactly one of φ or $\neg\varphi$ is true in $\mathcal{A}$ (by definition of $\neg$), it follows that $\mathrm{Th}(\mathcal{A})$ is consistent and complete for any structure $\mathcal{A}$. For a set of structure C, the theory of C (Th(C)) is set of sentences that hold in all the structures of C. That is,

$$\mathrm{Th}(C) = \bigcap_{\mathcal{A}\in C} \mathrm{Th}(\mathcal{A}).$$

A couple of observations about this definition are worth making. First if C is an empty set then Th(C) is the set of *all* sentences and is therefore inconsistent. On the other hand, for a non-empty set C, since Th($\mathcal{A}$) is consistent for every structure $\mathcal{A} \in C$, it follows that Th(C) is also consistent. However, it may or may not be complete depending on what C.

Axioms or sets of sentences, are another way in which theories are defined. For a set of sentences Γ, the theory of Γ is given by

$$\mathrm{Th}(\Gamma) = \{\varphi \text{ a sentence} \mid \Gamma \models \varphi\}.$$

We could define Th(Γ) in another way. Recall that for a sentence φ, $[\![\varphi]\!]$ is the set of all structures in which φ holds. We can extend this to a set of sentences Γ by defining $[\![\Gamma]\!]$ to be the set of structures in which every sentence in Γ holds. In other words, $[\![\Gamma]\!] = \cap_{\varphi\in\Gamma}[\![\varphi]\!]$. Then Th($\Gamma$) is nothing but Th($[\![\Gamma]\!]$). Based on the discussion in the preceding paragraph on the consistency of the theory of a set of structures, we can conclude that Th(Γ) is consistent if and only if $[\![\Gamma]\!]$ is non-empty. Depending on the set Γ, Th(Γ) may or may not be complete.

3.3 Overview

There are a number of computational questions related to first order logic that we will investigate in these notes. The main ones relate to whether a sentence is true in *some* structure (*satisfiability*), in *all* structures (*validity*), and in all structures belonging to some set C over a signature. These computational questions are much harder than similar questions asked in the context of propositional logic.

Let us start with the question that is conceptually the simplest: Given a structure $\mathcal{A}$ and sentence φ, is $\mathcal{A} \models \varphi$ or equivalently, is $\varphi \in \mathrm{Th}(\mathcal{A})$? In the context of propositional logic the analogous question (given a truth assignment v and formula φ determine if $\mathsf{v} \models \varphi$ is a computationally simple problem — we simply evaluate φ in v which can be done in time that is linear in the size of φ. In first order logic, it is not clear how this problem can be solved. If $\mathcal{A}$ is a finite structure, we could simply unwind the definition of satisfaction (as we did in Example 3.12) and check if the sentence holds. When $\mathcal{A}$ is infinite, the challenge is that existential quantifiers would require us to search in an infinite universe for a witness that the formula holds. But this begs an even more basic question, if $\mathcal{A}$ is infinite, how is it given as input to the problem? We will consider structures $\mathcal{A}$ that are "computable" in the sense that interpretations to constant symbols can be computed, and given representations of elements in the universe, one can compute the value of a function symbol on these arguments and one can decide if any tuple formed by these elements belongs to the interpretation of any relation symbol in the signature. Notice that we do not require the universe $u(\mathcal{A})$ itself to be a recursive set. We will not typically worry about these computability assumptions on the structure $\mathcal{A}$ because we will consider "standard" structures that are known to be computable in this sense. We will consider structures involving numbers and arithmetic operations, like naturals, integers, rationals, reals, equipped with the standard ordering relation and arithmetic operations of addition and multiplication.

We will begin our study of computational questions related to first order logic by investigating structures $\mathcal{A}$ for which the set $\mathrm{Th}(\mathcal{A})$ is decidable. Decidability results in this space are often proved by a general technique of *quantifier elimination*. A theory $\mathrm{Th}(\mathcal{A})$ is said to *admit quantifier elimination* if for every formula φ, there is a quantifier-free formula φ' such that $\mathrm{free}(\varphi') \subseteq \mathrm{free}(\varphi)$ and φ' is equivalent to φ with respect to $\mathrm{Th}(\mathcal{A})$, i.e., $\mathrm{Th}(\mathcal{A}) \models \varphi \leftrightarrow \varphi'$ [2]. If the process of constructing the quantifier-free formula φ' is computable, and the problem of determining if $\psi \in \mathrm{Th}(\mathcal{A})$ is decidable for quantifier-free formulas ψ, then composing these steps, gives a decision procedure for checking if $\varphi \in \mathrm{Th}(\mathcal{A})$; this is often the case, and so if a theory admits quantifier elimination, then it is typically decidable. We will see that $\mathrm{Th}((\mathbb{R}, <))$ and $\mathrm{Th}((\mathbb{R}, 0, 1, +, <))$ admit quantifier elimination and are therefore decidable; here $<$ denotes the natural ordering on numbers and $+$ denotes addition on numbers. In fact, $\mathrm{Th}((\mathbb{Q}, <)) = \mathrm{Th}((\mathbb{R}, <))$ and $\mathrm{Th}((\mathbb{Q}, 0, 1, +, <)) = \mathrm{Th}((\mathbb{R}, 0, 1, +, <))$, and therefore these theories over the rational numbers are also decidable. The observations $\mathrm{Th}((\mathbb{Q}, <)) = \mathrm{Th}((\mathbb{R}, <))$ and $\mathrm{Th}((\mathbb{Q}, 0, 1, +, <)) = \mathrm{Th}((\mathbb{R}, 0, 1, +, <))$ demonstrate that there are non-isomorphic structures that are indistinguishable in terms of the first order sentences that they satisfy. We will see that $\mathrm{Th}((\mathbb{N}, 0, 1, +, <))$, which is known as Presburger's arithmetic, is also decidable. An even more surprising result is that $\mathrm{Th}((\mathbb{R}, 0, 1, +, \times, <))$ ($\times$ denotes multiplication on numbers) admits quantifier elimination and is decidable. This is a celebrated result due to Tarski and Seidenberg, and is beyond the scope of these notes. In contrast, both $\mathrm{Th}((\mathbb{Q}, 0, 1, +, \times, <))$ and $\mathrm{Th}((\mathbb{N}, 0, 1, +, \times, <))$ are not recursively enumerable. The later is a form of Gödel's Incompleteness theorem, while the former is a result due to Robinson. Notice that even though $\mathrm{Th}((\mathbb{Q}, 0, 1, +, <)) = \mathrm{Th}((\mathbb{R}, 0, 1, +, <))$, $\mathrm{Th}((\mathbb{R}, 0, 1, +, \times, <)) \neq \mathrm{Th}((\mathbb{Q}, 0, 1, +, \times, <))$. This can be seen as follows: the sentence $\varphi = \exists x\ x \times x = 1 + 1$ is in $\mathrm{Th}((\mathbb{R}, 0, 1, +, \times, <))$ (as we can take $x = \sqrt{2}$) but does not belong to $\mathrm{Th}((\mathbb{Q}, 0, 1, +, \times, <))$.

The classical decision problem is that of determining if a sentence φ is valid. That is, given a sentence φ over signature τ, determine if φ holds in every τ-structure. This problem, on first glance, may seem very general and of little practical importance. However, this is not true. It provides a framework to study general meta-theorems in logic that are independent of a particular structure or class of structures. Equally importantly, many computational questions can be reduced to this classical problem. Suppose we want to reason about a class of structures, and the class of structures can be described by a *finite* set of sentences or *axioms* Γ. For example, suppose we want to study properties that are true about groups. Recall that groups are structures

[2] $\varphi \leftrightarrow \psi$ is the formula $(\varphi \to \psi) \wedge (\psi \to \varphi)$.

where the universe is equipped with a binary operation $\circ : S \times S \to S$ that satisfies the following properties.

1. **Associativity:** For every a, b, c, $a \circ (b \circ c) = (a \circ b) \circ c$.
2. **Identity:** There is an element $e \in S$ such that for all a, $a \circ e = e \circ a = a$.
3. **Inverse:** For every a, there is an element a' such that $a \circ a' = a' \circ a = e$, where e is the identity.

Let Γ be the set of sentences encoding the properties of $\circ$ being associative, and having an identity and inverses. Checking if a property φ_{uniq} that says that the identity is unique — $\forall x \forall y (\varphi_{\mathsf{id}}(x) \wedge \varphi_{\mathsf{id}}(y)) \to x = y$, where $\varphi_{\mathsf{id}}(u) = \forall a\, a \circ u = a \wedge u \circ a = a$ — holds in every group is equivalent to checking if $\Gamma \models \varphi_{\mathsf{uniq}}$. This question is equivalent to checking if $\models (\wedge_{\psi \in \Gamma} \psi) \to \varphi_{\mathsf{uniq}}$, which is the classical decision problem.

One of the most important results is that the classical decision problem is recursively enumerable. This is due Gödel's completeness theorem which says that there is a sound and complete proof system (like the ones we saw for propositional logic in Chap. 2) for determining validity of first order logic sentences. Since checking if a sequence of formulas constitutes formal proof in these proof systems can be mechanized, the RE-procedure simply searches for a proof of validity. Unfortunately, the problem is RE-hard, and hence undecidable. The RE-procedure for the classical decision problem means that if Γ is an RE set of sentences then $\mathrm{Th}(\Gamma)$ is also RE. Moreover, if $\mathrm{Th}(\Gamma)$ is *consistent and complete* then $\mathrm{Th}(\Gamma)$ is decidable! The decision procedure for checking if $\varphi \in \mathrm{Th}(\Gamma)$ simply dovetails the RE-procedures for checking if $\varphi \in \mathrm{Th}(\Gamma)$ and the procedure for checking $\neg\varphi \in \mathrm{Th}(\Gamma)$. One of these is guaranteed to succeed since the consistency and completeness of Γ guarantees that exactly one out of φ and $\neg\varphi$ belong to $\mathrm{Th}(\Gamma)$.

Chapter 4
Quantifier Elimination and Decidability

In this chapter, we will look at the computational problem of determining if a formula belongs to the theory of a structure. The structures we will consider involve numbers and arithmetic. We will mainly focus on structures when this problem is decidable. The key idea behind the decidability algorithms in this chapter will be *quantifier elimination*. Let us begin with this key definition. A formula φ is said to be *quantifier-free* if the quantifiers $\exists$ and $\forall$ do not appear in φ.

Definition 4.1 (Quantifier Elimination)

A theory Γ over signature τ admits quantifier elimination if for every formula φ over signature τ, there is a quantifier free formula φ^* such that $\mathrm{free}(\varphi^*) \subseteq \mathrm{free}(\varphi)$ and φ^* is equivalent to φ with respect to Γ. That is,

$$\Gamma \models \varphi \leftrightarrow \varphi^*$$

where $\varphi \leftrightarrow \psi = (\varphi \rightarrow \psi) \wedge (\psi \rightarrow \varphi)$. Another way to say this is, for every structure τ-structure $\mathcal{A}$ such that $\mathcal{A} \models \Gamma$, and every assignment α,

$$\mathcal{A} \models \varphi[\alpha] \text{ iff } \mathcal{A} \models \varphi^*[\alpha].$$

There are a few points worth noting about Definition 4.1. First, the free variables of quantifier-free formula φ^* is required to be a subset of the free variables of φ. Thus, in any structure $\mathcal{A}$ that satisfies Γ, the set of assignments α that satisfy φ is exactly the same as the set of assignments that satisfy φ^*. Second, while there is no requirement that the construction of φ^* from φ be computable, it is often the case that φ^* can be effectively constructed. Hence, if in addition, for any quantifier-free sentence ψ (i.e., a Boolean combination of atomic formulas built using the constants in the signature), the problem of determining if $\Gamma \models \psi$ is decidable, then $\mathrm{Th}(\Gamma)$ is decidable when Γ admits quantifier elimination. This is the approach we will use in this chapter to establish the decidability of $\mathrm{Th}(\mathcal{A})$ for some structures $\mathcal{A}$.

Finally, observe that if Γ is *inconsistent*, then it trivially admits quantifier elimination — *any* quantifier-free formula φ^* over the same free variables as φ is (vaccuously) equivalent to φ with respect to Γ. Let us look at a simple example of quantifier elimination.

Example 4.2 Consider the structure $(\mathbb{R}, 0, 1, +, \times, <)$. Consider the formula φ with free variables a, b, c given by $\exists x\ a \times x \times x + b \times x + c = 0$. The formula φ identifies assignments to the variables a, b, c such that the polynomial $ax^2 + bx + c$ has real roots. From high school algebra, we know that this happens when the discriminant of the polynomial is non-negative. That is, the quantifier-free formula φ^* equivalent to φ is

$$4 \times a \times c \leq b \times b$$

where by $s \leq t$ we mean the formula $(s < t) \vee (s = t)$.

We conclude this section by observing that to prove that a theory Γ admits quantifier elimination, we only need to establish this for special formulas. If every formula φ of the form $\exists x \psi$, where ψ is quantifier-free, there is a quantifier-free formula φ^* such that $\text{free}(\varphi^*) \subseteq \text{free}(\varphi)$ and φ^* is equivalent to φ with respect to Γ, then Γ admits quantifier elimination. The reason for this is that we can take any formula, express $\forall$ quantification using $\exists$ and negation, and starting with the innermost quantified formulas, systematically eliminate one quantifier at a time in order to eliminate all quantifiers. Hence eliminating quantifiers from formulas with a single existential quantifier is all that is needed to show that a theory admits quantifier elimination. We can make one additional simplifying assumption, if needed. We can assume that when doing quantifier elimination of $\varphi = \exists x \psi$, ψ is a conjunction of literals, i.e., a conjunction of atomic formulas or their negation. The reason is that for any arbitrary formula $\exists x \rho$, where ρ is quantifier-free, we can always treat it as a Boolean formula over atomic formulas, and write it in *disjunctive normal form* — a formula is in disjunctive normal form (DNF) if it is a disjunction of one or more conjunctions of literals — and every quantifier-free formula can be rewritten into an equivalent DNF formula. Then, we notice that $\exists$ quantifier distributes over disjunctions, and hence we can write the formula as a disjunction of formulae of the form $\exists x \rho'$, where ρ' is a conjunction of literals. If we can do quantifier elimination on such formulae, we can simply do this for each disjunct to obtain a quantifier-free formula.

Proposition 4.3 *Consider a theory Γ such that for every formula φ of the form $\exists x\ \bigwedge_{i=1}^{n} \alpha_i$, where each α_i is a literal, there is a quantifier-free formula φ^* such that free$(\varphi^*) \subseteq$ free(φ) and $\Gamma \models \varphi \leftrightarrow \varphi^*$. Then Γ admits quantifier elimination.*

Proof We will prove this by structural induction on the formulas. In the induction below, for a formula ψ, we will denote by $\mathsf{qe}(\psi)$ the equivalent quantifier-free formula constructed by the proof.

Base Case If ψ is an atomic formula, then simply take $\mathsf{qe}(\psi)$ to simply be ψ itself.

Case $\psi = \neg\psi_1$ It is easy to see that ψ is equivalent to the quantifier-free formula $\neg\mathsf{qe}(\psi_1)$.

Case $\psi = \psi_1 \vee \psi_2$ It is easy to see that ψ is equivalent to the quantifier-free formula $\mathsf{qe}(\psi_1) \vee \mathsf{qe}(\psi_2)$.

Case $\psi = \exists x\ \psi_1$ Observe that ψ is equivalent to $\exists x\ \mathsf{qe}(\psi_1)$. Converting $\mathsf{qe}(\psi_1)$ to DNF, suppose $\mathsf{qe}(\psi_1)$ is equivalent to $\bigvee_{i=1}^{k} \gamma_i$, where each γ_i is a conjunction of literals. Then $\exists x\ \mathsf{qe}(\psi_1)$ is equivalent to $\bigvee_{i=1}^{k} \exists x \gamma_i$. By our assumption, each $\exists x \gamma_i$ is equivalent to the quantifier-free formula $\mathsf{qe}(\exists x \gamma_i)$. Thus, ψ is equivalent to the quantifier-free formula $\bigvee_{i=1}^{k} \mathsf{qe}(\exists x \gamma_i)$. □

4.1 Dense Linear Orders without Endpoints

The first structure we will look at is $(\mathbb{R}, <)$, where the universe is the set of real numbers and we have one binary relation $<$ which is interpreted as the standard ordering relation on real numbers. We will show that $\mathrm{Th}((\mathbb{R}, <))$ admits quantifier elimination and is decidable. The procedure to eliminate quantifiers relies on the following properties of the ordering relation $<$.

$$\forall x\ \neg(x < x) \qquad \text{(Irreflexive)}$$
$$\forall x \forall y\ (x < y) \rightarrow \neg(y < x) \qquad \text{(Asymmetric)}$$
$$\forall x \forall y \forall z\ ((x < y) \wedge (y < z)) \rightarrow (x < z) \qquad \text{(Transitive)}$$
$$\forall x \forall y\ (x < y) \vee (x = y) \vee (y < x) \qquad \text{(Total)}$$
$$\forall x \forall y\ (x < y) \rightarrow (\exists z\ (x < z) \wedge (z < y)) \qquad \text{(Dense)}$$
$$\forall x \exists y\ (y < x) \qquad \text{(No Min)}$$
$$\forall x \exists y\ (x < y) \qquad \text{(No Max)}$$

The set of these 7 sentences will be denoted as the set DLOWE. The first 4 sentences ((Irreflexive) , (Asymmetric) , (Transitive) , and (Total)) state that $<$ is a total, strict, linear order. Equation (Dense) says that the ordering $<$ is *dense*, i.e., between any two elements one can always find a third element. The last two sentences ((No Min) and (No Max)) state that there is no minimum or maximum element.

We now show that $\mathrm{Th}((\mathbb{R}, <))$ admits quantifier elimination. Observe that over the signature $\{<\}$, the only atomic formulas are of the form $y < z$ or $y = z$, where y, z are variables. Our first observation shows that, in the presence of (Total) , Proposition 4.3 can specialized even further, and we can restrict our attention to formulas without negation.

Proposition 4.4 *Suppose every formula of the form $\exists x\ \bigwedge_{i=1}^{k} \beta_i$, where each β_i is atomic of the form $x < y$, $x = y$, or $y < x$, where y is a variable that is different from x, is equivalent to a quantifier-free formula φ^* with respect to $Th((\mathbb{R}, <))$. Then $Th((\mathbb{R}, <))$ admits quantifier elimination.*

Proof By Proposition 4.3, to prove that $\mathrm{Th}((\mathbb{R}, <))$ admits quantifier elimination, we only need to consider formulas of the form $\varphi = \exists x\ \bigwedge_{i=1}^{n} \alpha_i$, where each α_i is a literal. We first show that (Total) allows us to eliminate negation, and so $\bigwedge_{i=1}^{n} \alpha_i$ is equivalent to a *positive* Boolean combination (i.e., no negations) of *atomic* formulas.

Observe that, by (Total) ,

$$\neg(y = z) \equiv (y < z) \vee (z < y)$$
$$\neg(y < z) \equiv (y = z) \vee (z < y)$$

Thus, negation can be eliminated, and $\bigwedge_{i=1}^{n} \alpha_i$ is equivalent to a positive Boolean combination of atomic formulas. We can convert this into disjunctive normal form, push existential quantification inside, and see that

$$\varphi \equiv \bigvee_{i=1}^{\ell} \exists x \bigwedge_{j=1}^{k_i} \beta_{ij},$$

where each β_{ij} is an atomic formula. Thus, to prove that $\mathrm{Th}((\mathbb{R}, <))$ admits quantifier elimination, we can focus our attention to formulas of the form $\exists x \ \bigwedge_{i=1}^{k} \beta_i$, where each β_i is atomic.

Consider $\psi = \exists x \ \bigwedge_{i=1}^{k} \beta_i$, where each β_i is atomic. Observe that, by (Irreflexive) , $x < x \equiv \bot$. Thus, if any $\beta_i = x < x$ then ψ is equivalent to the quantifier-free formula $\bot$. Next, since $x = x$ is equivalent to $\top$, if any $\beta_i = (x = x)$, we can drop β_i from the conjunct. Finally, if one of the β_is is of the form $y \bowtie z$, where $\bowtie \in \{=, <\}$ and $y, z \neq x$, then we can "pull out" β_i from the quantification. This is because

$$\exists x \ (y \bowtie z) \wedge \rho \equiv (y \bowtie z) \wedge \exists x \ \rho.$$

Thus, without loss of generality, each β_i is of the form $x < y$, $x = y$ or $y < x$, for $y \neq x$ and so the proposition is established. □

Having established Proposition 4.4, we are ready to complete the proof. Consider a formula $\varphi = \exists x \ \bigwedge_{i=1}^{k} \beta_i$, where each β_i is either $x < y$, $x = y$, or $y < x$, for $y \neq x$. We consider two cases.

- Consider the case when there is an i and variable y such that $\beta_i = (x = y)$, i.e., one of the conjuncts is an equality constraint. Assume, without loss of generality, $\beta_1 = (x = y)$. In this case, the value for x must be the same as y. We can substitute x with y and eliminate the variable x. That is,

$$\varphi \equiv \bigwedge_{i=2}^{k} \beta_i[x \mapsto y]$$

- Assume that none of the conjuncts β_i are equality constraints. That is, each β_i is either $x < y$ or $y < x$ for some variable $y \neq x$. In other words, we can write φ as

$$\varphi = \exists x \left(\bigwedge_{\ell \in L} \ell < x \right) \wedge \left(\bigwedge_{u \in U} x < u \right)$$

  where L and U are sets of variables. Clearly, if there is a value of x that satisfies φ with respect to an assignment α, then for every $\ell \in L$ and $u \in U$, $\alpha(\ell) < \alpha(u)$.

Conversely, when L and U are non-empty, if for an assignment α, $\alpha(\ell) < \alpha(u)$ for every $\ell \in L$ and $u \in U$, then by picking x to be a value between the the "largest" element in L and the "smallest" element in U we can show that φ holds with respect to assignment α. This can always be accomplished, since $<$ is dense. Now, if either L or U is empty, then φ can be satisfied by picking a value for x that is either very small or very large, which is possible since our structure has no minimum or maximum. This reasoning shows that there is some x satisfying the constraints if and only if every variable in L takes a value that is less than the value taken by every variable in U. Using this observation, we can say

$$\varphi \equiv \bigwedge_{\ell \in L,\ u \in U} \ell < u.$$

When L or U is empty, the above formula is an *empty* conjunction, which by convention is $\top$.

We can summarize the above observations in the main theorem for this section.

Theorem 4.5 *Th*$((\mathbb{R}, <))$ *admits quantifier elimination.*

The arguments in this section that establish Theorem 4.5, only rely on (Irreflexive) , (Asymmetric) , (Transitive) , (Total) , (Dense) , (No Min) , and (No Max) , i.e., the sentence in DLOWE. Thus, *any* structure over the signature $\{<\}$ that satisfies all the sentence in DLOWE admits quantifier elimination. For example, since the rationals also satisfy all the properties in DLOWE, they also admit quantifier elimination. More generally, we will say structure $\mathcal{A}$ over signature $\{<\}$ is said to be *dense linear order without endpoints* if $\mathcal{A} \models$ DLOWE. Two examples of dense linear orders without endpoints are $(\mathbb{R}, <)$ and $(\mathbb{Q}, <)$. We can strengthen Theorem 4.5 as follows.

Theorem 4.6 *If $\mathcal{A}$ is a dense linear order without endpoints, then Th$(\mathcal{A})$ admits quantifier elimination.*

Observe that our argument for Theorem 4.6 is *constructive*. Hence, given a formula φ over $\{<\}$, there is an algorithm that will construct the equivalent quantifier-free formula φ^*. Next, if φ is a sentence, the equivalent quantifier-free formula φ^* is also a sentence (no free variables), and therefore, just a Boolean combination of $\top$ and $\bot$, which can be checked to see if it is true. Thus, for example, Th$((\mathbb{R}, <))$ and Th$((\mathbb{Q}, <))$ are decidable. It is worth observing that our procedure of constructing the quantifier-free formula, relies only on the sentences in DLOWE, and so the formula we construct is independent of the universe of the structure we are working in. Thus, Th$((\mathbb{R}, <))$ = Th$((\mathbb{Q}, <))$ = Th(DLOWE). This shows that there can be non-isomorphic structures (like $(\mathbb{R}, <)$ and $(\mathbb{Q}, <)$) that have the same first order theory. In general, as we shall later, for any infinite structure $\mathcal{A}$, it will always be the case that there are (infinitely many) different (non-isomorphic) structures $\mathcal{B}$ that will have the same theory as $\mathcal{A}$. We conclude this section with the main decidability result.

Theorem 4.7 *Th*(DLOWE) *is decidable. An immediate consequence of this is that Th$((\mathbb{R}, <))$ = Th$((\mathbb{Q}, <))$ = Th*(DLOWE) *are decidable.*

Each quantifier eliminated by our algorithm results in a quadratic blowup (because we construct a formula that compares each variable in L with each variable in U). Thus, if a sentence of size n has m quantifiers, its equivalent quantifier-free formula has size $O(n^{2^m})$ size. This analysis does not even take into account the fact that there are steps involving the construction of a DNF formula to get removing negations, etc. Thus our procedure has a doubly exponential complexity.

4.2 Linear Arithmetic

In this section, we will extend the results of Sect. 4.1 and consider properties of numbers that involve addition along with ordering. We will look at the structures $(\mathbb{R}, 0, 1, +, <)$ and $(\mathbb{Q}, 0, 1, +, <)$, where 0 and 1 are constants representing the numbers 0 and 1, respectively, and + is the binary function symbol representing addition.

The main result of this section is captured by the following two theorems.

Theorem 4.8 *The theories $Th((\mathbb{R}, 0, 1, +, <))$ and $Th((\mathbb{Q}, 0, 1, +, <))$ admit quantifier elimination.*

Since the processes of constructing quantifier-free equivalent formulas will be effective, we will in fact get a decision procedure for these theories.

Theorem 4.9 *The theories $Th((\mathbb{R}, 0, 1, +, <))$ and $Th((\mathbb{Q}, 0, 1, +, <))$ are decidable.*

We will prove Theorem 4.8. From Proposition 4.3, to show that quantifiers can be eliminated, we only need to show that quantifiers can be eliminated from formulas φ of the form $\exists x\ \psi$, where ψ is a quantifier-free formula. In other words, ψ is a Boolean combination of atomic formulas. Using de Morgan's laws, we can push negations inside all the way to atomic formulas. Since $<$ on both reals and rationals satisfies (Total), we can eliminate negations like in Proposition 4.4. Recall that in our signature, atomic formulas are of the form $u \bowtie v$, where $\bowtie \in \{=, <\}$ and u and v are expressions that look like $t_1 + t_2 + \cdots t_k$ with each t_i being either a variable y, or constants 0 or 1. And,

$$\neg(u = v) \equiv (u < v) \vee (v < u) \qquad \neg(u < v) \equiv (u = v) \vee (v < u).$$

Therefore, without loss of generality we may assume that ψ is a *positive* Boolean combination of atomic formulas.

Consider an atomic formula of the form $u \bowtie v$, where $\bowtie \in \{<, =\}$. We will treat these atomic formulas as equations/inequations over numbers, and use standard tricks to "solve for the variable x". That is, we will move all the terms involving variable x to one side with constants and other variables on the other side, and then "divide" by the coefficient of x. If x was present to begin with and does not get eliminated by this process, this will give us a formula of the form $x < u$ or $x = u$ or $x > u$, where u is a *linear expression* with rational coefficients involving the variables other than x. If x

gets eliminated or was not present to begin with, then we will get a constraint of the form $0 \bowtie u$, where $\bowtie \in \{<, =\}$ and u is a linear expression with rational coefficients involving variables other than x. Such constraints are technically not formulas in our signature, since our only constants are 0 and 1 and scalar multiplication is not a function symbol in our signature. However, this will just be an intermediate step. Before we construct the quantifier-free formula, we will get back to something that is in the legal syntax of our logic.

Let us look at an example to see what we mean by "solving for x".

Example 4.10 Consider the constraint $x + y + z + y + 1 < y + 1 + x + 1 + x + x$. In this solving for x, will result in the constraint

$$\frac{y}{2} + \frac{z}{2} - \frac{1}{2} < x.$$

Similarly, the constraint $x + y + x + x + z < 0$ when solved for x will result in the constraint

$$x < -\frac{y}{3} - \frac{z}{3}.$$

Based on the observations above, to show that $\mathrm{Th}((\mathbb{R}, 0, 1, +, <))$ admits quantifier elimination, we need construct quantifier-free equivalent formula for formulas of the form $\exists x\ \psi$, where ψ is a positive Boolean combination of constraints of the form $x < u$, $x = u$, $u < x$, $0 = u$ or $0 < u$ where u is a linear expression with rational coefficients not mentioning x. We will present two algorithms that will eliminate quantifiers from such formulas. The first algorithm due to Fourier and Motzkin, is very similar to the approach for dense linear orders without endpoints outlined in Sect. 4.1 . The second is a more efficient algorithm due Ferrante and Rackoff.

4.2.1 Fourier-Motzkin

Analogous to Proposition 4.4, we can show that we need to eliminate quantifiers only in formulas, where the quantifier-free formula in the scope of the quantifier is a conjunction of constraints involving x.

Proposition 4.11 *Suppose for every formula of the form $\exists x\ \bigwedge_{i=1}^{k} \beta_i$, where each β_i is of the form $x < u$, $x = u$, or $u < x$, where u is a linear expression with rational coefficients involving variables other than x, is equivalent to a quantifier-free formula φ^* with respect to $Th((\mathbb{R}, 0, 1, +, <))$ (or $Th((\mathbb{Q}, 0, 1, +, <))$). Then $Th((\mathbb{R}, 0, 1, +, <))$ (or $Th((\mathbb{Q}, 0, 1, +, <))$) admits quantifier elimination.*

Proof The proof is very similar to Proposition 4.4, and we recall the main ideas, leaving the details to the reader to work out. First, based on the discussion preceding this subsection, we need to eliminate the quantifier in a formula φ of the form $\exists x\ \psi$, where ψ is a positive Boolean combination of constraints of the form $x < u$, $x = u$, $u < x$, $0 < u$, or $0 = u$, where x does not appear in u. We can rewrite ψ in disjunctive normal form, push the existential quantifier inside the disjunction, and finally pull constraints of the form $0 < u$ and $0 = u$ out of the quantifier to get the result. □

Having established Proposition 4.11, the rest of the proof is similar to Sect. 4.1 . Consider a formula $\varphi = \exists x \ \bigwedge_{i=1}^{k} \beta_i$, where each β_i is either $x < u$, $x = u$, or $u < x$, for a linear expression u not involving x. We consider two cases.

- Consider the case when there is an i and expression u such that $\beta_i = (x = u)$, i.e., one of the conjuncts is an equality constraint. Assume, without loss of generality, $\beta_1 = (x = u)$. In this case, the value for x must be the same as u. We can substitute x with u and eliminate the variable x. That is,

$$\varphi \equiv \bigwedge_{i=2}^{k} \beta_i[x \mapsto u]$$

- Assume that none of the conjuncts β_i are equality constraints. That is, each β_i is either $x < u$ or $u < x$ for some linear expression u. In other words, we can write φ as

$$\varphi = \exists x \left(\bigwedge_{u \in L} u < x \right) \wedge \left(\bigwedge_{v \in U} x < v \right)$$

  where L and U are sets of linear expressions. As in the case of dense linear orders, we can argue that φ holds if and only if, every expression in L is smaller than every expression in U. Thus,

$$\varphi \equiv \bigwedge_{u \in L,\ v \in U} u < v.$$

  When L or U is empty, the above formula is an *empty* conjunction, which by convention is $\top$.

The final formula constructed by the above steps has constraints of the form $u = v$ or $u < v$, where u and v are linear expressions with rational coefficients. Such constraints are not in our signature. However, they can be rewritten into an equivalent formula in our syntax — we multiple each side by the LCM of the denominators, and rearrange terms to remove negative coefficients. We illustrate this through an example.

Example 4.12 Consider the constraint

$$\frac{y}{2} + \frac{z}{2} - \frac{1}{2} < -\frac{y}{3} - \frac{z}{3}$$

involving expressions constructed in Example 4.10. The LCM of the denominators is 6. Multiplying both sides by 6, and rearranging terms, we get the following sequence of steps.

$$\begin{gathered}
\tfrac{y}{2} + \tfrac{z}{2} - \tfrac{1}{2} < -\tfrac{y}{3} - \tfrac{z}{3} \\
3y + 3z - 3 < -2y - 2z \\
5y + 5z < 3 \\
y + y + y + y + y + z + z + z + z + z < 1 + 1 + 1
\end{gathered}$$

The last line is a formula in our syntax.

We conclude the section with an example that shows how the Fourier-Motzkin approach eliminates quantifiers.

Example 4.13 Consider the formula

$$\varphi = \forall x \ (0 < x) \longrightarrow (1 < x + y).$$

It is easy to see that the equivalent quantifier-free formula should be $y \geq 1$, or $(1 = y) \vee (1 < y)$. Let us see how the Fourier-Motzkin method constructs this expression.

Converting the for all quantifier in terms of exists, we get $\varphi = \neg\exists x \ \neg(0 < x \longrightarrow 1 < x + y)$. Consider $\psi = \exists x \ \neg(0 < x \longrightarrow 1 < x + y)$. We can rewrite the implication and push the negation inside to get

$$\psi \equiv \exists x \ (0 < x) \wedge \neg(1 < x + y).$$

Eliminating the negation using the totality axiom, solving for x, distributing the disjunctions over the conjunction, pushing existential quantifiers in, we get

$$\begin{aligned}
\psi &\equiv \exists x \ (0 < x) \wedge ((1 = x + y) \vee (x + y < 1)) \\
&\equiv \exists x \ (0 < x) \wedge ((x = 1 - y) \vee (x < 1 - y)) \\
&\equiv \exists x \ [(0 < x) \wedge (x = 1 - y)] \vee [(x < 0) \wedge (x < 1 - y)] \\
&\equiv [\exists x \ (0 < x) \wedge (x = 1 - y)] \vee [\exists x \ (0 < x) \wedge (x < 1 - y)]
\end{aligned}$$

Let $\psi_1 = \exists x \ (0 < x) \wedge (x = 1 - y)$ and $\psi_2 = \exists x \ (0 < x) \wedge (x < 1 - y)$. We will eliminate the quantifier in both ψ_1 and ψ_2 to get the formula for ψ.

Since ψ_1 contains an equality constraint, we have

$$\psi_1 \equiv 0 < 1 - y \equiv y < 1.$$

In ψ_2, we have one upper bound constraint and one lower bound constraint. So, when we eliminate the quantifier, we have

$$\psi_2 \equiv 0 < 1 - y \equiv y < 1$$

Thus,

$$\psi \equiv \psi_1 \vee \psi_2 \equiv (y < 1) \vee (y < 1) \equiv (y < 1)$$

Now, $\varphi = \neg\psi \equiv \neg(y < 1) \equiv (1 = y) \vee (1 < y)$, which is what we hoped.

4.2.2 Ferrante-Rackoff

Let us fix a formula φ of the form $\exists x\ \psi$, where ψ is a positive Boolean combination of constraints of the form $x < u$, $x = u$, $u < x$, $0 = u$ or $0 < u$ where u is a rational expression not mentioning x. Our goal is to elimnate the quantifier in φ. The Fourier-Motzkin algorithm relies on Proposition 4.11 which reduces the obligation to remove quantifiers to very special formulas. However, this step requires rewriting a formula to disjunctive normal form (see proof of Proposition 4.11), which can lead to an exponential blow-up. The Ferrante-Rackoff method avoids this conversion.

The key idea behind this approach is as follows. Let S be the expressions arising in constraints involving x in ψ. That is,

$$S = \{u \mid \exists \text{ constraint of the form } x = u,\ x < u,\ u < x \text{ in } \psi\}.$$

Depending on the valuation of the free variables, the expressions in S will be some rational numbers. Think of them on the “number line”. Now, x can be any number, but if two expressions u_1 and u_2 evaluate to two “consecutive” values on the number line, it doesn’t matter which value of x we pick in between u_1 and u_2. All of them will make the atomic constraints in ψ evaluate the same way. So we can just pick $(u_1 + u_2)/2$. Now, we don’t know what *order* the expressions in S will evaluate. But we can simply instantiate x to $(u + u')/2$ for *every pair* of expressions u, u'. Since, we will also do this also for the pair u, u (in which case $(u + u)/2 = u$), this will cover x being precisely equal to one of the expressions in S. For $u, v \in S$, define

$$\psi_{\frac{u+v}{2}} = \psi[x \mapsto \frac{u+v}{2}]$$

and take

$$\psi_m = \bigvee_{u,v \in S} \psi_{\frac{u+v}{2}}.$$

To cover the range of numbers less than *all* the expressions, we can instantiate x to anything smaller than all the expressions. But instead of doing this as a substitution, we just imagine instantiating x to some large negative value, and see how the atomic formulas in ψ will evaluate. Clearly, atomic formulas of the form $x < u$ will evaluate to $\top$, $x = u$ will evaluate to $\bot$, and $u < x$ will evaluate to $\bot$. So we can replace the atomic formulas by these values, and get a formula $\psi_{-\infty}$. That is,

$$\psi_{-\infty} = \psi[(x < u) \mapsto \top, (x = u) \mapsto \bot, (u < x) \mapsto \bot].$$

Similarly, to cover the range of rationals larger than all the expressions, we pretend instantiating x to a value much larger than the values of all the expressions. The atomic formulas $u < x$ evaluate to $\top$, and the formulas of the form $x = u$ and $x < u$ evaluate to $\bot$. Replacing these gives the formula $\psi_{+\infty}$. That is,

$$\psi_{+\infty} = \psi[(x < u) \mapsto \bot, (x = u) \mapsto \bot, (u < x) \mapsto \top].$$

We then take the disjunction of all the above formulas to eliminate x. In other words,

$$\varphi = \exists x\ \psi \equiv \psi_{-\infty} \vee \psi_{+\infty} \vee \psi_m.$$

Like in the Fourier-Motzkin case, the formula as written above will not be in the syntax of our logic. But as in Example 4.12, this can be rewritten in our syntax.

Let us look at an example to see how the Ferrante-Rackoff procedure works.

Example 4.14 Consider the formula $\varphi = \forall x\ (0 < x) \to (1 < x + y)$, from Example 4.13. Recall that the equivalent quantifier-free formula is $y \geq 1$, or $(1 = y) \vee (1 < y)$. As in Example 4.13, we can say that $\varphi = \neg\psi$, where $\psi = \exists x\ \neg(0 < x \to 1 < x + y) \equiv \exists x\ (0 < x) \wedge ((x = 1 - y) \vee (x < 1 - y))$. Let $\rho = (0 < x) \wedge ((x = 1 - y) \vee (x < 1 - y))$ and so $\psi = \exists x\ \rho$.

Let us first eliminate the quantifier in ψ. Based on the rewriting of ψ, we have $S = \{0, 1 - y\}$. The expressions we need to substitute x by are "$-\infty$", "$+\infty$", 0, $1 - y$, and $\frac{1-y}{2}$. We have,

$$\rho_{-\infty} = \bot \wedge (\bot \vee \top) \equiv \bot$$
$$\rho_{+\infty} = \top \wedge (\bot \vee \bot) \equiv \bot$$
$$\rho_0 = (0 < 0) \wedge ((0 = 1 - y) \vee (0 < 1 - y)) \equiv \bot$$
$$\rho_{1-y} = (0 < 1 - y) \wedge ((1 - y = 1 - y) \vee (1 - y < 1 - y)) \equiv (0 < 1 - y) \equiv y < 1$$
$$\rho_{\frac{1-y}{2}} = (0 < \tfrac{1-y}{2}) \wedge ((\tfrac{1-y}{2} = 1 - y) \vee (\tfrac{1-y}{2} < 1 - y)) \equiv (y < 1) \wedge ((y = 1) \vee (y < 1)) \equiv (y < 1)$$

Thus, we have

$$\psi \equiv \bot \vee \bot \vee \bot \vee (y < 1) \vee (y < 1) \equiv (y < 1).$$

Since $\varphi = \neg\psi$, we get $\varphi \equiv \neg(y < 1) \equiv (y = 1) \vee (1 < y)$.

Our procedures for eliminating quantifiers Sections 4.2.1 and 4.2.2, rely on properties that hold in both $(\mathbb{R}, 0, 1, +, <)$ and $(\mathbb{Q}, 0, 1, +, <)$. Further, starting with any sentence φ, the procedure (say the one by Ferrante and Rackoff) will construct the same quantifier-free formula φ^*, whether we are working with reals or rationals. Thus, $\mathrm{Th}((\mathbb{R}, 0, 1, +, <)) = \mathrm{Th}((\mathbb{Q}, 0, 1, +, <))$. No first order sentence can distinguish $(\mathbb{R}, 0, 1, +, <)$ and $(\mathbb{Q}, 0, 1, +, <)$.

Axiomatizations

One can ask, similar to dense linear orders without endpoints, whether there is a set of axioms/sentences that capture the property of linear arithmetic of reals/rationals. This is possible, but requires care. Chapter 3 of Calculus of Computation [**?**], presents such an axiomatization in parallel with the quantifier elimination procedure, and claims that they are the axioms for linear arithmetic. However, the axiomatization presented there is not complete, as it does not have any axioms for the constant 1.

In general, it is true however that for theories of a *single* structure (or more generally, for consistent and complete theories), the notions of the existence of a recursive axiomatization and decidability are synonymous. One direction is easy

— if the theory is decidable, we can simply take the theory itself as its recursive axiomatization (sounds like we are cheating, but we are not). And if there is a recursive axiomatization for a complete theory, it will follow, as we will show later (Gödel's strong completeness theorem), that the membership problem is recursively enumerable, and hence by simultaneously checking if φ or $\neg\varphi$ is in the theory, we can show the problem is decidable.

4.3 Other theories that admit quantifier elimination

There are several other important theories that admit quantifier elimination that we will not consider here.

Presburger arithmetic is $\mathsf{PresA} = \mathrm{Th}((\mathbb{N}, 0, 1, +, <))$, and can be shown to be decidable. However, PresA does not admit quantifier elimination. For example, one can show that there is no quantifier-free formula that is equivalent to the formula $\exists x\ x + x = y$, which says y is even. However, we can *extend* the signature so that it admits quantifier elimination. For each $c \in \mathbb{N}$, we will introduce a unary predicate $c|\cdot$ such that $c|x$ is true if x has a value that is a multiple of c. This extended logic does admit quantifier elimination, and leads to a decision procedure.

Another important theory that admits quantifier elimination is $\mathrm{Th}((\mathbb{R}, 0, 1, +, \times, <))$ and is decidable. This theorem is basically due to Tarski, and is called *Tarski-Seidenberg theorem*.

Chapter 5
Lower Bounds for the Validity Problem

Church-Turing Theorem and Trakhtenbrot's Theorem

The classical decision problem or *Entscheidungsproblem* is the following: Given a sentence φ over signature τ, determine if φ is valid. The problem was popularized by David Hilbert (*das Entscheidungsproblem*, or *the decision problem*), in an attempt to lead towards the formalization of mathematics. However, what *computation* meant was not clear then. These were resolved in 1936, when Church postulated that computability is captured by a class of functions using recursion schemes, and proved that the classical decision problem was not solvable using this notion of computability. A few months later, Alan Turing, in his paper that introduced Turing machines (and started the field of theoretical computer science, or even computer science), also examined the *Entscheidungsproblem* (mentioned in the title of the paper), and showed validity of first-order logic is undecidable. Soon people realized that the notions of computing defined by Turing and Church were the same — Turing in fact showed equivalence in his paper — and the Church-Turing postulate was that the notion of computability coincided with the notion of computability defined by λ-calculus and Turing machines. The undecidability of the *Entscheidungsproblem* is credited now to both Church and Turing.

The classical decision problem, in fact, turns out to be RE-complete (and hence undecidable). We will prove the hardness of this problem in this chapter. Membership in RE will be shown later in what essentially constitutes proving Gödel's completeness theorem. In fact what we will show, which is the content of the completeness theorem, is that for any *recursive* set of sentences Γ and sentence φ, the problem of determining if $\Gamma \models \varphi$ is in RE.

In this chapter, we will also consider another problem, namely that of validity in *finite* models. That is, given a sentence φ over a signature τ, determine if for every *finite* τ-structure $\mathcal{A}$, $\mathcal{A} \models \varphi$. We will show that this problem of validity in finite models is coRE-complete. This result is due to Trakhtenbrot. The reason for considering this problem here is because the proof of hardness is similar to showing the hardness of the classical decision problem. coRE-hardness of validity in finite models implies that checking validity in finite models is not in RE. Consequently, there is no proof systems to establish validity in finite models! This fundamental

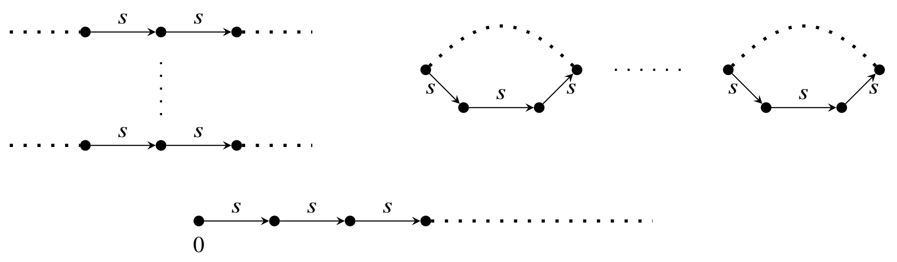

Fig. 5.1 Pictorial representation of structures satisfying (Succ) (or the sentence φ_{num}). Such structures must have a subset that is isomorphic to $\mathbb{N}$. They may have additional elements that form s-cycles or s-chains that are isomorphic to $\mathbb{Z}$ (integers).

incompleteness result is easier to understand than the incompleteness result for arithmetic (Gödel's first incompleteness theorem) which we will see later.

5.1 Number Lines

Before looking at the proof of hardness, let us informally discuss some of the challenges in solving the validity problem/classical decision problem, and introduce some ideas that are central to both hardness proofs. Notice that the RE-hardness of the validity problem means that the problem of checking the *non-validity* of sentence φ, or in fact the satisfiability of $\neg\varphi$, is not recursively enumerable. This, on first reading, sometimes seems surprising. Recall that to show that φ is not valid, we need to demonstrate a structure $\mathcal{A}$ in which φ does not hold, i.e., $\mathcal{A} \models \neg\varphi$. Can't the RE procedure for non-validity simply nondeterministically guess a structure $\mathcal{A}$ and checking if $\mathcal{A} \models \neg\varphi$? If one can prove that φ is not valid if and only if there is a *finite* structure $\mathcal{A}$ such that $\mathcal{A} \models \neg\varphi$, then this would indeed by a RE algorithm for non-validity. Unfortunately, it is easy to see that this not true, i.e., there are sentences ψ that satisfiable, but only in structures that are infinite. Let us look at an example.

Example 5.1 Consider the signature $\tau = \{0, s\}$, where 0 is a constant, and s is a unary function standing for *successor*. Consider the sentence

$$\varphi_{\mathsf{succ}} = (\forall x \neg(s(x) = 0)) \wedge (\forall x \forall y (s(x) = s(y)) \rightarrow (x = y) \qquad \text{(Succ)}$$

which says that s is an injective function and that the constant 0 is not the successor of any element. Consider a structure $\mathcal{A}$ in which φ_{succ} holds. Since 0 is not the successor of any element, it follows that $0 \neq s(0)$ in $\mathcal{A}$. Continuing, since s is injective, $s(s(0)) \neq s(0) \neq 0$, $s(s(s(0))) \neq s(s(0))$ ($\neq s(0) \neq 0$), and so on. We can, therefore, show by induction, that for any $i \neq j$, $s^i(0) \neq s^j(0)$ (where $s^i(0)$ is i applications of s to 0, assuming that $s^0(0) = 0$). Thus, $\mathcal{A}$ must be infinite, because there must be a subset of its universe that is isomorphic to the natural numbers.

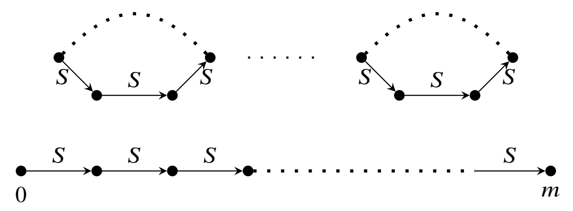

Fig. 5.2 Pictorial representation of finite structures satisfying $\varphi_{\mathsf{fin-num}}$. Such structures must have a subset that is isomorphic to an initial segment of $\mathbb{N}$. They may have additional elements that form S-cycles.

Example 5.1 argues that any structure $\mathcal{A}$ satisfying φ_{succ} (Succ) must have a subset of its universe that is isomorphic to the natural numbers. However, the universe of $\mathcal{A}$ may have additional elements. The function s on these elements may induce sub-structures that are isomorphic to $\mathbb{Z}$ (integers), or to cycles. A pictorial representation of such a structure is shown in Fig. 5.1 , where the number of additional cycles and $\mathbb{Z}$-chains could be 0, finite, or infinite.

Having structure, a subset of whose universe is isomorphic to $\mathbb{N}$ is very useful. This part of the universe can be used to model time or the number of steps of a Turing machine. It can also be used to model the indices of tape cells. If the tape symbols are encoded by numbers, then elements of this part of the universe can also be used to model tape symbols. Our reduction will use structures that have a “number line” as a sub-structure.

However, instead of using using a unary successor function to create a number line, we will find it more convenient to consider structures that have a binary relation S that will represent the *graph* of the successor function s. That is, our signature will be $\{0, S\}$, where 0 is a constant as before, and S is a binary relation. We will want our binary relation S to satisfy the following sentences.

$$\forall x \exists y\ S(x, y) \qquad \text{(Serial)}$$
$$\forall x \forall y \forall z\ (S(x, y) \wedge S(x, z)) \to (y = z) \qquad \text{(Functional)}$$
$$\forall x\ \neg S(x, 0) \qquad \text{(Zero)}$$
$$\forall x \forall y \forall z\ (S(x, z) \wedge S(y, z)) \to (x = y) \qquad \text{(Injective)}$$

The first two sentences state that S is the graph of a function, the third sentence states that 0 is not the successor of any element, and the last sentence states that S is the graph of an injective function. We will denote the conjunction of these 4 sentences as φ_{num}. As observed before, structures satisfying φ_{num} will look as shown in Fig. 5.1 .

Our reason for using a successor relation S, as opposed to a successor function s, to encode number lines is because this gives us the flexibility to use *partial functions* to encode an *initial segment* of $\mathbb{N}$. This will be used in proving the undecidability of checking validity over finite models (i.e., Trakhtenbrot’s theorem). Consider the sentence

$$\exists m\ (\forall x\ \neg S(m,x)) \wedge (\forall x\ \neg(x=m) \rightarrow (\exists y\ S(x,y))) \qquad \text{(Max)}$$

which says that all elements except a maximum (m) have a successor with respect to relation S. Take $\varphi_{\mathsf{fin-num}}$ to be the conjunction of (Max) , (Functional) , (Zero) , and (Injective) . Let $\mathcal{A}$ be *finite* structure such that $\mathcal{A} \models \varphi_{\mathsf{fin-num}}$. Observe that the maximum m must be an element in the successor chain starting at 0. This is because if m is not on the chain starting at 0, then since all elements except m have a S-successor, the chain starting at 0 will be infinite as argued in Example 5.1, and $\mathcal{A}$ would not be finite. Thus any *finite* model of $\varphi_{\mathsf{fin-num}}$ can be depicted as shown in Fig. 5.2 . We will exploit this in our proof of Trakhtenbrot's theorem.

5.2 Church-Turing Theorem

In this section we will prove the following theorem.

Theorem 5.2 (Church-Turing)

Given a sentence φ over signature τ, the problem of determining if φ is valid is RE*-hard.*

Our proof will reduce the RE-hard language MP to the problem of checking validity. Recall that the universal Turing machine U recognizes the language MP. Without loss of generality, we will make some simplifying assumptions about U. We will assume that U has one work-tape (and an input tape); we can ignore the output tape since we are not computing a function. We assume that the input alphabet of U is $\Sigma = \{0, 1\}$ and the tape alphabet is $\Gamma = \{0, 1, \sqcup, \rhd\}$, where $\sqcup$ is the blank symbol, and $\rhd$ is the left end marker. Let Q be the set of states of U, with q_0 as the initial state, and q_{acc} as the unique accept state. The transition function δ of U is such that it ensures that input head of U never leaves the input portion of the input tape. This is ensured by moving the head to the right (+1) when the left end marker ($\rhd$) is read, and moving the head left (-1) when a blank symbol ($\sqcup$) is read on the input tape. A configuration of U is described by current state, current input and work-tape head positions, and the contents of the work-tape.

Given binary string w, our reduction will construct a sentence φ_w such that w is accepted by U if and only if φ_w is valid. The construction will mimic the ideas in the proof of Theorem 1.23, where the sentence φ_w will describe constraints that a computation of U on w satisfies. φ_w will be sentence over the signature $\tau = \{0, S, \mathsf{State}, \mathsf{InpHd}, \mathsf{TapeHd}, \mathsf{TapeSymb}\}$, where 0 is a constant, $S, \mathsf{State}, \mathsf{InpHd}, \mathsf{TapeHd}$ are binary relations, and $\mathsf{TapeSymb}$ is a ternary relation. The intuition behind these relation symbols in τ is as follows. 0 together with S will encode a number line as in Sect. 5.1 . The remaining relation symbols are used to encode configurations of U: $\mathsf{State}(q,t)$ holds if q is the state at time t; $\mathsf{InpHd}(i,t)$ and $\mathsf{TapeHd}(j,t)$ hold if the input tape head is at cell i and work-tape head is at cell j, at time t; $\mathsf{TapeSymb}(a,i,t)$ if the symbol in cell i at time t on the work-tape is a.

As outlined in Sect. 5.1 , we will use the structure induced by 0 and S to encode time, cell numbers, states, and tape symbols.

Like in the proof of Theorem 1.23, the sentence φ_w will state that the relations State, InpHd, TapeHd, InpSymb, TapeSymb encode the initial configuration at "time 0", configurations at successive times follows the transition function δ, and that the accept state q_{acc} is reached at some point. In order to state these properties conveniently, we will find it convenient to make use of some auxiliary formulas that we first introduce.

- The property "Variable x stores a value which is the ith successor of 0" can be written as

$$i(x) = \exists x_1 \exists x_2 \cdots \exists x_i S(0, x_1) \wedge S(x_1, x_2) \wedge \cdots \wedge S(x_{i-1}, x_i) \wedge (x = x_i).$$

  The set of states Q and the tape alphabet Γ are finite sets. We will assume that each state is q is encoded as a number in $\{0, 1, \ldots |Q| - 1\}$ and symbol a is encoded as a number in $\{0, 1, 2, 3\}$. For $b \in Q \cup \Gamma$, it would be useful to have a formula that says that "variable x stores a value which the encoding of symbol b". Assuming b is encoded by number i, this formula is

$$b(x) = i(x).$$

- For a formula $\psi(x, \overrightarrow{y})$ (with free variables x and $\overrightarrow{y}$), we will find it convenient talk about the formula when x is instantiated by $b \in Q \cup \Gamma$. We can write this as

$$\psi(b, \overrightarrow{y}) = \exists x\ \psi(x, \overrightarrow{y}) \wedge b(x).$$

- For a finite set S of elements (either states or tape symbols), the formula that says that a variable x takes a value in set S can be written as

$$(x \in S) = \bigvee_{b \in S} b(x).$$

- Finally, the property that "there is a unique value for x that satisfies $\psi(x, \overrightarrow{y})$" can be written as

$$\exists! x\ \psi(x, \overrightarrow{y}) = \exists x\ \psi(x, \overrightarrow{y}) \wedge (\forall z\ \psi(z, \overrightarrow{y}) \rightarrow (z = x)).$$

Let us now write a few sentences that capture properties that a valid computation of U on input w must satisfy. We begin with the condition that at time 0 the relations must encode the initial configuration. That is, the work-tap the string $\triangleright$, the heads pointing to cell 0 (which contains $\triangleright$, and the state being the initial state q_0. Thus,

$$\begin{aligned}\varphi_{\mathsf{initial}} = & \mathsf{State}(q_0, 0) \wedge \mathsf{InpHd}(0, 0) \wedge \mathsf{TapeHd}(0, 0) \\ & \wedge (\forall c\ \neg(c = 0) \rightarrow \mathsf{TapeSymb}(\sqcup, c, 0).\end{aligned}$$

We demand that the interpretation of relations is consistent with an encoding of a configuration. That is, at all times, the state, the head positions, and symbols in each cell are unique.

$$\begin{aligned}\varphi_{\text{consistent}} = \forall t\ & (\exists! x\ \mathsf{State}(x,t)) \wedge (\exists! x\ \mathsf{InpHd}(x,t)) \wedge (\exists! x\ \mathsf{TapeHd}(x,t)) \\ & \wedge (\forall c \exists! x\ \mathsf{TapeSymb}(x,c,t)).\end{aligned}$$

Next, we require that configurations at successive time steps are consistent with the transition function. This is the most complicated property to write down. Consider a transition $\delta(p,a,b) = (q, d_{\text{in}}, b', d_w)$, i.e., U when in state p, reading a on the input tape, and b on the work-tape, moves to state q, writes b' on the work-tape, and moves input head in direction d_{in} and work-tape head in direction d_w. Without loss of generality, we will assume that we extend δ so that when U reaches a halting state q, $\delta(q,\cdot,\cdot)$ is defined so that the machine stays in state q; this is so that our relations can be defined for all times. For each such tuple $(p,a,b,q,d_{\text{in}},b',d_w)$, we will have a sentence $\varphi_{(p,a,b,q,d_{\text{in}},b',d_w)}$ which captures when U takes a step according to this transition. To describe this property let us introduce some notation. For a tape symbol a, let S_a denote the positions on the input tape where where symbol a is written. So $S_{\rhd} = \{0\}$, $S_{\sqcup} = \{|w|+1\}$ [1], and $S_a = \{i+1 \mid w[i] = a\}$ when $a \in \{0,1\}$ [2]. Finally, given a direction $d \in \{-1,+1\}$, we write $d(c,c')$ to indicate that cells c and c' are consistent with the head moving in direction d. Thus, $d(c,c') = S(c',c)$ when $d = -1$, and $d(c,c' = S(c,c')$ when $d = +1$. Given all this, we can write $\varphi_{(p,a,b,q,d_{\text{in}},b',d_w)}$ as follows.

$$\begin{aligned}\varphi_{(p,a,b,q,d_{\text{in}},b',d_w)} =& \forall t \forall t' \forall c_{\text{in}} \forall c'_{\text{in}} \forall c_w \forall c'_w \\ & (S(t,t') \wedge d_{\text{in}(c_{\text{in}},c'_{\text{in}}) \wedge d_w(c_w,c'_w)} \wedge \mathsf{State}(p,t) \wedge \mathsf{InpHd}(c_{\text{in}},t) \\ & \quad \wedge \mathsf{TapeHd}(c_w,t) \wedge (c_{\text{in}} \in S_a) \wedge \mathsf{TapeSymb}(b,c_w,t)) \\ & \rightarrow (\mathsf{State}(q,t') \wedge \mathsf{InpHd}(c'_{\text{in}},t') \wedge \mathsf{TapeHd}(c'_w,t') \\ & \quad \wedge \mathsf{TapeSymb}(b',c_w,t') \\ & \quad \wedge (\forall c \forall x\ (\neg(c = c_w) \wedge \mathsf{TapeSymb}(x,c,t)) \rightarrow \mathsf{TapeSymb}(x,c,t'))\end{aligned}$$

Finally, the sentence that says that every move is consistent with U's transition function is given by

$$\varphi_{\text{transition}} = \bigvee_{\delta(p,a,b)=(q,d_{\text{in}},b',d_w)} \varphi_{(p,a,b,q,d_{\text{in}},b',d_w)}$$

The last condition in our sentence φ_w is the one that says that U reaches the accepting state q_{acc}. This is easy to write as

$$\varphi_{\text{accept}} = \exists t\ \mathsf{State}(q_{\text{acc}},t).$$

[1] Since we are assuming that the input head of U moves left when reading $\sqcup$, we can assume that the head never moves beyond the first $\sqcup$ symbol on the input tape. Thus we can take $S_{\sqcup} = \{|w|+1\}$.

[2] Our indices of strings start at position 0. So, $w[0]$ is written in cell 1 as cell 0 contains $\rhd$.

Putting all of this together, φ_w needs to say that if 0 and S encode a number line, the computation starts in the initial configuration and follows the transition function, then U accepts. Using all the sentences we have defined, this is

$$\varphi_w = (\varphi_{\mathsf{num}} \wedge \varphi_{\mathsf{initial}} \wedge \varphi_{\mathsf{consistent}} \wedge \varphi_{\mathsf{transition}}) \to \varphi_{\mathsf{accept}}. \tag{5.1}$$

It is easy to see that, given w, the formula φ_w can be computed by a Turing machine. To complete the proof, we need to argue that the reduction is correct. Let us consider the easy case first. Assume that φ_w is valid. Consider structure $\mathcal{A}$ such $u(\mathcal{A}) = \mathbb{N}$. The constant 0 is interpreted as the number 0, $S(n, n')$ holds exactly when $n' = n + 1$. Next, we interpret the relations State, InpHd, TapeHd, $\mathsf{TapeSymb}$ in manner that is consistent with U's computation on string w. Observe that in such a structure $\mathcal{A}$, φ_{num}, $\varphi_{\mathsf{initial}}$, $\varphi_{\mathsf{consistent}}$, and $\varphi_{\mathsf{transition}}$ all hold. Thus, since φ_w is valid, it must be the case that $\varphi_{\mathsf{accept}}$ also holds in this model. This means that the computation of U on w reaches q_{acc} and $w \in \mathsf{MP}$.

Let us now assume that U accepts w. Let the accepting computation of U be

$$\mathsf{c}_0 \longmapsto \mathsf{c}_1 \longmapsto \cdots \longmapsto \mathsf{c}_m.$$

Our goal is to argue that φ_w is valid. Consider a structure $\mathcal{A}$ in which φ_{num}, $\varphi_{\mathsf{initial}}$, $\varphi_{\mathsf{consistent}}$, and $\varphi_{\mathsf{transition}}$ all hold. We need to argue that $\varphi_{\mathsf{accept}}$ also holds. This becomes challenging because $\mathcal{A}$ may have additional elements that are not part of the main number line starting with 0 (see Fig. 5.1). Unfortunately, first order logic is not expressive enough to ensure that $\mathcal{A}$ *only* contains elements of the main number line. To understand this subtlety, let us attempt to reduce $\overline{\mathsf{MP}}$ to validity. It is tempting to think that all we need to do is to modify the sentence and demand that $\mathsf{State}(q_{\mathsf{acc}}, t)$ does not hold for any t. That is, consider the sentence

$$\psi_w = (\varphi_{\mathsf{num}} \wedge \varphi_{\mathsf{initial}} \wedge \varphi_{\mathsf{consistent}} \wedge \varphi_{\mathsf{transition}}) \to (\forall t\ \neg \mathsf{State}(q_{\mathsf{acc}}, t)). \tag{5.2}$$

The sentence ψ_w is not valid even if U's computation on w is non-halting — consider a structure $\mathcal{B}$ where U's computation faithfully encoded using State, InpHd, TapeHd, and $\mathsf{TapeSymb}$ on the main number line starting from 0 but $\mathsf{State}(q_{\mathsf{acc}}, t)$ is true for t that is not on the main number line!

Coming back, the key to showing $\varphi_{\mathsf{accept}}$ holds in structure $\mathcal{A}$ is to argue that when U accepts w and $\mathcal{A}$ satisfies the properties in the antecedent of φ_w, then $\mathcal{A}$'s interpretation of State, InpHd, TapeHd, and $\mathsf{TapeSymb}$ is faithful to the computation on the main number line. More precisely, let the number i denote the ith successor of 0 in $\mathcal{A}$. We can prove by induction, that the interpretations of $\mathsf{State}(\cdot, i)$, $\mathsf{InpHd}(\cdot, i)$, $\mathsf{TapeHd}(\cdot, i)$ and $\mathsf{TapeSymb}(\cdot, \cdot, i)$ in $\mathcal{A}$ encode the configuration c_i (i.e., the ith configuration of U's computation on w). We leave the proof of this fact as an exercise for the reader. Having proved this, it follows that $\mathsf{State}(q_{\mathsf{acc}}, m)$ must hold in $\mathcal{A}$ and therefore, so does $\varphi_{\mathsf{accept}}$. This completes the proof of Theorem 5.2.

Discussion.

Our proof shows that validity is undecidable even if the signature has one constant, 4 relation symbols and no function symbols. We could strengthen the result to the case when the signature has *only one* relation symbol — the 4 relation symbols in our reduction are modeled as a single relation which has an additional argument whose value determines which of our relations we are talking about. We could also get rid of our constant symbol 0 — we just existential quantify to get the element representing 0. Similarly, we could encode our entire reduction if our signature had only a single function symbol. It is, therefore, hard to find reasonable restrictions on the signature that make checking validity decidable. Note that quantifier alternation does play a critical role in our reduction. One could ask if there are restructed quantifier sequences that lead to decidability. A fairly complete characterization of what is decidable and what is not can be found in the book "The Classical Decision Problem" [**?**].

5.3 Trakhtenbrot's Theorem

In computer science, computational problems often involve finite objects. In the context of validity, it is therefore, natural to ask how difficult is the computational problem where given a sentence φ over structure τ, we are asked to determine if φ is true in all *finite* structures. Given that the reduction in Sect. A.2 crucially relies on the sentence φ_{num} which forces the structure to be infinite, does validity become easier when considering only finite models? Clearly, *satisfiability* problem, which is the complement of the validity problem, is easy on finite structures. To determine if a sentence φ is satisfiable in a finite structure, we can simply enumerate finite models one by one, and check whether φ holds in any of them. For a finite structure $\mathcal{A}$, determining if $\mathcal{A} \models \varphi$ is decidable as we can simply go through our inductive definition of satisfiability to answer this question. Surprisingly, validity on finite structures is a very hard problem.

Theorem 5.3 (Trakhtenbrot)

Given a sentence φ over signature τ, the problem of determining if φ is holds in all finite structures is coRE*-hard.*

Thus, in some sense, Theorem 5.3 says reasoning about finite structures is hard. If we want to prove theorems, then reasoning about infinite structures makes life easier as by Gödel's completeness theorem, when theorems are true (valid), we can build machines that can identify that they are true. But over finite structures, there is no such algorithm. There is no proof system to establish theorems that hold on finite structures.

To prove Theorem 5.3 we will reduce $\overline{\mathsf{MP}}$ to the problem of checking validity in finite models. Recall that from Theorems A.24 and A.26, we can conclude that $\overline{\mathsf{MP}}$ is coRE-hard and hence, establishing such a reduction, will prove Theorem 5.3. Again if U is the universal Turing machine accepting MP, given an input w, our reduction will construct a sentence ρ_w such that w is not accepted by U if and only if ρ_w holds in all finite structures. Notice that our construction of ρ_w must differ from φ_w in (5.1) in some fundamental ways. This is because φ_w is trivially valid in all finite structures — since φ_{num} has no finite models, φ_w is vaccuously true in any finite structure!

Instead of using φ_{num}, we will use $\varphi_{\mathsf{finite-num}}$. Recall that finite structures satisfying $\varphi_{\mathsf{finite-num}}$ look like those shown in Fig. 5.2 , and so they will have a substructure that is isomorphic to an initial segment of $\mathbb{N}$ starting at 0. Notice that the sub-structure starting from 0 has a maximum element, which is the only element that does not have an S-successor. We will make use of this as follows. To say that U does not accept w, we will say

$$\varphi_{\mathsf{not-accept}} = \forall t\ (\forall x\ \neg S(t,x)) \rightarrow \neg\mathsf{State}(q_{\mathsf{acc}},t). \tag{5.3}$$

Notice that $\varphi_{\mathsf{not-accept}}$ requires that the state not be q_{acc} at the unique element of the structure that does not have an S-successor. On input w, our reduction will return the following sentence.

$$\rho_w = (\varphi_{\mathsf{finite-num}} \wedge \varphi_{\mathsf{initial}} \wedge \varphi_{\mathsf{consistent}} \wedge \varphi_{\mathsf{transition}}) \rightarrow \varphi_{\mathsf{not-accept}}. \tag{5.4}$$

The proof that this is a correct reduction, is similar to the proof used in Theorem 5.2. Suppose U accepts w by computation that has k steps. Consider finite structure $\mathcal{A}$ whose universe is $\{0, 1, \ldots k\}$, with the constant 0 being interpreted as the number 0, and $S(i,i')$ holding iff $i' = i + 1$; here k is the unique element that does not have an S-successor. Interpret the relations State, InpHd, TapeHd, and $\mathsf{TapeSymb}$ such that it mimics the accepting computation of U on w. Clearly $\mathcal{A}$ satisfies the antecedent of ρ_w and $\mathsf{State}(q_{\mathsf{acc}},k)$ holds. Thus $\mathcal{A} \not\models \rho_w$ and so ρ_w is not valid. Conversely, suppose U does not accept w. Consider any finite structure $\mathcal{A}$ that satisfies the antecedent of ρ_w. Like in the proof of Theorem 5.2, one can prove by induction that the relations State, InpHd, TapeHd, and $\mathsf{TapeSymb}$ faithfully encode a prefix of U's computation on w when time t is restricted to take values on sub-structure consisting of 0 and its S-successors. Notice that since U does not accept, the states at any of the times corresponding to this chain of 0 and its successors is going to be q_{acc}. Moreover, since the unique element that does not have a S-successor is guaranteed to be a successor of 0 in any finite model satisfying $\varphi_{\mathsf{finite-num}}$, $\varphi_{\mathsf{not-accept}}$ holds in $\mathcal{A}$. This establishes the correctness of the reduction.

It is worth recalling the discussion in Sect. A.2 about the formula ψ_w in (5.2) and why that does not describe a correct reduction from $\overline{univL}$ to validity. The problem there was that we may have models where $\mathsf{State}(q_{\mathsf{acc}},t)$ holds for t that are not on the primary number line. Hence ψ_w may not be valid even if U does not accept w. Now, we no longer face that problem, because $\varphi_{not-accept}$ checks the state at time (i.e., the maximum) that is guaranteed to be on the main number line in any finite model of $\varphi_{\mathsf{finite-num}}$.

Since validity and satisfiability are undecidable, a simple corollary of Theorem 5.3 is that for sentences satisfiable in finite models, there is not computable bound on the size of the model.

Corollary 5.4 *There is no computable function f such that for any sentence φ, φ is satisfiable in a finite model if and only if φ is satisfiable in a structure whose universe is bounded by $f(|\varphi|)$.*

Proof Observe that Theorem 5.3 implies that satisfiability in finite models is undecidable. Suppose (for contradiction) there was a computable function f satisfying the properties in the statement of Corollary 5.4. Then checking if a sentence is satisfiable in a finite model would be decidable as follows. Compute $f(|\varphi|)$ and check if φ holds in all finite models of size bounded by $f(|\varphi|)$. This contradicts the undecidability of finite satisfiability and hence the corollary is true. □

Appendix A
Computability and Complexity Theory

A Brief Primer

We will now review the basic definitions and theorems in the area of *computational complexity*, which tries to study various models of computation with the goal of understanding their relative computational power, and classify computational problems in terms of computational resources they need. Here, we will primarily consider time and space as the principal resources we will measure for an algorithm.

Recall that the computational problems one studies in the context of theoretical computer science are usually *decision problems*. Decision problems are those where given an input, one expects a Boolean answer. Typically, input instances are encoded as strings over some alphabet of symbols. A decision problem partitions inputs into those for which the expected answer is “yes”/”true” and those for which the answer is “no”/”false”. Therefore, a decision problem is often identified with a *language*, or a collection of strings, namely, those for which the problem demands a “yes” answer. Similarly, the machines we will define, will answer “yes”/”accept” or “no”/”reject” on input strings, and we associate a language $\mathbf{L}(M)$ with machine M, which is the collection of all strings it accepts. Given this interpretation of problems and machines, we will typically say that a machine M solves a problem L (or rather *accepts/recognizes*) if $L = \mathbf{L}(M)$, i.e., M answers “yes” on exactly the inputs that the problem demands the answer to be “yes”.

The main model of computation that we will consider is that of a Turing machine. However before introducing this model, let us recall some of the notation on strings and languages that we will use.

Alphabet, Strings, and Languages.

An *alphabet* Σ is a finite set of elements. A (finite) *string* over Σ is a (finite) sequence $w = a_0 a_1 \cdots a_k$ over Σ (i.e., $a_i \in \Sigma$, for all i). The *length* of a string $w = a_0 a_1 \cdots a_k$, denoted $|w|$, is the number of elements in it, which in this case is $k + 1$. The unique string of length 0, called the *empty string*, will be denoted by ε. For a string $w = a_0 a_1 \cdots a_k$, the ith symbol of the string a_i will be denoted

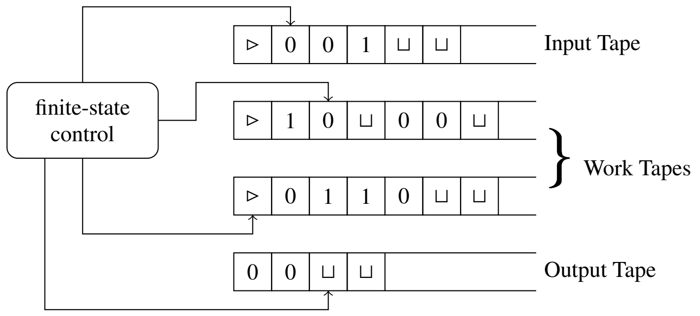

Fig. A.1 Turing machine with a read-only input tape, finitely many read/write worktapes, and a write-only output tape.

as $w[i]$ [1]. For strings $u = a_0a_1 \cdots a_k$ and $v = b_0b_1 \cdots b_m$, their *concatenation* is the string $uv = a_0a_1 \cdots a_kb_0b_1 \cdots b_m$. The set of all (finite) strings over Σ is denoted by Σ^*; we will sometimes use Σ^i to denote the set of strings of length i. A *language* A is a set of strings, i.e., $A \subseteq \Sigma^*$. Given languages A, B, their concatenation $AB = \{uv \mid u \in A,\ v \in B\}$. For a language A, $A^0 = \{\varepsilon\}$, and A^i denotes the i-fold concatenation of A with itself, i.e., $A^i = \{u_1u_2 \cdots u_i \mid \forall j.\ u_j \in A\}$. Finally, the *Kleene closure* of a language A, is $A^* = \bigcup_{i\geq 0} A^i$.

A.1 Turing Machines

We now recall the definition of a Turing machine. Since we will use this model to define the time and space bounds during a computation, as well as define computable functions, the most convenient model to consider is that of a multi-tape Turing machine shown in Fig. A.1 . Such a model has a read-only *input* tape, a write-only *output* tape and finitely many read/write *work* tape, and a write-only *output* tape. Intuitively, the machine works as follows. Initially, the input string is written out on the input tape, and all the remaining tapes are *blank*. The tape heads are scanning the leftmost cell of each tape, which we will refer to as cell 0. This cell contains a special symbol $\rhd$ in every take except the output tape. This is the *left end marker*, which helps the machine realize which cell is the leftmost cell. We will assume these cells are never overwritten by any other symbol, and whenever $\rhd$ is read on a particular tape, the tape head of the Turing machine will move right. At any given step of the Turing machine does the following. Based on the current state of its finite control, and symbols scanned by each tape head, the machine will change the state of its finite control, write new symbols on each of the work tapes, and move it's heads on the input and work tapes either one cell to the left or one cell to the right. During the step, the machine may also choose to write some symbol on its output tape. If

[1] Here we are assuming the the 0th symbol is the "first".

it writes something on the output tape, then the output tape head moves one cell to the right. If it does not write anything, then the output tape head does not move. We will assume that the machine has two special *halting* states — q_{acc} and q_{rej} — with the property that the machine cannot take any further steps from these states. These are captured in the formal definition of deterministic Turing machines below.

Definition A.1 A *deterministic Turing machine with k-work tapes* is a tuple $M = (Q, \Sigma, \Gamma, \delta, q_0, q_{\mathsf{acc}}, q_{\mathsf{rej}}, \sqcup, \triangleright)$ where

- Q is a finite set of control states
- Σ is a finite set of input symbols
- $\Gamma \supseteq \Sigma$ is a finite set of tape symbols. We assume that $\{\sqcup, \triangleright\} \subseteq \Gamma \setminus \Sigma$.
- $q_0 \in Q$ is the initial state
- $q_{\mathsf{acc}} \in Q$ is the accept state
- $q_{\mathsf{rej}} \in Q$ is the reject state, with $q_{\mathsf{rej}} \neq q_{\mathsf{acc}}$, and
- $\delta : (Q \setminus \{q_{\mathsf{acc}}, q_{\mathsf{rej}}\}) \times \Gamma^{k+1} \to Q \times \{-1, +1\} \times (\Gamma \times \{-1, +1\})^k \times (\Gamma \cup \{\varepsilon\})$ is the transition function; here -1 indicates moving the head one position to the left and $+1$ indicates moving the head one position to the right. For $\delta(p, \gamma_0, \gamma_1, \ldots \gamma_k) = (q, d_0, \gamma'_1, d_1, \gamma'_2, d_2, \ldots, \gamma'_k, d_k, o)$, for any $i \in \{0, 1, \ldots k\}$, if $\gamma_i = \triangleright$ then $\gamma'_i = \triangleright$ and $d_i = +1$.

We will now formally describe how the Turing machine computes. For this we begin by first identifying information about the Turing machine that is necessary to determine it's future evolution. This is captured by the notion of a *configuration*. A single step of a Turing machine depends on all the factors that determine which transition is taken. This clearly includes the control state, and the symbols being read on the input tape and the work tape. However this is not enough. The contents of the work tape change, and what is stored influences what will be read in a future step. Thus we need to know what is stored in each cell of the work tape. Since the input tape is read-only, its contents remain static and so we don't need to carry around its contents. We also need to know the position of each tape head because that determines what is read in this step, how the contents of a tape will change based on the current step, and what will be read in the future as the heads move. Because of all of these observations, a configuration of a Turing machine is taken to be the control state, the position of the input head, the contents of the work tape, and the position of the work tape head. The work tape contents and head position is often represented as a single string where a special marker indicates the head position. These are captured formally by the definition below.

Definition A.2 (Configurations)

A *configuration* c of a Turing machine $M = (Q, \Sigma, \Gamma, \delta, q_0, q_{\mathsf{acc}}, q_{\mathsf{rej}}, \sqcup, \triangleright)$ is a member of the set $Q \times \mathbb{N} \times (\Gamma^*\{*\}\Gamma\Gamma^*\sqcup^\omega)^k$ [2], where we assume that $* \notin \Gamma$ indicates the position of the head. For example, a configuration $\mathsf{c} = (q, i, u_1 * a_1 v_1 \sqcup^\omega, u_2 * a_2 v_2 \sqcup^\omega)$

[2] $\sqcup^\omega$ is an *infinite* sequence of blank symbols. Recall that almost all cells contain $\sqcup$, and so the tape contents are a string of the form $u\sqcup^\omega$, where u is initial portion of the tape containing some non-blank symbols.

is the configuration of a 2-work tape Turing machine, whose control state is currently q, the input head is scanning cell i, work tape i ($i \in \{1, 2\}$) contains u_i to left of the head, head is scanning symbol a_i and $v_i \sqcup^\omega$ are the contents of cells to the right of the head.

The *initial configuration* (the configuration of the Turing machine when it starts) is $(q_0, 0, * \triangleright \sqcup^\omega, \ldots, * \triangleright \sqcup^\omega)$. An *accepting configuration* is a member of the set $\{q_{\mathsf{acc}}\} \times \mathbb{N} \times (\Gamma^* \{*\} \Gamma\Gamma^* \sqcup^\omega)^k$. In other words, it is a configuration whose control state is q_{acc}. A *halting configuration* is a configuration whose control state is either q_{acc} or q_{rej}, i.e., it is a member of the set $\{q_{\mathsf{acc}}, q_{\mathsf{rej}}\} \times \mathbb{N} \times (\Gamma^* \{*\} \Gamma\Gamma^* \sqcup^\omega)^k$.

Having defined configurations, we can formally define how configurations change in a single step of the Turing machine. We begin by defining a function that updates the work tape. For a work tape $u * av\sqcup^\omega$, $\mathsf{upd}(u * av\sqcup^\omega, b, d)$ is the resulting work tape when b is written and the head is moved in direction d. This can be formally defined as

$$\mathsf{upd}(u * av\sqcup^\omega, b, d) = \begin{cases} ub * \sqcup\sqcup^\omega & \text{if } d = +1 \text{ and } v = \varepsilon \\ ub * cv'\sqcup^\omega & \text{if } d = +1 \text{ and } v = cv' \\ u' * cbv\sqcup^\omega & \text{if } d = -1 \text{ and } u = u'c \end{cases}$$

Recall also that for a finite string $w \in \Gamma^*$, $w[i]$ denotes the ith symbol in the string. We can extend this notion to tape contents that are sequences of the form $w\sqcup^\omega$ as follows.

$$w \sqcup^\omega [i] = \begin{cases} w[i] & \text{if } i < |w| \\ \sqcup & \text{otherwise} \end{cases}$$

Definition A.3 (Computation Step)

Consider configurations $\mathsf{c}_1 = (q_1, i_1, u_1 * a_1v_1, \ldots u_k * a_kv_k)$ and $\mathsf{c}_2 = (q_2, i_2, t_1, \ldots t_k)$ of Turing machine $M = (Q, \Sigma, \Gamma, \delta, q_0, q_{\mathsf{acc}}, q_{\mathsf{rej}}, \sqcup, \triangleright)$. Let the input string be w. We say $\mathsf{c}_1 \overset{o}{\longmapsto} \mathsf{c}_2$ (machine M moves from configuration c_1 to c_2 in one step and writes o on the output tape) if the following conditions hold. Let $\delta(q_1, w \sqcup^\omega [i_1], a_1, \ldots a_k) = (p, d_0, b_1, d_1, \ldots b_k, d_k)$. Then,

- $q_2 = p$, and $i_2 = i_1 + d_1$,
- for each i, $t_i = \mathsf{upd}(u_i * a_iv_i, b_i, d_i)$

When the output symbol written during a step is not important, we will write $\mathsf{c}_1 \longmapsto \mathsf{c}_2$ to indicate a step from c_1 to c_2.

Having defined how the configuration of a Turing machine changes in each step, we can define the result of a computation on an input.

Definition A.4 (Computation)

A *computation* of Turing machine M on input w, is a sequence of configurations $\mathsf{c}_1, \mathsf{c}_2, \ldots \mathsf{c}_m$ such that c_1 is the initial configuration of M, and for each i, $\mathsf{c}_i \longmapsto \mathsf{c}_{i+1}$.

Definition A.5 (Acceptance)

An input w is *accepted* by Turing machine M if there is a computation $\mathsf{c}_1, \mathsf{c}_2, \ldots \mathsf{c}_m$ such that c_m is an accepting configuration.

The *language recognized/accepted* by M is $\mathbf{L}(M) = \{w \mid w \text{ is accepted by } M\}$. We say that a language $A \subseteq \Sigma^*$ is *accepted/recognized* by M if $\mathbf{L}(M) = A$.

Definition A.6 (Halting)

A Turing machine M is said *halt* on input w if there is a computation $\mathsf{c}_1, \mathsf{c}_2, \ldots \mathsf{c}_m$ such that c_m is a halting configuration.

The Turing machine model we introduced with an output tape can be used to compute (partial) functions as follows.

Definition A.7 (Function Computation)

The *partial function* computed by a Turing machine M, denoted $\mathbf{f}_M$, is as follows. If on input w, M has a halting computation $\mathsf{c}_1 \overset{o_1}{\longmapsto} \mathsf{c}_2 \overset{o_2}{\longmapsto} \cdots \overset{o_{m-1}}{\longmapsto} \mathsf{c}_m$ then $\mathbf{f}_M(w)$ is defined and equal to $o_1 o_2 \cdots o_{m-1}$. On inputs w such that M does not halt, $\mathbf{f}_M(w)$ is undefined.

We say that a (partial) function g is *computable* if there is a Turing machine M such that for every w, $g(w)$ is defined if and only if $\mathbf{f}_M(w)$ is defined, and whenever $g(w)$ is defined, $g(w) = \mathbf{f}_M(w)$.

Most of the time we will be considering Turing machines that accept or recognize languages, rather than those that compute functions. In this context, the symbols written on the output tape don't matter, and so we will often ignore the output tape when describing transitions and computations of such machines.

A.2 Church-Turing Thesis

The Turing machine model introduced in the previous section, is a canonical model to capture mechanical computation. The Church-Turing thesis embodies this statement by saying that anything solvable using a mechanical procedure can be solved using a Turing machine. Our belief in the Church-Turing thesis is based on decades of research in alternate models of computation, which all have turned out to be computationally equivalent to Turing machines. Some of these models include the following.

- Non-Turing machine models: Random Access Machines, λ-calculus, type 0 grammars, first-order reasoning, π calculus, ...
- Enhanced Turing machine models: Turing machines with multiple 2-way infinite tapes, nondeterministic Turing machines, probabilistic Turing machines, quantum Turing machines, ...
- Restricted Turing machine models: Single tape Turing machines, Queue machines, 2-stack machines, 2-counter machines, ...

We will choose to highlight two of these results, that will play a role in our future discussions. The first is the observation that a one work tape Turing machine is computationally as powerful as the multi-work tape model introduced in Definition A.1.

Theorem A.8 *For any k work tape Turing machine M, there is a Turing machine with a single work tape single(M) such that $\mathbf{L}(M) = \mathbf{L}(\mathit{single}(M))$ and $\mathbf{f}_M = \mathbf{f}_{\mathit{single}(M)}$* [3].

Proof of Theorem A.8 can be found in any standard textbook and its precise details are skipped. The idea behind the proof is as follows. The single work tape machine single(M) will simulate the steps of the k-work tape machine M on any input. But in order to simulate M, single(M) needs to keep track of M's configuration at each step. That means keeping track of M's state, its work tape contents, and its tape head. This single(M) accomplishes by storing M's state in its own state, and the contents of all k work tapes of M (including the head positions) on the single work tape of single(M). In general, cell i of the single work tape, stores cell $(i \div k) + 1$ of tape $i \bmod k$; here $i \div m$ denotes the quotient when i is divided by m and $i \bmod m$ denotes the remainder. Then to simulate a single step of M, single(M) will make multiple passes over its single work tape, to first identify the symbols on each tape read by M to determine the transition to take, and then update the contents of the tape according to the transition.

The second result relates to the nondeterministic Turing machines. The Turing machine model introduced in Definition A.1 is *deterministic*, in the sense that at any given time during the computation of the machine, there is at most on possible transition the machine can take. *Nondeterminism*, on the other hand, is the computational paradigm where the computing device, at each step, may have multiple possible transitions to *choose from*. As a consequence, on a given input the machine may have multiple computations, and the machine is said to accept an input, if any one of these computations leads to an accepting configuration. Formally, we can define a nondeterministic Turing machine as follows.

Definition A.9 A *nondeterministic Turing machine* with k work tapes (and one input tape [4]) is a tuple $M = (Q, \Sigma, \Gamma, \delta, q_0, q_{\mathsf{acc}}, q_{\mathsf{rej}}, \sqcup, \rhd)$, where $Q, \Sigma, \Gamma, q_0, q_{\mathsf{acc}}, q_{\mathsf{rej}}, \sqcup, \rhd$ are just like that for deterministic Turing machine, and

$$\delta : (Q \setminus \{q_{\mathsf{acc}}, q_{\mathsf{rej}}\}) \times \Gamma^{k+1} \to 2^{Q\times\{-1,+1\}\times(\Gamma\times\{-1,+1\})^k}$$

is the transition function. The transition function, given current state and symbols read on the input and work tapes, returns a set of possible next states, direction to move the input head, and symbols to be written and direction to move the head in for each work tape.

The definition of configurations, initial configuration, accepting and halting configurations is the same as in Definition A.2. The definitions of computation step (Definition A.3), computation (Definition A.4), and acceptance and language recognized (Definition A.5) are also the same. Hence we skip defining these formally.

[3] For partial functions f and g, we write $f = g$ to indicate that f and g have the same domains (i.e., they are defined for exactly the same elements), and further when $f(x)$ is defined, $f(x) = g(x)$.

[4] We assume there is no output tape for a nondeterministic Turing machine since such machines are used for function computation.

Every deterministic Turing machine is a special kind of nondeterministic machine, namely, one which has the property that at each time step there is at most one transition enable. One of the important results concerning nondeterministic Turing machines is that the converse is also true, i.e., nondeterministic Turing machines are not more powerful than deterministic Turing machines.

Theorem A.10 *For every nondeterministic Turing machine N, there is a deterministic Turing machine $det(N)$ such that* $\mathbf{L}(N) = \mathbf{L}(det(N))$.

A detailed proof of Theorem A.10 is skipped. It can be found in any standard textbook in theory of computation. The broad idea behind the result is the observation that once the length of computation, and the nondeterministic choices at each step are fixed, a deterministic machine can simulate N for that length, on those choices. Thus, the deterministic Turing machine $\det(N)$ simulates N for increasingly longer computations, and for each length, $\det(N)$ will cycle through all possible nondeterministic choices at each step. If any of these computations is accepting for N, then $\det(N)$ will halt and accept.

A.3 Recursive and Recursively Enumerable Languages

The Church-Turing thesis establishes the canonicity of the Turing machine as a model of mechanical computation. The collection of problems solvable on Turing machines is, therefore, worthy of study. Recall that when a Turing machine M is run on an input string w there are 3 possible outcomes — M may (halt and) accept w, M may (halt and) reject w, or M may not halt on w (and therefore not accept). Depending on how a Turing machine behaves we can define two different classes of problems solvable on a Turing machine.

Definition A.11 A language A is *recursively enumerable/semi-decidable* if there is a Turing machine M such that $A = \mathbf{L}(M)$.

A language A is *recursive/decidable* if there is a Turing machine M that halts on *all* inputs and $A = \mathbf{L}(M)$.

Observe that when a problem A is recursive/decidable, it has a special algorithm that solves it and in addition always halts, i.e., on inputs not in A, this algorithm explicitly rejects. Thus, by definition, every recursive language is also recursively enumerable.

Proposition A.12 *If A recursive then A is recursively enumerable.*

We will denote the collection of recursive languages as REC and the collection of all recursively enumerable languages as RE; thus, Proposition A.12 can be seen as saying that $\mathsf{REC} \subseteq \mathsf{RE}$. The collection of recursive and recursively enumerable languages enjoy some closure properties that are worth recalling.

Theorem A.13 REC *is closed under all Boolean operations while* RE *is closed under monotone Boolean operations. That is,*

- *If* $A, B \in \mathsf{RE}$*, then* $A \cup B$ *and* $A \cap B$ *are also in* RE.
- *If* $A, B \in \mathsf{REC}$*, then* $\overline{A}$*,* $A \cup B$*, and* $A \cap B$ *are all in* REC.

Proof We will focus on the two most interesting observations in Theorem A.13; the rest we leave as an exercise for the reader. The first observation we will prove is the closure of RE under union. Let us assume M_A and M_B are Turing machines recognizing A and B, respectively. The computational problem $A \cup B$ asks one to determine if a given input string w belongs to either A or B. We could determine membership in A and B by running M_A and M_B, respectively, but we need to be careful about *how* we run M_A and M_B. Suppose we choose to first run M_A on w and *then* run M_B on w, then we could run into problems. For example, consider the situation where M_A does not halt on w, but $w \in B$. Then, running M_A followed by M_B will never run M_B and therefore never accept, even though $w \in A \cup B$. Switching the order of running M_A and M_B also does not help. What one needs to instead do is, to run M_A and M_B *simultaneously* on w. How does one M_A and M_B at the same time? There are many ways to achieve this. One way is to initially run one step of M_A and then one step of M_B on w from the initial configuration. If either them accept, the algorithm for $A \cup B$ accepts. If not, it will run M_A for two steps, and M_B for two steps, again starting from the respective initial configurations. Again, the algorithm for $A \cup B$ accepts if either simulation accepts. If not the computations of M_A and M_B are increased by one more step, and this process continues, until at some point one of them accepts.

The second result we would like to focus on is the observation that REC is closed under complementation. Let $A \in \mathsf{REC}$ and let M be a Turing machine that halts on all inputs and $\mathbf{L}(M) = A$. The algorithm $\overline{M}$ for $\overline{A}$, runs M on input w, and if M accepts it rejects and if M rejects then it accepts. Notice that $\mathbf{L}(\overline{M}) = \overline{A}$ only because M halts on all inputs — if M does on halt on (say) w, then $w \in \overline{A}$ but $\overline{M}$ would never accept w! □

The following theorem is a useful way to prove that a problem is decidable.

Theorem A.14 *A is recursive if and only if A and* $\overline{A}$ *are recursively enumerable.*

Proof If $A \in \mathsf{REC}$ then $\overline{A} \in \mathsf{REC}$ by Theorem A.13. Then both A and $\overline{A}$ are recursively enumerable by Proposition A.12.

Conversely, suppose A and $\overline{A}$ are recognized by M_A and $M_{\overline{A}}$ respectively. The recursively algorithm M for A, on a given input w, will run both M_A and $M_{\overline{A}}$ simultaneously (as in the proof of Theorem A.13), and accept if either M_A accepts or $M_{\overline{A}}$ rejects. Notice, that any given input w belongs to either A or $\overline{A}$, and therefore at least one out of M_A and $M_{\overline{A}}$ is guaranteed to halt on each input. Therefore M will always halt. □

Encodings.

Every object (graphs, programs, Turing machines, etc.) can be encoded as a binary string. The details of the encoding scheme itself are not important, but it should be simple enough that the data associated with the object should be easily recoverable by reading the binary encoding. For example, one should be able to reconstruct the vertices and edges of a graph from its encoding, or one should be able to reconstruct the states, transitions, etc. of a Turing machine from its encoding. For a list of objects $O_1, O_2, \ldots O_n$, we will use $\langle O_1, O_2, \ldots O_n \rangle$ to denote their binary encoding. In particular, for a Turing machine M, $\langle M \rangle$ is its encoding as binary string. Conversely, for a binary string x, M_x denotes the Turing machine whose encoding is the string x.

Once we establish an encoding scheme, we can construct a *Universal Turing machine*, which is an *interpreter* that given an encoding of a Turing machine M and an input w, can simulate the execution of M on the input string w. This is an extremely important observation that establishes the recursive enumerability of the membership problem for Turing machines.

Theorem A.15 *There is a Turing machine U (called the universal Turing machine) that recognizes the language* $\mathsf{MP} = \{\langle M, w \rangle \mid w \in \mathbf{L}(M)\}$. *In other words,* $\mathsf{MP} \in \mathsf{RE}$.

Not every decision problem/language is recursively enumerable. Using Cantor's diagonalization technique, one can establish the following result.

Theorem A.16 *The language* $\overline{\mathsf{K}} = \{x \mid x \notin \mathbf{L}(M_x)\}$ *is not recursively enumerable.*

Proof The proof of Theorem A.16 relies on a diagonalization argument to show that the language of every Turing machine differs from $\overline{\mathsf{K}}$, and therefore $\overline{\mathsf{K}}$ is not recursively enumerable.

Consider an arbitrary Turing machine M_x whose encoding as a binary string is x. We will show that $\mathbf{L}(M_x) \neq \overline{\mathsf{K}}$, thereby proving the theorem. Observe that if $x \in \mathbf{L}(M_x)$ then by definition $x \notin \overline{\mathsf{K}}$ and if $x \notin \mathbf{L}(M_x)$ then again by definition $x \in \overline{\mathsf{K}}$. Therefore $x \in (\overline{\mathsf{K}} \setminus \mathbf{L}(M_x)) \cup (\mathbf{L}(M_x) \setminus \overline{\mathsf{K}}) \neq \emptyset$. □

A.4 Reductions

Theorem A.16 is the first result that establishes that there are problems that are computationally difficult. Further results on the computational hardness of problems are usually established using the notion of *reductions*. Reductions demonstrate how one problem can be converted into another in such a way that a solution to the second problem can be used to solve the first. Formally, it is defined as follows.

Definition A.17 A *(many-one/mapping) reduction* from A to B is a computable (total) function $f : \Sigma^* \to \Sigma^*$ such that for any input string w,

$$w \in A \text{ if and only if } f(w) \in B$$

In this case, we say A is *(many-one/mapping) reducible* to B and we denote it by $A \leq_m B$.

Since many-one/mapping reductions are the only form of reduction we will study, we will drop the adjective "many-one" and "mapping" and simply call these reductions. Let us look at a couple of examples of reductions.

Example A.18 Let us consider the complement of MP, i.e., $\overline{\mathsf{MP}} = \{\langle M, w\rangle \mid w \notin \mathbf{L}(M)\}$. One can show that $\overline{\mathsf{K}} \leq_m \overline{\mathsf{MP}}$ as follows. The reduction f is the following function: $f(x) = \langle M_x, x\rangle$.

To prove that f is a reduction, we need to argue two things. First that f is computable, i.e., we need to come up with a Turing machine M_f that always halts and produces the string $f(x)$ on input x. In this example, to construct $f(x)$, we simply need to "copy" the string x which clearly is a computable function. Second we need to argue that $x \in \overline{\mathsf{K}}$ iff $f(x) \in \overline{\mathsf{MP}}$. This can be argued as follows: $x \in \overline{\mathsf{K}}$ iff $x \notin \mathbf{L}(M_x)$ (definition of $\overline{\mathsf{K}}$) iff $\langle M_x, x\rangle \in \overline{\mathsf{MP}}$ (definition of $\overline{\mathsf{MP}}$) iff $f(x) \in \overline{\mathsf{MP}}$ (definition of f).

Example A.19 Consider the problem

$$\overline{\mathsf{HP}} = \{\langle M, w\rangle \mid M \text{ does not halt on } w\}.$$

We will prove that $\overline{\mathsf{K}} \leq_m \overline{\mathsf{HP}}$.

Given a binary string x, let us consider the following program H_x.

```
$H_x(w)$
  result = $M_x(x)$
  if (result = accept)
    return accept (* on input $w$ *)
  else
    while true do
```

In other words, the program H_x on input w, ignores its input and runs the program M_x on x. If M_x halts and accepts x then H_x halts and accepts w. Otherwise, H_x does not halt. Thus, the program H_x halts on some (all) inputs if and only if $x \in \mathbf{L}(M_x)$.

Let us now describe the reduction from $\overline{\mathsf{K}}$ to $\overline{\mathsf{HP}}$: $f(x) = \langle H_x, x\rangle$. Observe first that f satisfies the properties of a reduction because $x \in \overline{\mathsf{K}}$ iff $x \notin \mathbf{L}(M_x)$ iff H_x does not halt on x (and all input strings) iff $\langle H_x, x\rangle \in \overline{\mathsf{HP}}$. To establish that f is a reduction, we also need to argue that f is computable. On input string x, we need a program that produces the source code for H_x (given above) and copies the string x after the source code. This is clearly computable.

Reductions are a way for one to compare the computational difficulty of problems — if A reduces to B then A is at most as difficult as B, or B is at least as difficult as A. This is formally captured in the following proposition.

Theorem A.20 *If $A \leq_m B$ and B is recursively enumerable (recursive) then A is recursively enumerable (recursive).*

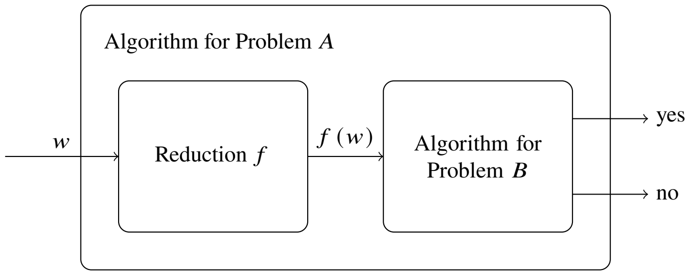

Fig. A.2 Schematic argument for Theorem A.20.

Proof Let f be a reduction from A to B that is computed by Turing machine M_f, and let M_B be a Turing machine that recognizes B. The algorithm for A is schematically shown in Fig. A.2 — on input w, compute $f(w)$ using M_f and run M_B on $f(w)$. Notice that this algorithm always halts if M_B always halts. Thus, if B is recursive then A is also recursive. □

Theorem A.20 can be seen to informally say "if A reduces to B and B is computationally easy then A is computationally easy". It is often used in the contrapositive sense and it is useful to explicitly state this observation.

Corollary A.21 *If $A \leq_m B$ and A is not recursively enumerable (undecidable) then B is not recursively enumerable (undecidable).*

We can use the above corollary to argue the computational hardness of some problems.

Theorem A.22 $\overline{\mathsf{MP}}$ *is not recursively enumerable. Therefore,* MP *is undecidable.*

Proof Example A.18 establishes that $\overline{\mathsf{K}} \leq_m \overline{\mathsf{MP}}$. Together with Theorem A.16 and Corollary A.21, we can conclude that $\overline{\mathsf{MP}}$ is not recursively enumerable. Finally, since $\overline{\mathsf{MP}}$ is not recursively enumerable, Theorem A.14 establishes that MP is not decidable/recursive. □

Since $\mathsf{MP} \in \mathsf{RE}$ (Theorem A.15) and $\overline{\mathsf{MP}} \notin \mathsf{RE}$ (Theorem A.22), we have a witness to the fact that RE is not closed under complementation. Just like *Theorem A.22*, we could establish similar properties for the *halting problem*.

Theorem A.23 $\overline{\mathsf{HP}}$ *is not recursively enumerable. Therefore,* $\mathsf{HP} = \{\langle M, w\rangle \mid M \text{ halts on } w\}$ *is undecidable.*

Proof Follows from Example A.19 and the argument in the proof of Theorem A.22.□

Reductions are transitive and hence a *pre-order*; thus, the use of $\leq$ to denote them is justified.

Theorem A.24 *The following properties hold for reductions.*

- *If $A \leq_m B$ then $\overline{A} \leq_m \overline{B}$.*
- *If $A \leq_m B$ and $B \leq_m C$ then $A \leq_m C$.*

Proof If f is a reduction from A to B, then one can argue that f is also a reduction from $\overline{A}$ to $\overline{B}$. And, if f is a reduction from A to B and g a reduction from B to C then $g \circ f$ is a reduction from A to C. Establishing these observations to prove the theorem is left as an exercise. □

Having found a lens to compare the computational difficulty of two problems (namely, reductions), one can use them to argue that a problem is at least as difficult as a whole collection of problems, or something is the "hardest" problem in a collection. This leads us to notions of hardness and completeness.

Definition A.25 A language A is RE-hard if for every $B \in$ RE, $B \leq_m A$.
A language A is RE-complete if A is RE-hard and $A \in$ RE.

Thus, an RE-complete problem is the hardest problem that is recursively enumerable, while an RE-hard problem is something that is at least as hard as any other RE problem. Are there examples of such problems? It turns out that MP, HP, and K are all RE-complete. We establish this for MP in the following theorem.

Theorem A.26 MP *is* RE-*complete.*

Proof Membership in RE has been established in Theorem A.15. So all we need to prove is the hardness. Let B be any recursively enumerable language, and let M be a Turing machine recognizing B. The reduction from B to MP is as follows: $f(w) = \langle M, w\rangle$. It is easy to see that $w \in B$ iff $w \in \mathbf{L}(M)$ (since M recognizes B) iff $\langle M, w\rangle \in$ MP (definition of MP) iff $f(w) \in$ MP (definition of f). It is also easy to see that f is computable — in order to compute $f(w)$, all we need to do is prepend the source code of M. □

Establishing RE-hardness of a problem is sufficient to guarantee it's undecidability.

Theorem A.27 *If A is* RE-*hard then A is undecidable.*

Proof If A is RE-hard then since MP $\in$ RE, we have MP $\leq_m A$. Since MP is undecidable (Theorem A.22), by properties of a reduction (Corollary A.21) A is undecidable. □

A.5 Complexity Classes

Computational resources needed to solve a problem depend on the size of the input instance. For example, it is clearly easier to compute the sum of two one digit numbers as opposed to adding two 15 digit numbers. The resource requirements of an algorithm/Turing machine are measured as a function of the input size. We will only study time and space as computational resources in this presentation. We begin by defining time bounded and space bounded Turing machines, which are defined with respect to bounds given by functions $T : \mathbb{N} \to \mathbb{N}$ and $S : \mathbb{N} \to \mathbb{N}$ that are non-decreasing, i.e., for all $n \leq m \in \mathbb{N}$, $T(n) \leq T(m)$ and $S(n) \leq S(m)$. Our definitions apply to both deterministic and nondeterministic machines.

Definition A.28 A (deterministic/nondeterministic) Turing machine M is said to run in *time* $T(n)$ if on any input u, *all* computations of M on u take at most $T(|u|)$ steps; here $|u|$ refers to the length of input u.

A (deterministic/nondeterministic) Turing machine M is said to use *space* $S(n)$ if on any input u, *all* computations of M on u use at most $S(|u|)$ work tape cells. In this context, a work tape cell is said to be used if it is written to at least once during the computation. Notice that, if a work tape cell is written multiple times during a computation, it counts as only one cell when measuring the space requirements; thus, work tape cells can be reused without adding to the space bounds.

It is worth examining Definition A.28 carefully. Our requirement for a Turing machine running within some time or space bound applies to *all* computations, whether they are accepting or not. Notice also that the definition is the same for both deterministic and nondeterministic models — in a deterministic machine the *unique* computation on a given input must satisfy the resource bounds, and in a nondeterministic machine, *all* computations on the input must satisfy the bounds. In particular, if a Turing machine (deterministic or nondeterministic) runs within a time bound, then it *halts* in every computation of every input.

Having defined time and space bounded machines, we can define the basic complexity classes which are collections of (decision) problems that can be solved within certain time and space bounds.

Definition A.29 We define the following basic complexity classes.

- A language $A \in \mathsf{DTIME}(T(n))$ iff there is a *deterministic* Turing machine that runs in time $T(n)$ such that $A = \mathbf{L}(M)$.
- A language $A \in \mathsf{NTIME}(T(n))$ iff there is a *nondeterministic* Turing machine that runs in time $T(n)$ such that $A = \mathbf{L}(M)$.
- A language $A \in \mathsf{DSPACE}(S(n))$ iff there is a *deterministic* Turing machine that uses space $S(n)$ such that $A = \mathbf{L}(M)$.
- A language $A \in \mathsf{NSPACE}(S(n))$ iff there is a *nondeterministic* Turing machine that uses space $S(n)$ such that $A = \mathbf{L}(M)$.

Our computational model of Turing machines, and our definitions of time and space bounded computations are robust with respect to constant factors. This observation is captured by two central results in theoretical computer science, namely, the linear speedup and compression theorems. It says that one can always improve the running time or space requirements for solving a problem by a constant factor.

Theorem A.30 (Linear Speedup)

If $A \in \mathsf{DTIME}(T(n))$ (or $A \in \mathsf{NTIME}(T(n))$) and $c > 0$ is any constant, then $A \in \mathsf{DTIME}(cT(n) + n)$ ($A \in \mathsf{NTIME}(cT(n) + n)$).

Proof (Sketch) Let $A = \mathbf{L}(M)$. We will describe a machine M' which will simulate k steps of M in 8 steps; if $k > \frac{8}{c}$, we will get the desired result. M' will have one more work tape, a much larger tape alphabet, and control states than M.

- M' copies the input onto the additional work tape in compressed form: k successive symbols of M will be represented by one symbol in M'. Time taken is n. M' will maintain M's work tape contents in compressed form on the second work tape as well.
- M' uses the additional work tape as "input tape". The head positions of M, within the k symbols represented by current cells, is stored in finite control.

One *basic move* of M' (consisting of 8 steps), will simulate k steps of M as follows.

- At the start of basic move, M' moves its tape heads one cell left, two cells right and one cell left, storing the symbols read in the finite control. Now, M' knows all symbols within the radius of k cells of any of M's tape heads. This takes 4 steps.
- Based on the transition function of M, M' can compute the effect of the next k steps of M.
- Using any additional (at most) 4 steps, M' updates the contents of its tapes as a result of the k steps, and moves the heads appropriately. □

Theorem A.31 (Linear Compression)

If $A \in \mathsf{DSPACE}(S(n))$ *(or* $A \in \mathsf{NSPACE}(S(n))$*) and* $c > 0$ *is any constant then* $A \in \mathsf{DSPACE}(cS(n))$ *(*$A \in \mathsf{NSPACE}(cS(n))$*).*

Proof Increase the tape alphabet size and store work tape contents in compressed form as in Theorem A.30. □

Theorems A.30 and A.31 suggest that when analyzing the time and space requirements of an algorithm we can ignore constant terms. This leads to the use of the order notation.

Definition A.32 Consider functions $f : \mathbb{N} \to \mathbb{N}$ and $g : \mathbb{N} \to \mathbb{N}$.

- $f(n) = O(g(n))$ if there are constants c, n_0 such that for $n > n_0$, $f(n) \leq cg(n)$. $g(n)$ is an *asymptotic upper bound.*
- $f(n) = \Omega(g(n))$ if there are constants c, n_0 such that for $n > n_0$, $f(n) \geq cg(n)$. $g(n)$ is an *asymptotic lower bound.*

The complexity classes identified in Definition A.29 are a very fine classification of problems. They include complexity classes whose classification of problems is sensitive to our use of Turing machines as a model of computation. Ideally we would like to study complexity classes such that if a problem is classified in a certain class then that classification should be "platform independent". That is, whether we choose to study complexity on Turing machines or Random Access Machines, our observations should still hold. They should also be invariant under small changes to the Turing machine model, like changing the number of work tapes, alphabet, nature of the tapes, etc. There is a strengthening of the Church-Turing thesis, called the *invariance thesis* articulated by Church, that underlies our belief in the robustness of the Turing machine model, subject to small changes in the time and space bounds. It says that

> Any effective, mechanistic procedure can be simulated on a Turing machine using the same space (if space is $\geq \log n$) and only a polynomial slowdown (if time $\geq n$)

In addition to the requirement that complexity classes be robust to changes to the computational platform, we would like the classes to be closed under function composition — making function/procedure calls to solve sub-problems is a standard algorithmic tool, and we would like the complexity to remain the same as long as the sub-problems being solved are equally simple. Finally, we would like our complexity classes to capture natural, "interesting", real-world problems. For these reasons, we typically study the following complexity classes that provide a coarser classification of problems than that provided in Definition A.29.

Definition A.33 Commonly studied complexity classes are the following.

$$\begin{array}{ll}
\mathsf{L} = \mathsf{DSPACE}(\log n) & \mathsf{NL} = \mathsf{NSPACE}(\log n) \\
\mathsf{P} = \cup_k \mathsf{DTIME}(n^k) & \mathsf{NP} = \cup_k \mathsf{NTIME}(n^k) \\
\mathsf{PSPACE} = \cup_k \mathsf{DSPACE}(n^k) & \mathsf{NPSPACE} = \cup_k \mathsf{NSPACE}(n^k) \\
\mathsf{EXP} = \cup_k \mathsf{DTIME}(2^{n^k}) & \mathsf{NEXP} = \cup_k \mathsf{NTIME}(2^{n^k})
\end{array}$$

In addition to the above classes, for any class C, $\mathsf{co}C = \{A \mid \overline{A} \in C\}$. Please note that $\mathsf{co}C$ is *not* the complement of C but instead is the collection of problems whose complement is in C.

A.6 Relationship between Complexity Classes

We begin by relating time and space complexity classes.

Theorem A.34 $\mathsf{DTIME}(T(n)) \subseteq \mathsf{DSPACE}(T(n))$ *and* $\mathsf{NTIME}(T(n)) \subseteq \mathsf{NSPACE}(T(n))$

Proof A Turing machine can scan at most one new work tape cell in any step. Therefore, the number of work tape cells used during a computation cannot be more than the number of steps. □

Theorem A.35 $\mathsf{DSPACE}(S(n)) \subseteq \mathsf{DTIME}(n \cdot 2^{O(S(n))})$ *and* $\mathsf{NSPACE}(S(n)) \subseteq \mathsf{NTIME}(n \cdot 2^{O(S(n))})$. *In particular, when* $S(n) \geq \log n$, *we have* $\mathsf{DSPACE}(S(n)) \subseteq \mathsf{DTIME}(2^{O(S(n))})$ *and* $\mathsf{NSPACE}(S(n)) \subseteq \mathsf{NTIME}(2^{O(S(n))})$.

We will skip giving a direct proof of Theorem A.35. It will follow from Theorem A.37 and Theorem A.38. An immediate consequence of Theorems A.34 and A.35 are the following relationships between the complexity classes.

Corollary A.36

$$\mathsf{L} \subseteq \mathsf{P} \subseteq \mathsf{PSPACE} \subseteq \mathsf{EXP}$$
$$\mathsf{NL} \subseteq \mathsf{NP} \subseteq \mathsf{NPSPACE} \subseteq \mathsf{NEXP}$$

We will establish relationships between deterministic and nondeterministic complexity classes.

Theorem A.37 $\mathsf{DTIME}(T(n)) \subseteq \mathsf{NTIME}(T(n))$ *and* $\mathsf{DSPACE}(S(n)) \subseteq \mathsf{NSPACE}(S(n))$.

Proof This follows from the fact that, by definition, every deterministic Turing machine is a special nondeterministic Turing machine, namely, those that have exactly one transition enabled from every non-halting configuration. □

Nondeterministic complexity class can also be related to deterministic complexity classes. In fact, we now prove a result that subsumes the containment results for nondeterministic classes established in Theorems A.34 and A.35.

Theorem A.38 $\mathsf{NTIME}(T(n)) \subseteq \mathsf{DSPACE}(T(n))$ *and* $\mathsf{NSPACE}(S(n)) \subseteq \mathsf{DTIME}(n2^{O(S(n))})$.

Proof Let us begin by proving the first inclusion. Consider $A \in \mathsf{NTIME}(T(n))$ and let M be $T(n)$-time bounded nondeterministic machine recognizing A. On any input w of length n, the computations of M can be organized as a tree, and since M runs in time $T(n)$, this tree has height $T(n)$. Now the deterministic algorithm D to solve A will perform a *depth first search* (DFS) on this computation tree of M, constructing this tree as it is explored, and accepting if some node in this computation tree corresponds to an accepting configuration. The space needed by D to perform this DFS is the memory needed to store the call stack. The stack during a DFS keeps track of the path being currently explored in the tree to enable backtracking. Since the computation tree of M is of height $T(n)$, the height of the call stack is also $T(n)$. A naïve implementation of the DFS algorithm will store the sequence of tree vertices on the current path; since in this case each vertex is a configuration of M, these can be represented by strings of length $T(n)$ (as work tape cells cannot exceed $T(n)$ as in Theorem A.34). This gives us a space bound of $T(n)^2$ for algorithm D. However, instead of storing the actual configurations in the computation being currently explored, D can just store the sequence of nondeterministic choices made by M in the current computation. With this information about the nondeterministic choices, D can *reconstruct* the configuration at the end of a sequence of steps, by resimulating M from the beginning — this increases the running time of D, but reduces the space requirements of D which is what we care about for this result. If M has k choices at each step, the stack of D during DFS is simply a k-ary string of length $\leq T(n)$, which means that D is $T(n)$-space bounded.

For the second result, let us consider $A \in \mathsf{NSPACE}(S(n))$ and a nondeterministic Turing machine M that recognizes A in $S(n)$ space. On a given input w of the length n, it is useful to define the notion of a *configuration graph* of M. The configuration graph is a directed graph that has as vertices, configurations of M, and has an edge from c_1 to c_2, if M can move from configuration c_1 to configuration c_2 in one step

$$\begin{tikzcd}
 & & & \mathsf{NP} \arrow[dr] & & & \\
\mathsf{L} \arrow[r] & \mathsf{NL} \arrow[r] & \mathsf{P} \arrow[ur] \arrow[dr] & & \begin{array}{c}\mathsf{PSPACE}\\=\\\mathsf{NPSPACE}\end{array} \arrow[r] & \mathsf{EXP} \arrow[r] & \mathsf{NEXP} \\
 & & & \mathsf{coNP} \arrow[ur] & & &
\end{tikzcd}$$

Fig. A.3 Relationship between Complexity Classes. $\rightarrow$ indicates containment, though whether it is strict is unknown.

given input w. Observe that M accepts w if an accepting configuration is reachable from the initial configuration in this configuration graph. Notice also that since M is $S(n)$-space bounded, the total number of vertices in this graph is $\leq n2^{O(S(n))}$ (see proof of Theorem A.35). Now we can run our favorite graph search algorithm (depth first search or breadth first search) on this configuration graph to see if an accepting configuration is reachable; the graph will be constructed on-the-fly as it is being explored. Such an algorithm (which deterministic), takes time that is linear in the size of the graph, which gives us a $n2^{O(S(n))}$ deterministic algorithm for A. $\square$

Our new observations relating deterministic and nondeterministic complexity classes gives us the following relationships.

Corollary A.39

$$\mathsf{L} \subseteq \mathsf{NL} \subseteq \mathsf{P} \subseteq \mathsf{NP} \subseteq \mathsf{PSPACE} \subseteq \mathsf{NPSPACE} \subseteq \mathsf{EXP} \subseteq \mathsf{NEXP}$$

An important result due to Savitch, relates nondeterministic and deterministic space complexity classes. The interested reader can find its proof in textbooks like [**?**].

Theorem A.40 (Savitch)

For $S(n) \geq \log n$, $\mathsf{NSPACE}(S(n)) \subseteq \mathsf{DSPACE}(S(n)^2)$. *In particular, this means that* $\mathsf{PSPACE} = \mathsf{NPSPACE}$.

Putting all our observations together we get the relationships shown in Fig. A.3 . It is worth observing that for any deterministic complexity class C, $C = \mathsf{co}C$; this is because an algorithm for $\overline{A}$ is to run the deterministic algorithm for A and flip the final answer. Thus, $\mathsf{P} = \mathsf{coP}$. Further, from Theorem A.37, we have $\mathsf{coP} \subseteq \mathsf{coNP}$, giving us the containment $\mathsf{P} \subseteq \mathsf{coNP}$. Finally, due the space hierarchy theorem, we also know that $\mathsf{L} \neq \mathsf{PSPACE}$ and $\mathsf{NL} \neq \mathsf{PSPACE}$, and from the time hierarchy theorem, we know that $\mathsf{P} \neq \mathsf{EXP}$ and $\mathsf{NP} \neq \mathsf{NEXP}$; the hierarchy theorems are beyond the scope of this brief primer.

A.7 P and NP

Cobham-Edmonds Thesis, named after Alan Cobham and Jack Edmonds, asserts that the only computational problems that have "efficient" or "feasible" algorithmic solutions are those that belong to P. In other words, P is the collection of *tractable* computational problems. There are many features of P that justify this view.

- The invariance thesis suggests that any problem in P can be solved in polynomial time on *any* reasonable computational model. Thus, the statement of a problem being efficiently computable is platform independent.
- Most encodings of an input structure are polynomially related in terms of their length. Thus, if a problem is in P for one encoding, it will be in P even if input instances are encoded in a different manner. Therefore, P is insensitive is problem encodings.
- Most natural problems in P have algorithms whose running time is bounded by a low-order polynomial. Thus, their running times are likely to be low for most problem instances.
- The asymptotic growth of polynomials is moderate when compared to the astounding growth of exponential functions. Thus, problems in P are likely to be feasibly solved even on large problem instances.

The crux of the Cobham-Edmonds thesis is that for problem to be solvable in practice, it should have a polynomial time algorithm. Therefore, much effort in the past 50 years has been devoted to understanding the class of problems in P. In particular, can we prove that the complexity classes containing P in Fig. A.3 , like NP and PSPACE, also have efficient solutions, i.e., are contained in P? Or can we say for certain that some problems in NP and PSPACE *cannot* be solved efficiently?

A.7.1 Alternate characterization of NP

We defined NP as the collection of problems that can be solved in polynomial time on a nondeterministic Turing machine. In this section, we will give an alternate definition, namely, as those problems that are efficiently "verifiable".

Definition A.41 A language A is *polynomially verifiable* if there is a $k \in \mathbb{N}$ and a deterministic Turing machine V such that

$$A = \{w \mid \exists p.\ V \text{ accepts } \langle w, p\rangle\}$$

and V takes at most $|w|^k$ steps on input $\langle w, p\rangle$, i.e., V running time is *independent* of the length of p. Here V is called a *verifier* for A, and for $w \in A$, the strings p such that $\langle w, p\rangle$ is accepted by V are called a *proof* of w (with respect to V).

The notion of a language A being polynomial verifiable says that when a string $w \in A$, there is a *proof* p (maybe even more than one) such that w augmented with p "convinces" V, i.e., causes V to accept. However, if $w \notin A$ then there is no proof string p that can convince V of w's membership in A. Notice also that V's running time on input $\langle w, p\rangle$ is independent of the length of p; it always runs in time $|w|^k$ no matter what p is. Now, since in $|w|^k$ steps, V cannot read more that $|w|^k - |w|$ bits of p, we can without loss of generality assume that p is a string whose length is bounded by a polynomial in the length of w. Thus, we could informally say that A is polynomially verifiable, if for any string $w \in A$ there is a "short" proof (of polynomial length) that can be efficiently checked (in polynomial time) by a verifier, and if $w \notin A$ there is no proof that can convince a verifier.

A language being polynomially verifiable is equivalent to a problem having a nondeterministic polynomial time verifier.

Theorem A.42 $A \in \mathsf{NP}$ *if and only if A is polynomially verifiable.*

Proof Consider $A \in \mathsf{NP}$, and let M be nondeterministic Turing machine recognizing A in time n^k for some k. We can assume without loss of generality that M has at most two choices at any given step. The verifier V for A will work as follows. On input $\langle w, p \rangle$, where p is a binary string, it will first copy w onto a work-tape, and compute n^k. It will then simulate M for n^k steps using the work-tape with w as the input tape, taking p to be the sequence of nondeterministic choices. V accepts $\langle w, p \rangle$ if M accepts w with p as the nondeterministic choices. Observe that V is a deterministic algorithm running in $O(n^k)$ time on $|w| = n$. Further $A = \{w \mid \exists p.\, V \text{ accepts } \langle w, p \rangle\}$.

Conversely, suppose V is a polynomial time verifier for A. Suppose V runs in time $|w|^k$ on input $\langle w, p \rangle$. The nondeterministic algorithm M for A will work as follows. On input w, M will guess a string p of length $|w|^k$. Then M will simulate V on w and the guessed string p, accepting if and only if V accepts. It is easy to see that $\mathbf{L}(M) = A$ and M runs in time $O(n^k)$. □

Thus, NP is the collection of all problems A whose membership question has short, efficiently checkable proofs. The question of whether all problems in NP have polynomial time algorithms — whether $\mathsf{P} \overset{?}{=} \mathsf{NP}$ — is thus the question of whether every problem that has a short, efficiently checkable proofs also have the property that these proofs can be *found* efficiently. Phrased in this manner, the likely answer seems to be no. There are also results that seem to suggest that P is likely to be not equal to NP, though a firm resolution of this question has eluded researchers for the past 50 years.

A.7.2 Reductions, Hardness and Completeness

In an effort to resolve the P versus NP question, researchers have tried to identify canonical problems whose study can help address this challenge. The goal is to identify, in some sense, the most difficult problems in NP such that either (a) they are candidate problems that may not have polynomial time algorithms, or (b) finding a polynomial time algorithms for these problems will constructively demonstrate that $\mathsf{P} = \mathsf{NP}$. In order to identify such difficult problems, we need to be able to compare the difficulty of two problems. For this the most convenient technique is that of reductions. Unlike, many-one reductions introduced before, we will require that these be computed in polynomial time.

Definition A.43 A *polynomial time reduction* from A to B is a *polynomial time computable function* f such that for every input w,

$$w \in A \text{ if and only if } f(w) \in B.$$

In such a case we say that A is *polynomial time reducible* to B and is denoted by $A \leq_{\mathsf{P}} B$.

Example A.44 Consider the following problems.

$$\begin{array}{l} \mathsf{SAT} = \{\langle\varphi\rangle \mid \varphi \text{ is in CNF and } \varphi \text{ is satisfiable}\} \\ k\text{−}\mathsf{COLOR} = \{\langle G, k\rangle \mid G \text{ is an undirected graph that can be colored using } k \text{ colors}\} \end{array}$$

Proposition 1.35 shows that for any graph G and $k \in \mathbb{N}$ there is a set of clauses $\Gamma_{G,k}$ such that G is k-colorable if and only if $\Gamma_{G,k}$ is satisfiable. The number of clauses in $\Gamma_{G,k}$ is proportional to the number of vertices and edges in G, and the each clauses has at most k-literals. It is also easy to see that $\Gamma_{G,k}$ can be constructed from G in time that linear in the size of G. Thus, these observations together establish that $k\text{−}\mathsf{COLOR} \leq_{\mathsf{P}} \mathsf{SAT}$.

Example A.45 A formula φ in CNF is said to be in 3-CNF if every clause in φ has exactly 3 literals. For example, $(x_1 \vee \neg x_2 \vee \neg x_3) \wedge (\neg x_1 \vee \neg x_4) \wedge (x_4) \wedge (\neg x_2 \vee \neg x_3 \vee x_4)$ is not in 3-CNF, while $(x_1 \vee \neg x_2 \vee \neg x_3) \wedge (\neg x_1 \vee x_4 \vee x_2)$ is in 3-CNF. Recall the SAT problem is one whether given a formula φ in CNF, we need to determine if φ is satisfiable. A special case of this problem is one where the input formula is promised to be in 3-CNF. Formally we have,

$$3\text{−}\mathsf{SAT} = \{\langle\varphi\rangle \mid \varphi \text{ is in 3-CNF and is satisfiable}\}.$$

Since 3−SAT is a "special" version of SAT, the identity function is a reduction from 3−SAT to SAT; thus, $3\text{−}\mathsf{SAT} \leq_{\mathsf{P}} \mathsf{SAT}$. It turns out the one also has a reduction the other way around.

The reduction from SAT to 3−SAT is as follows. Consider a CNF formula φ; it will be convenient to this of φ as a set of clauses. Our reduction will convert (in polynomial time) each clause $c \in \varphi$ into a 3-CNF formula $f(c)$ such that c and $f(c)$ are satisfied by (almost) the same set of truth assignments. Then $f(\varphi) = \{f(c) \mid c \in \varphi\}$, and it will be the case that φ is satisfiable iff $f(\varphi)$ is satisfiable.

Let us now describe the translation of clauses. The translation of clause c will depend on how many literals c has. Let $c = \ell_1 \vee \vee \ell_2 \vee \cdots \vee \ell_k$. Depending on k, we have the following cases.

Case $k = 1$ Let u and v be "new" propositions not used before. Define $f(c)$ to be

$$(\ell_1 \vee u \vee v) \wedge (\ell_1 \vee u \vee \neg v) \wedge (\ell_1 \vee \neg u \vee v) \wedge (\ell_1 \vee \neg u \vee \neg v)$$

Case $k = 2$ Let u be a "new" proposition. $f(c)$ is given by

$$(\ell_1 \vee \ell_2 \vee u) \wedge (\ell_1 \vee \ell_2 \vee \neg u)$$

Case $k = 3$ In this case $f(c) = c$.
Case $k > 3$ Let $y_1, y_2, \ldots y_{k-3}$ be new propositions. Then $f(c)$ is

$$(\ell_1 \vee \ell_2 \vee y_1) \wedge (\ell_3 \vee \neg y_1 \vee y_2) \wedge (\ell_4 \vee \neg y_2 \vee y_3) \wedge \cdots$$
$$\wedge (\ell_{k-2} \vee \neg y_{k-4} \vee y_{k-3}) \wedge (\ell_{k-1} \vee \ell_k \vee \neg y_{k-3})$$

It is easy to see that f can be computed in time that is linear in the size of φ. Moreover, φ is satisfiable iff $f(\varphi)$ is satisfiable (left as exercise). Thus, f is a polynomial time reduction showing $\mathsf{SAT} \leq_{\mathsf{P}} 3\!-\!\mathsf{SAT}$.

Polynomial time reductions satisfy properties similar to many-one reductions: they are transitive and if A reduces to B then $\overline{A}$ reduces to $\overline{B}$.

Proposition A.46 *The following properties hold for polynomial time reductions.*

- *If* $A \leq_{\mathsf{P}} B$ *then* $\overline{A} \leq_{\mathsf{P}} \overline{B}$.
- *If* $A \leq_{\mathsf{P}} B$ *and* $B \leq_{\mathsf{P}} C$ *then* $A \leq_{\mathsf{P}} C$.

Proof Detailed proof of these observations is left as an exercise. But the sketch is as follows. If f is a polynomial time reduction from A to B then f is also a polynomial time reduction from $\overline{A}$ to $\overline{B}$. And if f is a polynomial time reduction from A to B and g is a polynomial time reduction from B to C, then $g \circ f$ is a polynomial time reduction from A to C. □

Finally, polynomial time reductions do serve as a way to compare the computational difficulty of two problems. We show that if $A \leq_{\mathsf{P}} B$ and B is "easy" then A is easy.

Theorem A.47 *If* $A \leq_{\mathsf{P}} B$ *and* $B \in \mathsf{P}$ *then* $A \in \mathsf{P}$.

Proof Let f be a polynomial time reduction from A to B and let M be a deterministic polynomial time algorithm recognizing B. Then the polynomial time algorithm N for A does the following: On input w, compute $f(w)$ and then run M on $f(w)$. It is easy to see that N recognizing A from the properties of a reduction.

The tricky step is to argue that N runs in polynomial time. Let us assume that f is computed in time n^k and let M run in time n^ℓ. Since f can be computed in time n^k, it means that $|f(w)| \leq |w|^k$; this is because a single step in the computation of f can produce at most one bit of $f(w)$. Therefore, the total running time of N is $|w|^k$ (time to compute $f(w)$) + $(|w|^k)^\ell$ (time to run M on $f(w)$ which is a string of length $|w|^k$). This is bounded by $O(n^{k\ell})$ which is polynomial. □

In Theorem A.47, we could have replaced P by any of the complexity classes in Fig. A.3 that contain P, and the proof would go through. Thus, polynomial time reductions are an appropriate lens by which measure the relative difficulty of problems that belong to complexity classes that contain P.

Definition A.48 (Hardness and Completeness)

Let $\mathcal{C}$ be a complexity class in Fig. A.3 that contains P. A is said to be $\mathcal{C}$*-hard* iff for every $B \in \mathcal{C}$, $B \leq_{\mathsf{P}} A$.

A is said to be $\mathcal{C}$*-complete* iff $A \in \mathcal{C}$ and A is $\mathcal{C}$-hard.

In other words, informally, a problem is $\mathcal{C}$-hard if it is at least as difficult as any problem in $\mathcal{C}$. It is $\mathcal{C}$-complete if in addition it also belongs to $\mathcal{C}$. Fixing $\mathcal{C}$ to be NP, we could say that a problem is NP-complete if it is the "hardest" problem that belongs to NP. Because of their status as the most difficult problems in NP, they are candidate problems to study to help resolve the P versus NP question. This is captured by the following observation.

Proposition A.49 *If A is* NP*-hard and $A \in \mathsf{P}$ then* $\mathsf{NP} = \mathsf{P}$.

Proof Consider any problem $B \in \mathsf{NP}$. Since $B \leq_{\mathsf{P}} A$ and $A \in polytime$, by Theorem A.47, we have $B \in \mathsf{P}$. □

In the absence of a firm resolution of the P versus NP questions, classifying a problem as NP-hard suggests that it is unlikely that the problem has a polynomial time algorithm given our belief that $\mathsf{P} \neq \mathsf{NP}$.

Many natural problems are NP-complete. The historically (and pedagogically) first problem known to be NP-complete is SAT (Cook-Levin Theorem Theorem 1.23).

Observe that from Fig. A.3 , we have $\mathsf{P} \subseteq \mathsf{NP}$ and $\mathsf{P} \subseteq \mathsf{coNP}$. From this we can conclude that $\mathsf{P} \subseteq \mathsf{NP} \cap \mathsf{coNP}$. Related but independent of the P versus NP question is whether $\mathsf{P} \overset{?}{=} \mathsf{NP} \cap \mathsf{coNP}$. This question also remains open. Many problems that were previously known to be in $\mathsf{NP} \cap \mathsf{coNP}$ were proved to be in P years later. Two classical examples are Linear programming that was shown to be in P by Khachiyan in 1979 and testing whether a number is prime, which was proved by Agarwal-Kayal-Saxena in 2002 to be in P. However, there are some natural problems in $\mathsf{NP} \cap \mathsf{coNP}$ whose status with respect to P is still unresolved. One is the problem of solving parity games, and the other is the factoring problem.